Programowanie
przed 8 wykładem

Andrzej Giniewicz

26.04.2024

Dotychczas poznaliśmy jeden typ metod i atrybutów. Nazywaliśmy je po prostu „metodami" i „atrybutami", podczas gdy precyzyjniej nazywają się one „publicznymi metodami instancji" oraz „publicznymi atrybutami instancji". Słowo „instancja" jest tutaj rodzajem metody, które omówimy w pierwszej sekcji, natomiast „publiczny" jest modyfikatorem dostępu, które omówimy w sekcji drugiej. Dowiemy się, jak za pomocą metod zaimplementować atrybuty, korzystając z konceptu właściwości. Na koniec omówimy jak dodawać adnotacje o typach do kodu Pythona oraz jak tworzyć tak zwane abstrakcyjne klasy bazowe.

1 Rodzaje metod

Drugi obok atrybutów instancji rodzaj atrybutów dostępny w Pythonie to atrybuty klas. Atrybuty instancji ustawiane są wewnątrz metod, najczęściej wewnątrz konstruktora, zawsze z wykorzystaniem odwołania do obiektu

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

Zwróćmy uwagę, że ustawiając lub dostając się do atrybutu instancji, piszemy `self.x`, co podkreśla, że jest on unikatowy dla każdej instancji, czyli dla każdego obiektu. Jeśli klasa to `Punkt`, każdy konkretny obiekt tej klasy, ma swoje współrzędne. Współrzędne nie są zatem cechą wspólną dla wszystkich obiektów, tylko cechą każdego obiektu, indywidualną dla każdego.

Python pozwala na ustalanie atrybutów dla całej klasy. Jeśli skorzystamy z atrybutów klasy, wszystkie obiekty będą miały dostęp do tego samego (a nie takiego samego) atrybutu, ponieważ będzie cechą współdzieloną. Atrybuty klas definiujemy, podając je na równi z metodami.

```
class Punkt:
    liczba_punktów = 0
```

```
    def __init__(self, x, y):
        self.x = x
        self.y = y
        Punkt.liczba_punktów += 1
```

Początkowa wartość atrybutu `liczba_punktów` jest ustalana, gdy deklarowana jest klasa. Następnie przy każdym wywołaniu konstruktora, wartość ta zwiększana jest o 1. Zwróćmy uwagę, że nie piszemy `self.liczba_punktów`, gdzie `self` to odniesienie do konkretnego obiektu, tylko `Punkt.liczba_punktów`, gdzie `Punkt` to nazwa klasy.

Do atrybutów klasy możemy dostać się również przez obiekt, przykładowo

```
a = Punkt(1, 2)
print(a.liczba_punktów)
b = Punkt(0, 0)
print(a.liczba_punktów)
print(b.liczba_punktów)
print(Punkt.liczba_punktów)
```

po wykonaniu powyższego kodu, na ekranie pojawią się cztery liczby, przykładowo 1, 2, 2, 2, jeśli a jest pierwszym utworzonym punktem. Pomimo, że możemy w ten sposób pobrać wartość atrybutu klasy w tym przypadku, nie zawsze jest to możliwe. Jeśli zdefiniowany jest jednocześnie atrybut instancji i klasy, to musimy poprzedzić atrybut klasy nazwą klasy, ponieważ w innym przypadku dostaniemy się do atrybutu instancji.

```
class A:
    x = None
    def __init__(self, x):
        self.x = x  # tworzymy i ustawiamy atrybut instancji
        A.x = x+1   # ustawiamy atrybut klasy

a = A(3)
a.x == 3  # wartość atrybutu instancji
A.x == 4  # wartość atrybutu klasy
```

Warto zwrócić uwagę też, że nieważne ile obiektów utworzymy, atrybuty klasy zajmują tyle samo miejsca, podczas gdy dla atrybutów instancji, ilość zajętego miejsca rośnie liniowo z liczbą obiektów, ponieważ każdy obiekt otrzymuje swoje własne miejsce na ich przechowywanie.

Do przykładowych użyć atrybutów klas można zaliczyć:

- zliczanie lub przechowywanie innych informacji o wszystkich obiektach danej klasy,
- miejsce dla stałych specyficznych dla danej klasy,
- sposób ustalania wartości domyślnych dla większości instancji,
- alternatywę dla dowolnych wartości globalnych.

Najważniejszym zastosowaniem, zawierającym technicznie również pozostałe, jest ostatnie — używanie zmiennych globalnych jest zwykle postrzegane jako zła praktyka programistyczna, zatem dostępność prostej alternatywy, polegającej na umieszczeniu ich w klasie, jest dobrym rozwiązaniem.

Obok atrybutów klas, podobną funkcję pełnią metody klas. Metody klas mogą być uruchomione dla klasy, nie tylko dla obiektu jak dzieje się to w przypadku metod instancji. Aby stworzyć metodę klasy, musimy ją udekorować za pomocą dekoratora `classmethod`. Konwencja nakazuje wtedy użyć nazwy `cls` zamiast `self` jako pierwszego argumentu.

```
class Osoba:
    anonimów = 0

    def __init__(self, imię):
        self.imię = imię

    @classmethod
    def anonim(cls):
        cls.anonimów += 1
        return cls("<anonim>")

    def kto_to(self):
        print(self.imię)

a = Osoba("Alojzy")
b = Osoba.anonim()
a.kto_to()  # Alojzy
b.kto_to()  # <anonim>
Osoba.anonimów == 1
```

Dla obiektu `a` w kodzie powyżej nie dzieje się nic nowego, są to zwykłe metody oraz atrybuty instancji. Co się jednak dzieje, gdy piszemy `Osoba.anonim()`? Python wywołuje metodę klasy `anonim` i jako pierwszy argument metody przekazuje klasę (`Osoba`). Dzięki temu w metodzie klasy mamy dostęp do atrybutów klasy (`cls.anonimów`) oraz możliwości tworzenia obiektów (`cls(...)`). W metodzie tej nie mamy jednak możliwości dostępu do atrybutów instancji oraz metod instancji, co jest podkreślone brakiem argumentu `self`.

Częstym zastosowaniem metod klas są alternatywne konstruktory jak w przykładzie powyżej. `Osoba.anonim()` tworzy nową anonimową osobę i zwraca obiekt, pełni więc rolę drugiego wariantu konstruktora. Innym przykładowym zastosowaniem może być manipulacja atrybutów klas, używanych do zliczania lub innej formy przechowywania informacji o obiektach. Przykładowo w klasie `Punkt` definiowanej wcześniej możemy zdefiniować metodę klasy resetującą liczbę punktów do zera.

```
class Punkt:
    liczba_punktów = 0
```

```
    def __init__(self, x, y):
        self.x = x
        self.y = y
        Punkt.liczba_punktów += 1

    @classmethod
    def reset(cls):
        cls.liczba_punktów = 0
```

Podobnie jak w przypadku atrybutów klas, jeśli nie ma metody instancji o tej samej nazwie, do metody klasy możemy odwołać się za pomocą klasy lub obiektu. Niezależnie jednak od sposobu wywołania, pierwszym argumentem wywołania zawsze będzie klasa.

Metody klas możemy widzieć jako sposób ograniczenia dostępu, ponieważ definiujemy metodę, która nie ma dostępu do atrybutów i metod instancji, zachowuje jednak dostęp do atrybutów klas. Kolejnym stopniem izolacji są metody statyczne. Metody statyczne nie posiadają odniesienia ani do klasy, ani do instancji. Aby stworzyć metodę statyczną, używamy dekoratora `staticmethod`. Metody statyczne nie mają z góry określonego pierwszego argumentu metody, ponieważ nie muszą otrzymać przez niego ani klasy ani obiektu.

```
class Osoba:
    @staticmethod
    def pełnoletnia(wiek):
        return wiek>18

if not Osoba.pełnoletnia(14):
    print("Dowodzik poproszę")
```

Wykorzystanie metod statycznych na pierwszy rzut oka nie jest przydatne, ponieważ konkurują z funkcjami, jeśli jednak umieścimy je w klasie, mamy możliwość przedefiniowywania ich w klasach dziedziczących z pisanej klasy. W porównaniu z metodami klas do zalet metod statycznych możemy zaliczyć też większą szybkość, ponieważ Python nie musi zarządzać stanem ani przekazywać instancji obiektu lub klasy. Pojawienie się też statycznej metody jest jednoznacznym komunikatem dla użytkownika kodu, że dana metoda na pewno nie modyfikuje stanu. Dobrą praktyką jest wybór takiego rodzaju metod, aby komunikować, jakie możliwe zmiany mogą zajść po ich użyciu.

2 Modyfikatory dostępu metod

Niektóre języki programowania dają więcej kontroli dla implementowanych metod — można w nich określić, czy metoda jest publiczna, chroniona lub prywatna. Metody publiczne mogą być użyte z każdego miejsca kodu, chronione tylko w klasie definiującej metodę oraz klasach z niej dziedziczących, natomiast prywatne, mogą być używane jedynie w klasach definiujących daną metodę. Python jest językiem dynamicznym, w którym introspekcja, czyli wgląd do wnętrza obiektu, jest istotnym elementem języka. Oznacza to, że

w ostateczności nie da się określić innych metod niż publiczne — jeśli ktoś chce skorzystać z jakiejś metody lub atrybutu, w Pythonie zawsze jest to możliwe. Niemniej jednak istnieje konwencja, określająca metodę określania intencji zakresu stosowalności danej metody lub atrybutu.

Jeśli nazwa w klasie zaczyna się od pojedynczego znaku podłogi, traktowana jest jako zmienna chroniona. Będzie bez trudu dostępna, ale istnieje umowa zawarta w definicji języka, mówiąca, że tej zmiennej nie powinniśmy wykorzystywać wprost. Zintegrowane środowiska programistyczne, notatniki i interpretery Pythona respektują tę zasadę i gdy używamy tabulacji do auto-uzupełniania po napisaniu `obiekt.` i naciśnięciu tabulacji nie zobaczymy metod chronionych. Zobaczymy je dopiero po napisaniu `obiekt._` i naciśnięciu tabulacji, czyli po bezpośrednim poinformowaniu Pythona, że chcemy, aby pokazał metody chronione.

W przypadku metod prywatnych jest podobnie, ale tutaj oprócz umowy mamy niewielką pomoc od Pythona w postaci mieszania nazw (ang. *name mangling*). Mieszanie nazw to automatyczne uzupełnianie nazwy, aby różniła się dla różnych klas, dzięki czemu przy dziedziczeniu, wartości prywatne nie będą się przedefiniowywać, tylko pozostaną prywatne dla klasy. Wartości prywatne definiujemy, używając symbolu dwóch podłóg na początku nazwy oraz braku dwóch podłóg na końcu nazwy[1]. Jeśli Python zobaczy na przykład atrybut `__x` w kasie `A`, zmieni wszystkie jego wystąpienia na `_A__x`. Możemy się o tym przekonać, uruchamiając kod poniżej, prezentujący trzy atrybuty klasy o różnych dostępach

```
class A:
    public = 0
    _protected = 1
    __private = 2

print(A.public)
print(A._protected)  # nie podpowiada przez [Tab]
print(A._A__private) # nie podpowiada przez [Tab], zmieniona nazwa
```

Kod ten zadziała i wypisze kolejno 0, 1, oraz 2.

Niezależnie od wyboru dostępu, czyli od tego, czy atrybut lub metoda jest publiczna, chroniona lub prywatna, ostatecznie jest to zabezpieczenie jedynie przez konwencję, zatem konieczny jest pewien poziom zaufania do użytkowników naszego kodu.

3 Właściwości

Niekiedy chcemy, aby podczas ustawiania lub odczytywania pewnych atrybutów, uruchamiał się dodatkowy kod. Standardowym zastosowaniem jest podawanie różnych atrybutów, które są ze sobą powiązane, jak promień, obwód i pole koła. Aby stworzyć klasę mającą trzy atrybuty, które są powiązane, możemy skorzystać z właściwości, które tworzymy dekoratorem `property` oraz dekoratorów postaci `metoda.setter`.

[1] Zestaw dwóch podłóg na początku i na końcu jest zarezerwowany dla metod magicznych.

```
import math

class Koło:
    def __init__(self, r=0):
        self.promień = r

    @property
    def promień(self):
        return self._promień

    @promień.setter
    def promień(self, r):
        self._promień = r

    @property
    def obwód(self):
        return 2*math.pi*self._promień

    @obwód.setter
    def obwód(self, l):
        self._promień = l/(2*math.pi)

    @property
    def pole(self):
        return math.pi*self._promień**2

    @pole.setter
    def pole(self, p):
        self._promień = math.sqrt(p/math.pi)
```

Zwróćmy uwagę na fakt, że każda metoda we właściwościach w naszym przykładzie występuje dwa razy. Zanim omówimy ten kod, spróbuj go uruchomić, pisząc na przykład

```
k = Koło(r=2)
print(k.promień)  # 2
print(k.obwód)    # 12.566370614359172
print(k.pole)     # 12.566370614359172
k.promień = 1
print(k.promień)  # 1
print(k.obwód)    # 6.283185307179586
print(k.pole)     # 3.141592653589793
k.obwód = 1
print(k.promień)  # 0.1591549430918953 5
print(k.obwód)    # 1
```

```
print(k.pole)      # 0.07957747154594767
k.pole = 1
print(k.promień)   # 0.5641895835477563
print(k.obwód)     # 3.5449077018110318
print(k.pole)      # 0.9999999999999999
```

Obliczenia powyżej możemy stosunkowo łatwo sprawdzić, ponieważ opierają się w pełni na wzorach $l = 2\pi r$ oraz $p = \pi r^2$. Pomijając zaokrąglenia numeryczne, wszystkie trzy wartości uaktualniają się prawidłowo po zmianie dowolnej z nich. W rzeczywistości, gdy piszemy `k.pole`, nie korzystamy wprost z atrybutu, tylko wywołujemy metodę o nazwie `pole` oznaczoną dekoratorem `property`. Gdy piszemy `k.pole = 1`, wywołujemy metodę o nazwie `pole`, oznaczoną dekoratorem `pole.setter`. W ten możemy przeliczyć wszystkie trzy warianty i zapamiętać w obiekcie jedynie promień, zapisany w chronionym atrybucie instancji `_promień`.

Właściwości powodują, że metody instancji działają jak atrybuty instancji. Nasuwa się pytanie, czy istnieje możliwość stworzenia właściwości klas. Odpowiedź ta jest twierdząca, ale poziom jej skomplikowania zależy od wersji Pythona. W Pythonie od 3.9 w górę, dekorator `classmethod` oraz `property` można łączyć, wypisując je jeden po drugim[2]. We wcześniejszych wersjach Pythona konieczne było zastosowanie meta-klas, czyli klas klas[3].

4 Adnotacje typów

Począwszy od wersji 3.5 Python ma możliwość dodawania adnotacji zawierających informacje o typach[4]. Dokument PEP484[5], został napisany przez trzy osoby — byli to Guido van Rossum (twórca języka Python), Łukasz Langa (jeden z developerów pracujących w pełnym wymiarze dla fundacji PSF zajmującej się rozwojem Pythona) oraz Jukka Lehtosalo (twórca programu MyPy). W dokumencie tym gwarantują, że Python zawsze był i będzie językiem dynamicznym, jednak między innymi dzięki programowi MyPy, na horyzoncie pojawiły się potencjalne zastosowania dla elementów języka pozwalających na dopisywanie informacji, jakiego typu wartości może przyjmować wartość w Pythonie. Istnieją trzy główne zastosowania takiego narzędzia:

1. Dokumentacja dla programisty, który widząc informacje o typie, od razu wie, jakie wartości może przekazać do funkcji i jakiego wyniku się spodziewać, bez analizowania kodu funkcji;

[2]Patrz `https://docs.python.org/3.9/library/functions.html#classmethod` oraz `https://github.com/python/cpython/pull/27115`.

[3]W Pythonie wszystko jest obiektem, nawet klasy. Klasy klas nazywane są meta-klasami. Zainteresowanym meta-klasami, polecam rozdział 24 drugiego wydania książki „Fluent Python" autorstwa Luciano Ramalho. Książka jest dostępna przez platformę Safari `https://biblioteka.pwr.edu.pl/e-zasoby/platformy/oreilly-safari`.

[4]Patrz `https://peps.python.org/pep-0484/`.

[5]PEP to skrót od *Python Enhancement Proposal* — są to dokumenty, w których proponowane są nowe funkcjonalności języka, zanim zostaną dodane do standardowej implementacji CPython.

2. Informacja dla środowiska programistycznego, które może podpowiadać odpowiednią wersję funkcji oraz metod. Jeśli na przykład środowisko graficzne wie, że metoda `upper()` dla napisów zwraca napis, gdy napiszemy

   ```
   napis.upper().
   ```

   i naciśniemy tabulację po kropce, edytor będzie w stanie podpowiedzieć listą wszystkich dostępnych metod dla napisów, na przykład `count`. Bez informacji, że wynik działania metody `upper` to napis, edytory nie są w stanie udzielić nam pomocy w tej sytuacji;

3. Możliwość sprawdzenia typów dla programisty za pomocą analizatora statycznego kodu MyPy. Przykładowo, jeśli napiszemy program `testowy.py`, w którym umieścimy błędny kod

   ```
   "Hello".upper() + 3
   ```

   za pomocą programu MyPy wyłapiemy błąd polegający na tym, że nie można dodać napisu do liczby. Co ważne, dzieje się to bez uruchamiania kodu, tylko dzięki temu, że standardowa biblioteka (w tym metoda `upper`) posiada odpowiednie adnotacje. Aby samodzielnie sprawdzić ten kod, zainstaluj MyPy[6] oraz przetestuj program komendą

   ```
   mypy testowy.py
   ```

Te trzy zastosowania dla adnotacji powodują, że stanowią one wartościowy dodatek do języka Python, choć nie mają one wpływu na wykonanie samego kodu — przekazanie wartości niewłaściwego typu nie jest błędem w Pythonie tak długo, jak niewłaściwa wartość nie spowoduje błędu gdzieś wewnątrz funkcji. Takie rozwiązanie pozwala zachować elastyczność techniki *duck-typing* oraz daje możliwość skorzystania z najważniejszych korzyści statycznego typowania.

Należy pamiętać, że typy statyczne rozwijają się bardzo dynamicznie, od wersji 3.5 do wersji 3.11 nie było praktycznie wydania, które nie dodawałoby nowej funkcjonalności w tym zakresie. Listę obecnych dokumentów PEP dotyczących typowania można zobaczyć w dokumentacji[7].

5 Kilka przykładów

Zacznijmy od prostego przykładu kodu z typami.

```
pi: float = 3.14

def pole(promień: float = 0) -> float:
    return pi*promień**2
```

[6] Przez `pip`, informacje w materiałach po poprzednim wykładzie

[7] Patrz: `https://docs.python.org/3/library/typing.html#relevant-peps`.

Mamy tu kilka nowych elementów. Po pierwsze, w zmiennych po dwukropku możemy dopisać nazwę typu (tutaj `float`). Po drugie, w ten sam sposób możemy dopisać typ do argumentu (pomiędzy nazwę i wartość domyślną). Oba te zastosowania sygnalizujemy dwukropkiem po nazwie zmiennej lub argumentu. Wartość zwracaną dopisujemy po argumentach, ale przed dwukropkiem, sygnalizując jej wystąpienie strzałką `->`.

Zwróćmy uwagę, że jeśli funkcja nie ma instrukcji `return` zwraca `None`, zatem powinniśmy dodać typ

```
def hello() -> None:
    print("Hello World!")
```

Jeśli wykorzystujemy `None` jako wartość domyślną, musimy posłużyć się typem `Optional`. Typ ten nie jest standardowo importowany, zatem musimy to zrobić ręcznie

```
from typing import Optional

def funkcja(x: int, y: Optional[int] = None) -> int:
    if y is None:
        return x//2
    else:
        return x-y
```

Zwróćmy uwagę na to, że typ `Optional` ma argument, który podajemy w nawiasach kwadratowych. `Optional[int]` oznacza, że wartością może być albo wartość typu `int`, albo `None`.

6 Co może pojawić się jako typ?

Typem może być dowolny typ standardowy oraz dowolna klasa. Z przykładów powyżej wiemy jednak, że to nie wyczerpuje listy możliwości, ponieważ zobaczyliśmy typ `Optional`. Moduł `typing` definiuje więcej przydatnych typów. Możemy również wykorzystać tak zwane abstrakcyjne klasy bazowe, które opiszemy w późniejszej sekcji.

Przyjrzyjmy się teraz kilku typom z modułu `typing`. Zacznijmy od `Any`. Typ `Any` wskazuje, że zmienna będzie typowana całkowicie dynamicznie. Jest to sposób na wyłączenie sprawdzania typów. Przykładowym zastosowaniem może być typ dla słownika, którego kluczami są napisy, ale może przechowywać dowolne wartości. Oznaczenie `Any` pozwala na wykorzystanie MyPy z zachowaniem sprawdzania statycznego w innych miejscach programu.

Poznany wcześniej typ `Optional[T]` jest w rzeczywistości skrótem dla `Union[T, None]`. Typ `Union` może być wykorzystany dla większej liczby typów, na przykład

```
from typing import Union

def f(x: Union[int, float, complex]) -> Union[int, float, complex]:
    return 2*x
```

Począwszy od wersji Pythona 3.10 typ `Union[int, float, complex]` można zapisać jako `int | float | complex`, czyli za pomocą operatora `|`.

Począwszy od Pythona 3.9, który jest wymagany na zajęciach, istnieje możliwość zapisania typów generycznych. Takim typem są na przykład listy, krotki i słowniki. Generyczność oznacza, że ich typ ma parametr, który możemy ustawić, podobnie jak w typie `Optional`. I tak od wersji 3.9 legalne są zapisy takie, jak `list[int]`, `tuple[int, int]` lub `set[Union[int, float, complex]]`.

Generyczność pozwala też używać zmiennych typów. Popatrzmy na następujący przykład:

```
from typing import Optional, TypeVar

T = TypeVar('T')

def first(x: list[T]) -> Optional[T]:
    if x:
        return x[0]
    else:
        return None
```

Tu zaczyna się już coś dziać. Zmienna `T` jest zmienną typu. Jeśli użytkownik przekaże jako argument listę `list[int]`, to zmienna `T` będzie równa `int`, a wynik działania funkcji zostanie ustalony na `Optional[int]`.

7 Abstrakcyjne klasy bazowe

Abstrakcyjne klasy bazowe (nazywane też klasami ABC od angielskiego *Abstract Base Class*) są narzędziem w języku Python, które pozwalają sformalizować technikę *duck-typing* oraz dodatkowo, mogą być wykorzystywane w typach. Abstrakcyjne klasy bazowe służą do określenia, jakie metody musi mieć dana klasa, aby móc być konkretnego typu. Abstrakcyjne klasy bazowe dziedziczą z `abc.ABC` oraz deklarują wymagane metody za pomocą dekoratora `@abc.abstractmethod`.

Jeśli przykładowo chcemy stworzyć klasę `Zwierzę`, która obligatoryjnie ma mieć metodę `daj_głos`, napiszemy

```
from abc import ABC, abstractmethod

class Zwierzę(ABC):
    @abstractmethod
    def daj_głos(self):
        pass
```

Zwróćmy uwagę, że metoda `daj_głos` ma implementację, choć w tym przypadku nic ona nie robi. Dekorator `abstractmethod` może być łączony z wcześniej poznanymi dekoratorami metod, czyli `classmethod`, `staticmethod` oraz `property`. Osoby znające inne języki programowania mogą rozpoznać metody abstrakcyjne, ponieważ działają podobnie, jak występujące na przykład w języku C++ czy Java. Niemniej jednak, metody abstrakcyjne w Pythonie mogą mieć implementację domyślną, do której możemy mieć dostęp za pomocą słowa kluczowego `super()` tak samo, jak do każdej implementacji w rodzicu.

Jeśli stworzymy nową klasę dziedziczącą z klasy ABC i zapomnimy dodać implementację metody abstrakcyjnej, dziedzicząca klasa również będzie abstrakcyjna.

```
class Pies(Zwierzę):
    def obsikaj_latarnię(self):
        pass  # TODO: zaimplementować
```

Python nie pozwoli nam utworzyć instancji tej klasy, gdy napiszemy

```
fafik = Pies()  # tu pojawi się błąd
```

Jeśli natomiast zaimplementujemy brakującą metodę, tworzenie obiektu tej klasy zadziała.

```
class Pies(Zwierzę):
    def daj_głos(self):
        print("hau hau")

    def obsikaj_latarnię(self):
        pass  # TODO: zaimplementować

fafik = Pies()  # teraz zadziała
```

Drugim sposobem na stworzenie klasy, która realizuje abstrakcyjną klasę, jest jej rejestracja, przez co utworzymy relację wirtualnego dziedziczenia. Wirtualne dziedziczenie nie pojawia się w procedurze wyznaczania kolejności metod (MRO), zatem nie zadziała `super()` i nie będzie dostępu domyślnej implementacji metody abstrakcyjnej. Niemniej jednak funkcje `isinstance` oraz `issubclass` działają też dla wirtualnego dziedziczenia.

```
class Kot:
    def daj_głos(self):
        print("miau miau")

    def zniszcz_tapicerkę(self):
        pass  # TODO: zaimplementować

Zwierzę.register(Kot)

mruczek = Kot()
```

W obu przypadkach sprawdzenia za pomocą `issubclass` oraz `isinstance` działają zgodnie z oczekiwaniami. W kodzie poniżej wszystkie cztery wywołania zwracają `True`.

```
issubclass(Kot, Zwierzę)
issubclass(Pies, Zwierzę)
isinstance(fafik, Zwierzę)
isinstance(mruczek, Zwierzę)
```

Druga metoda rejestracji pozwala dodatkowo sprawić, że klasa zdefiniowana w innym miejscu przedstawia się jako nasza klasa, pomimo że nie dziedziczy z niej bezpośrednio. Dzięki temu można było zdefiniować w Pythonie wiele klas bazowych dla już istniejących typów, bez ich modyfikacji. Omówimy teraz dwie rodziny standardowych klas ABC.

7.1 Standardowe klasy ABC

W module `collections.abc` znajduje się szereg abstrakcyjnych klas bazowych[8] dla kolekcji, czyli typów takich jak lista, słownik czy zbiór. Zdefiniowany jest tam na przykład typ `Sequence`, który dziedziczy z typów `Reversible` i `Collection`, przy czym ma dwie metody abstrakcyjne: `__getitem__` oraz `__len__`. Typ ten możemy wykorzystać na kilka sposobów.

Po pierwsze, możemy wykorzystać go, aby upewnić się, czy dostaliśmy odpowiedni obiekt jako argument.

```
from collections.abc import Sequence

issubclass(list, Sequence)  # True
issubclass(tuple, Sequence) # True
issubclass(set, Sequence)   # False
```

W powyższym przykładzie lista i krotka są sekwencjami, natomiast zbiór nie jest sekwencją, ponieważ nie ma metody `__getitem__`, która służy do wybierania elementu na n-tej pozycji.

Drugim sposobem wykorzystania typów z modułu `collections.abc` jest tworzenie własnych klas sekwencji. Jeśli dziedziczymy z `Sequence` musimy zaimplementować dwie metody abstrakcyjne, ale w zamian dostaniemy gratis implementację metod `__contains__`, `__iter__`, `__reversed__`, `index` oraz `count`. Przykładowo

```
from collections.abc import Sequence

class Następnik(Sequence):
    def __init__(self, ile=0):
        self.ile = ile
    def __len__(self):
        return self.ile
    def __getitem__(self, n):
        if 0 <= n < self.ile:
            return n+1
        else:
```

[8] Patrz: https://docs.python.org/3/library/collections.abc.html#collections-abstract-base-classes.

```
                raise IndexError("Indeks poza granicami")

lista = Następnik(10)
print(lista[3])
print(lista.count(7))
print(list(reversed(lista)))
```

Spróbuj wpisać powyższy kod do Pythona i przekonaj się, jak działa. Zwróć uwagę, że metody `count` nie implementowaliśmy, tak samo jak iteratora `__reversed__`.

Trzecim częstym zastosowaniem abstrakcyjnych klas bazowych jest wykorzystanie ich w typach. Przykład z wcześniej

```
from typing import Optional, TypeVar

T = TypeVar('T')

def first(x: list[T]) -> Optional[T]:
    if x:
        return x[0]
    else:
        return None
```

możemy udoskonalić

```
from typing import Optional, TypeVar
from collections.abc import Sequence

T = TypeVar('T')

def first(x: Sequence[T]) -> Optional[T]:
    if x:
        return x[0]
    else:
        return None
```

dzięki czemu będzie działał nie tylko dla list, ale dla dowolnych typów spełniających wymogi, aby być sekwencją (ważna jest dla nas metoda `__getitem__`, ponieważ korzystamy z niej we fragmencie `x[0]` oraz metoda `__len__`, ponieważ jest użyta podczas sprawdzenia `if x:`).

Drugą pulą standardowych klas ABC obok `containers.abc` jest biblioteka `numbers`, która zawiera hierarchię pomiędzy typami numerycznymi (na przykład „każda liczba rzeczywista jest też zespolona z częścią urojoną zero") za pomocą abstrakcyjnych klas bazowych. Zachęcam do rzucenia okiem na dokumentację[9].

[9] Patrz: `https://docs.python.org/3/library/numbers.html`.