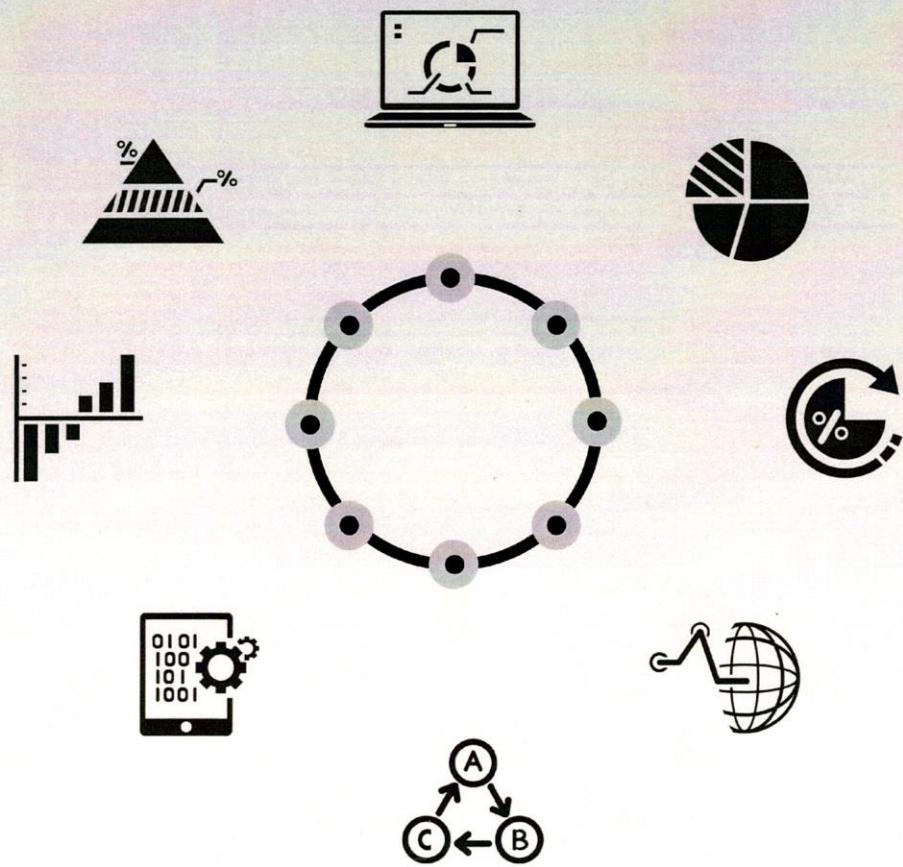


Marek Gągolewski
Maciej Bartoszuk
Anna Cena

Przetwarzanie i analiza danych w języku Python



```
import doctest
doctest.testmod()
```

Przykładowe wywołanie:

```
$ /opt/anaconda3/bin/python test4b.py
*****
File "test4b.py", line 11, in __main__.dzielenie
Failed example:
    dzielenie(4, 16, True) # niepoprawny wynik
Expected: 0.25
Got: 0
*****
1 items had failures:
 1 of  4 in __main__.dzielenie
***Test Failed*** 1 failures.
```

Poprawienie ostatniego warunku testowego zostawiamy jako ćwiczenie. Odnosimy się do tego, że jeśli wszystkie testy jednostkowe wykonują się poprawnie, uruchomienie skryptu nie zakończy się wygenerowaniem żadnego wyniku – to znak, że wszystko jest w porządku. Innym wariantem poznania pakietem do wykonywania testów jednostkowych jest `unittest`.

W kolejnym rozdziale omówimy podstawy programowania obiektowego. Tworzenie nowych klas (typów) stanowi inny sposób radzenia sobie ze złożonością projektów matematycznych. Ponadto przedstawione zagadnienia pozwolą nam jeszcze lepiej zrozumieć, w jaki sposób świadomie *używać* obiekty dobrze poznanych już typów (czyli tablice, ramek danych, list itd.).

PROGRAMOWANIE OBIEKTOWE

16

Wiemy już, jak lepiej organizować kod, zamkując poszczególne funkcje w modułach i pakietach języka Python. Teraz przedstawimy zupełnie inne podejście, które także pozwala sprawić, że nadaje się on do ponownego użycia: programowanie obiektowe. To ważne, obie techniki są od siebie niezależne – możemy nie tylko tworzyć definicje nowych klas w osobnych modułach, ale i bezpośrednio w notatnikach Jupyter.

Programowanie obiektowe polega na tworzeniu nowych *klas* (typów), które grupują dane (struktury, stany obiektów) i funkcje (działania na tychże obiektach), zwane tutaj, odpowiednio, *polami* i *metodami*. Takie podejście do programowania sprawia, że duże projekty informatyczne są łatwiejsze do zaprojektowania, zrozumienia i utrzymywania. Innymi słowy, dobrze odzwierciedla ono nasz ludzki sposób postrzegania rzeczywistości, w której operujemy na różnych obiektach, z których każdy potencjalnie cechuje się innymi właściwościami. Ponadto ich użytkownicy mogą korzystać z nich znacznie łatwiej niż z osobnych, luźno ze sobą powiązanych funkcji.

Przykładem dobrze znanej nam już klasy jest `DataFrame` z pakietu pandas. Tworząc obiekt typu ramka danych – czy ściślej: instancję klasy `DataFrame` – otrzymujemy obiektu tego rodzaju *byt*, który dość łatwo odróżniemy od danych innego rodzaju (list, tablic, słowników).

```
>>> import pandas as pd
>>> x = pd.DataFrame({"a": [1, 2, 3], "b": [True, False, True]})

>>> x
   a      b
0  1  True
1  2  False
2  3  True
```

Każda ramka danych charakteryzuje się właściwymi sobie cechami (polami), np.:

```
>>> x.shape
(3, 2)
```

Można też wykonać *na niej* różne operacje (metody), np.:

```
>>> x.transpose()
   0      1      2
a  1      2      3
b  True  False  True
```

16.1. Klasy i relacje między nimi

Przegląd technik programowania obiektowego dostępnych w języku Python od jednej prostej klasy, którą następnie będziemy rozbudowywać. Ponadto obok będziemy tworzyć nowe klasy, pozostające z nią w jakiejś relacji.

16.1.1. Definiowanie klasy

Wyobraźmy sobie, że chcemy zamodelować pracę ludzi w pewnej korporacji. Naszym punktem wyjścia będzie najprostsza i najbardziej ogólna klasa o nazwie *Osoba*.

Każda osoba ma swoje imię, nazwisko i jest w pewnym wieku. Atrybuty tego typu nazywamy *polami*. Warto od razu podkreślić różnicę: gdy mówimy *klasą*, to na myśli wyabstrahowaną ideę osoby i to, że ma ona określony zestaw cech, który wykonyuje pewne czynności. Gdy mówimy z kolei o *obiekcie* czy *instancji danej klasy*, mamy na myśli konkretnego jej przedstawiciela, a więc np. Davida Thomasa w wieku 30 lat lub Artura Jacksona (o artystycznym pseudonimie „Dwie Szopy”) w wieku 33.

Pozostaje nam jeszcze zdefiniować czynności, które osoba może wykonywać, określić *metody* w odpowiadającej jej klasie.

Załóżmy, że nasza przykładowa osoba może się tylko przedstawić. Jest to racjonalne założenie: o ile wiemy, że hydraulik może naprawić kran, a programista napisać program, to przy jedynie tak ogólnej wiedzy, jak ta, że mamy do czynienia z jakimś człowiekiem, nie możemy oczekwać od niego zbyt wiele.

Definicja klasy Osoba. Zdefiniujmy więc naszą pierwszą klasę:

```
class Osoba:
    """Klasa reprezentująca osobę i informacje o niej."""

    def __init__(self, imie, nazwisko, wiek):
        """Konstruktor klasy Osoba"""
        self.imie = imie
        self.nazwisko = nazwisko
        self.wiek = wiek

    def przedstaw_sie(self):
        """Przedstawia ładnie daną osobę."""
        print("Nazynam się %s %s i mam %d lat(a)." %
              (self.imie, self.nazwisko, self.wiek))
```

Definicję klasy zaczynamy od słowa kluczowego `class`, po którym następuje nazwa klasy. Zaraz po nim możemy opcjonalnie umieścić napis stanowiący jej dokumentację (*docstring*).

Dalej znajdują się definicje dwóch metod, które bardzo przypominają sposób tworzenia funkcji. Pierwszy parametr każdej z nich zwyczajowo będzie nazywany *self* – jak się zaraz okaże – będzie reprezentował konkretny obiekt, na którym wywołujemy daną metodę.

Metodę `__init__()` nazywamy *konstruktorem*. Celem jej działania jest zainicjowanie pól konkretnej instancji klasy. Nie ma sensu tworzyć osoby, która nie ma swojego imienia, nazwiska i wieku. Dlatego właśnie nasz konstruktor przyjmuje, obok `self`, trzy dodatkowe argumenty.

Odnosząc się do tego, odnotujmy, że nigdzie wcześniej nie zdefiniowaliśmy jawnie, że osoba ma takie pól, jak imię czy wiek. Robimy to „dynamicznie” przez przypisywanie wartości do kolejnych pól argumentu `self`; por. jednak podrozdz. 16.3.

Druga metoda, `przedstaw_sie()`, korzysta ze „wszczepionych” w obiekt `self` informacji i wypisuje na konsolę sympatyczny komunikat.

Tworzenie instancji klasy. Stwórzmy teraz pierwszą osobę (obiekt klasy `Osoba`) i następnie każmy się jej przedstawić:

```
>>> David = Osoba("David", "Thomas", 22)
>>> David.przedstaw_sie()
Nazynam się David Thomas i mam 22 lat(a).
>>> type(David)
<class 'Osoba'>
```

Właśnie w przypadku pierwszej z powyższych instrukcji interpreter języka Python wywołuje konstruktor.

Parametr self metod. Aby lepiej zrozumieć rolę parametru `self`, odnotujmy, że następujące dwie instrukcje są równoważne:

```
>>> David.przedstaw_sie()
Nazynam się David Thomas i mam 22 lat(a).
>>> Osoba.przedstaw_sie(David)
Nazynam się David Thomas i mam 22 lat(a).
```

Dzięki parametrowi `self`, każda metoda „wie”, na rzecz jakiej konkretnej instancji klasy działa. Dzięki temu można odróżnić Davida Thomasa od np. Artura Jacksona, który może być również przedstawicielem naszego uniwersum. Zauważmy także, że druga z powyższych instrukcji przypomina sposób wywoływanego funkcji w tworzonych przez nas modułach. Wywoływanie metod na konkretnym obiekcie (instrukcja pierwsza) to po prostu bardzo wygodny skrót notacyjny.

Dostęp do pól obiektu. Standardowo (por. jednak podrozdz. 16.3), każdy obiekt przechowuje swoje dane w osobnym słowniku. Możemy więc obejrzeć, co dana klasa ma nam do zaoferowania:

```
>>> David.__dict__
{'nazwisko': 'Thomas', 'imie': 'David', 'wiek': 22}
```

Co więcej, w dowolnej chwili możemy odwoływać się do pól obiektu, a także je modyfikować:

```
>>> David.wiek = 23
>>> print(David.wiek)
23
>>> David.przedstaw_sie()
Nazywam się David Thomas i mam 23 lat(a).
```

Da się także dodawać „dynamicznie” nowe pola w już istniejącym obiekcie:

```
>>> David.kobieta = False
>>> print("David to " + ("kobieta" if David.kobieta else "mężczyzna"))
David to mężczyzna
>>> David.__dict__
{'kobieta': False, 'imie': 'David', 'nazwisko': 'Thomas', 'wiek': 23}
```

16.1.2. Dziedziczenie

Jednym z ważniejszych paradymatów programowania obiektowego jest *dziedziczenie*. Dzięki niemu możemy odwzorowywać relacje zależności (zawierania) między klasami.

Wróćmy do naszego przykładu. Gdy precyzowaliśmy, które metody może mieć osoba, powiedzieliśmy sobie, że w tak ogólnej sytuacji nie możemy oczekiwac od niej więcej niż tylko umiejętności przedstawienia się. Założmy jednak, że chcielibyśmy wreszcie reprezentować osobę, która jest pracownikiem wspomnianej wcześniej kompanii. Pracownik, jak można się domyślić, pracuje (jest to metoda). Ponadto niezależnie od tego, czy wykonuje on bardzo odpowiedzialne, czy też proste zadania, pobiera pewną pensję (jej wysokość to pole).

Czy oznacza to, że musimy utworzyć kolejną klasę, bardzo podobną do klasy Osoba z takimi samymi polami imie, nazwisko i wiek? Okazuje się, że dzięki mechanizmowi *dziedziczenia* pól i metod wcale nie jest to konieczne. Utwórzmy więc nową klasę Pracownik, której każda instancja jest także Osobą:

```
class Pracownik(Osoba): # dziedziczy po klasie Osoba
    def __init__(self, imie, nazwisko, wiek, pensja):
        super().__init__(imie, nazwisko, wiek)
        self.pensja = pensja

    def pracuj(self):
        print("Przykro mi, ale teraz mam przerwę obiadową.")
```

Odnotujmy, że do metody (tutaj: konstruktora) klasy nadzędnej – czyli tej, po której dziedziczymy – odwołaliśmy się przy użyciu wywołania `super()`. Gdy tworzymy instancję klasy Pracownik, nadal wymagamy jego danych personalnych, ale tym razem potrzebujemy dodatkowo informacji o jego zarobkach.

```
>>> Artur = Pracownik("Artur", "Jackson", 45, 6000)
>>> Artur.przedstaw_sie() # metoda z klasy Osoba
```

```
>>> Artur.pracuj()          // metoda z klasy Pracownik
Przykro mi, ale teraz mam przerwę obiadową.
```

Gdy mamy już „ogólnego” pracownika, chcemy mieć klasy jeszcze bardziej szczegółowe. Niech będzie to np. programista i sprzedawca.

```
class Programista(Pracownik):
    def __init__(self, imie, nazwisko, wiek, pensja, język):
        super().__init__(imie, nazwisko, wiek, pensja)
        self.język = język

    def programuj(self):
        print("Programuję w języku %s." % self.język)

class Sprzedawca(Pracownik):
    def __init__(self, imie, nazwisko, wiek, pensja, produkt):
        super().__init__(imie, nazwisko, wiek, pensja)
        self.produkt = produkt

    def sprzedawaj(self):
        print("Sprzedaję %s." % self.produkt)
```

Stwórzmy przykładowe obiekty powyższych klas:

```
>>> Kasia = Programista("Katarzyna", "Pralinka", 32, 12000, "Python")
>>> type(Kasia)
<class 'Programista'>
>>> Kasia.programuj()
Programuję w języku Python.
>>> Kasia.przedstaw_sie()
Nazywam się Katarzyna Pralinka i mam 32 lat(a).
>>> Kasia.pracuj()
Przykro mi, ale teraz mam przerwę obiadową.
>>> Jaś = Sprzedawca("Jaś", "Cleese", 46, 3000, "Papugi")
>>> type(Jaś)
<class 'Sprzedawca'>
>>> Jaś.sprzedawaj()
Sprzedaję Papugi.
```

Oczywiście zawsze możemy sprawdzić, czy interesujący obiekt pochodzi z danej klasy, uwzględniając dziedziczenie (a więc Programista jest także Pracownikiem i Osobą, choć nie jest już Sprzedawcą).

```
>>> isinstance(Kasia, Programista)
True
>>> isinstance(Kasia, Pracownik)
True
>>> isinstance(Kasia, Osoba)
```

```
>>> isinstance(Kasia, Sprzedawca)
False
>>> isinstance(Kasia, int)
False
>>> isinstance(Kasia, object) # każda klasa dziedziczy po object
True
>>> isinstance(1, int)      # to też jest obiekt jakiejś klasy
True
```

Co więcej, możemy badać relacje między klasami:

```
>>> issubclass(Programista, Pracownik)
True
>>> issubclass(Pracownik, Programista)
False
>>> issubclass(Sprzedawca, Osoba)
True
```

CIEKAWOSTKA

Zwróćmy uwagę na pole `__class__` podanego obiektu:

```
>>> Artur.__class__
<class 'Pracownik'>
>>> Artur.__class__.__name__
'Pracownik'
>>> Artur.__class__.__bases__
(<class 'Osoba'>,)
>>> Artur.__class__.__bases__[0].__bases__
(<class 'object'>,)
>>> Artur.__class__.__bases__[0].__bases__[0].__bases__
()
```

16.2. Metody

Rozszerzmy naszą wiedzę na temat metod.

16.2.1. Przeciążanie metod. Polimorfizm

Zdefiniujmy klasy `Programista` i `Sprzedawca` raz jeszcze. Tym razem zamiast wprowadzania nowych metod `programuj()` i `sprzedawaj()`, przeciążymy metody

```
class Programista(Pracownik):
    def __init__(self, imie, nazwisko, wiek, pensja, język):
        super().__init__(imie, nazwisko, wiek, pensja)
        self.język = język

    def pracuj(self):
        print("Programuję w języku %s." % self.język)

class Sprzedawca(Pracownik):
    def __init__(self, imie, nazwisko, wiek, pensja, produkt):
        super().__init__(imie, nazwisko, wiek, pensja)
        self.produkt = produkt

    def pracuj(self):
        print("Ogólnie to sprzedaję %s." % self.produkt)
        super().pracuj() # wywołaj metodę z klasy bazowej
```

Metoda `pracuj()` jest zdefiniowana w klasach `Pracownik`, `Programista` i `Sprzedawca`, jednak w każdej z nich działa w trochę inny sposób: może być przecież oparta na informacjach zawartych w odmiennych polach obiektu. Czym innym jest bowiem praca programisty, a czym innym – sprzedawcy. Jednak na pewnym poziomie abstrakcji, z punktu widzenia np. dyrektora generalnego, praca to praca – podwładny ma po prostu wykonywać swoje obowiązki. Wydając polecenie „`pracuj!`” (jedno i to samo dla każdego), oczekujemy, że każdy pracownik będzie wiedział, co ma robić.

Rozważmy następującą listę pracowników:

```
firma = [
    Pracownik("Artur", "Jackson", 45, 6000),
    Programista("Katarzyna", "Pralinka", 32, 12000, "Python"),
    Sprzedawca("Jaś", "Cleese", 46, 3000, "Papugi")
]
```

Okazuje się, że gdy wywołamy jedną i tę samą metodę dla każdego Pracownika w firmie, wszyscy będą wykonywali właściwe sobie zadania:

```
for p in firma:
    p.przedstaw_sie()
    p.pracuj()
    print()

Nazywam się Artur Jackson i mam 45 lat(a).
Przykro mi, ale teraz mam przerwę obiadową.

Nazywam się Katarzyna Pralinka i mam 32 lat(a).
Programuję w języku Python.

Nazywam się Jaś Cleese i mam 46 lat(a).
Ogólnie to sprzedaję Papugi,
```

Należy pamiętać, że w tym kodzie nie wprowadziliśmy żadnych nowych metod, a jedynie przekształciliśmy istniejące.

Takie zachowanie się obiektów nazywamy *polimorfizmem*. Dzięki temu mechanizmowi użytkownik musi jedynie zapamiętać jedną nazwę metody. Szczegóły jej działania zależą już od tego, na obiekcie której klasy metoda jest wywołana.

16.2.2. Metody i pola statyczne

Do tej pory wszystkie obiekty, nawet jeśli były one tej samej klasy, działały na właściwych sobie danych. Dostęp do nich zapewniał za każdym razem argument *self* tworzonych przez nas metod. Okazuje się jednak, że pewne informacje i działania mogą być *współdzielone* przez reprezentantów określonego typu.

Metody i pola *statyczne* – bo o takich tutaj będzie mowa – są to takie atrybuty klasy, które występują w jednym egzemplarzu na klasę i do których można się odwoływać bez tworzenia jej instancji.

Typowym przykładem, pokazującym zastosowanie pola statycznego, jest zliczanie ile obiektów danej klasy jest aktualnie w użyciu.

```
class Licznik:
    ile = 0           # pole statyczne

    def __init__(self):      # konstruktor
        Licznik.ile += 1     # odwołanie do pola statycznego
        self.ktory = Licznik.ile
        print("To jest obiekt nr %d." % Licznik.ile)

    def __del__(self):       # destruktör
        Licznik.ile -= 1
        print("Niszczę obiekt nr %d, pozostało jeszcze %d." %
              (self.ktory, Licznik.ile))

    @staticmethod
    def policz():
        return Licznik.ile
```

Stwórzmy teraz kilka obiektów klasy *Licznik*.

```
>>> a = Licznik()
To jest obiekt nr 1.
>>> b = Licznik()
To jest obiekt nr 2.
>>> c = Licznik()
To jest obiekt nr 3.
```

Odnosząc się do obiektów klasy *Licznik*, możemy stwierdzić, że wszystkie obiekty dzielą między sobą to samo pole, *ile*. Z kolei *ktory* jest właściwe każdemu z nich. Usuńmy teraz jeden z obiektów.

```
>>> a = None
Niszczę obiekt nr 1, pozostało jeszcze 2.
```

Przetestujmy także działanie metody statycznej. Okazuje się, że możemy ją wywołać zarówno podając nazwę klasy, jak i konkretnego reprezentanta.

```
>>> Licznik.policz(), b.policz()
(2, 2)
```

Co więcej:

```
>>> Licznik.ile, b.ile, c.ile
(2, 2, 2)
>>> b.ktory, c.ktory
(2, 3)
```

WIEKOWSTKA

Modyfikacja pola statycznego przy użyciu odwołania się do obiektu, a nie klasy, powoduje utworzenie pola właściwego obiektowi, które „przyśłoni” pole statyczne:

```
>>> b.ile = 5
>>> Licznik.ile, b.ile, c.ile
(2, 5, 2)
>>> Licznik.ile = 6
>>> Licznik.ile, b.ile, c.ile
(6, 5, 6)
>>> del b.ile
>>> Licznik.ile, b.ile, c.ile
(6, 6, 6)
```

16.2.3. Metody specjalne

W języku Python istnieje kilkudziesiąt metod specjalnego zastosowania. Mają one zaszczycone nazwy oraz z góry określoną liczbę parametrów. Służą one m.in. do przeciążania różnych operatorów lub wypisywania obiektów na konsolę. Przyjrzymy się tutaj kilku z nich – ich pełen wykaz znajdziemy na stronie dokumentacji (zob. sekcję *Data model*).

Jako przykład w tym punkcie będziemy rozważać klasę reprezentującą czas.

```
class Czas:
    "czas: (godziny, minuty, sekundy)"

    def __init__(self, *args):
        "wieloaspektowy konstruktor"
        if len(args) == 1 and isinstance(args[0], Czas):
            # „konstruktor kopiący”
            self.godziny = args[0].godziny
```

```

        self.minuty = args[0].minuty
        self.sekundy = args[0].sekundy
    else:
        # od 1 do 3 liczb całkowitych
        self.godziny, self.minuty, self.sekundy = 0, 0, 0
        if len(args) == 1:
            self.sekundy = int(args[0])
        elif len(args) == 2:
            self.minuty = int(args[0])
            self.sekundy = int(args[1])
        elif len(args) == 3:
            self.godziny = int(args[0])
            self.minuty = int(args[1])
            self.sekundy = int(args[2])
        else:
            raise Exception("błędne dane")
    self.normalizuj()

def normalizuj(self):
    "przekrć" licznik w razie potrzeby"
    assert self.sekundy >= 0
    assert self.minuty >= 0
    assert self.godziny >= 0

    self.minuty += self.sekundy // 60
    self.sekundy -= (self.sekundy // 60) * 60

    self.godziny += self.minuty // 60
    self.minuty -= (self.minuty // 60) * 60

```

Powyższy konstruktor różnie reaguje w zależności od liczby i typów podanych argumentów.

Metody `__repr__()` i `__str__()`. Po pierwsze, odnotujmy, że domyślny sposób wypisywania na konsoli obiektów typu `Czas` nie jest zbyt ciekawy:

```
>>> Czas(1, 2, 3)
<Czas object at 0x7f6d97f0c6a0>
```

Okazuje się jednak, że jeśli w definicji klasy pojawi się metoda `__repr__()`, to właśnie ona będzie wywoływana przez interpreter w sytuacjach, kiedy potrzebna jest tekstowa reprezentacja obiektu.

```
def __repr__(self): # nowa metoda w klasie Czas
    return ("Czas(%r, %r, %r)" %
           (self.godziny, self.minuty, self.sekundy))
```

Zwracamy tutaj napis, który dostarcza użytkownikowi pełną informację potrzebną do samodzielnego stworzenia równoważnego obiektu.

```

>>> Czas(1, 2, 3)
Czas(1, 2, 3)
>>> Czas(2, 3)
Czas(0, 2, 3)
>>> Czas(3)
Czas(0, 0, 3)

```

Po drugie, możemy także przeciążyć metodę `__str__()`, którą interpreter będzie wykorzystywał, gdziekolwiek zajdzie potrzeba rzutowania obiektu na „ładny” napis, np. w funkcji `print()`; por. też metody `__int__()` i `__float__()`. Dodajmy do definicji klasy `Czas` następującą metodę:

```
def __str__(self): # nowa metoda w klasie Czas
    return ("%02dh%02dm%02ds"
           % (self.godziny, self.minuty, self.sekundy))
```

Dzięki temu mamy, co następuje:

```

>>> t = Czas(1, 2, 3)
>>> print(t) # __str__
01h02m03s
>>> [t, Czas(4, 5, 6)] # __repr__
[Czas(1, 2, 3), Czas(4, 5, 6)]

```

Na marginesie, jeśli `__str__()` nie jest określona jawnie, w jej miejsce jest używana `__repr__()`.

Operatory arytmetyczne i relacyjne. Zastanówmy się, co tak naprawdę się dzieje, gdy stosujemy binarny operator „`+`”. Otóż okazuje się, że zapis:

`a + b`

jest przez interpreter tłumaczony na wywołanie metody `__add__()` obiektu będącego lewym operandem, tj.:

`a.__add__(b)`

Swoje odpowiedniki mają wszystkie inne operatory arytmetyczne, m.in. `__sub__()` (odejmowanie), `__mul__()` (mnożenie), `__div__()` (dzielenie) i `__mod__()` (reszta z dzielenia).

Założymy, że chcielibyśmy mieć możliwość dodawania dwóch obiektów typu `Czas` lub zwiększania aktualnego czasu o podaną liczbę sekund. Tworzymy więc nową metodę:

```
def __add__(self, other): # nowa metoda w klasie Czas
    other = Czas(other)
    godziny = self.godziny + other.godziny
    minuty = self.minuty + other.minuty
    sekundy = self.sekundy + other.sekundy
    return Czas(godziny, minuty, sekundy) # wywoła też normalizuj()
```

Przetestujmy wprowadzone ulepszenia:

```
>>> t1 = Czas(1, 30, 59)
>>> t1 + Czas(3, 13, 0)
Czas(4, 43, 59)
>>> t1 + Czas(120, 1)
Czas(3, 31, 0)
>>> t1 + 1
Czas(1, 31, 0)
>>> t1 + 121
Czas(1, 33, 0)
```

Oprócz np. `__add__()` możemy także przeciążyć metodę `__radd__()` (*augmented addition*). Jeśli użytkownik próbuje wykonać `x + y`, a wywołanie `x.__add__(y)` nie powiedzie się, np.:

```
>>> 1 + t1
Traceback (most recent call last):
TypeError: unsupported operand type(s) for +: 'int' and 'Czas'
wówczas interpreter postara odwołać się do y.__radd__(x).
def __radd__(self, other): # nowa metoda w klasie Czas
    return self.__add__(other)
```

Teraz możliwe są następujące operacje:

```
>>> 1 + t1
Czas(1, 31, 0)
>>> sum([1, 2, 3, Czas(4), 5, 6]) # sum() korzysta z __radd__()
Czas(0, 0, 21)
```

CIEKAWOSTKA

A oto przykład przeciążania funkcji wbudowanej `abs()`:

```
def __abs__(self): # nowa metoda w klasie Czas
    return Czas(self.sekundy)
```

Przykładowe wywołanie:

```
>>> abs(Czas(1, 2, 3))
Czas(0, 0, 3)
```

Podobnie jak operatory arytmetyczne możemy przeciążać operatory relacyjne „`<=`”, „`>`”, „`>=`”, „`==`” i „`!=`”, odpowiednio, przy użyciu `__lt__()`, `__le__()`, `__gt__()`, `__ge__()`, `__eq__()` i `__ne__()` (por. też `__cmp__()`). Na przykład:

```
def __eq__(self, other): # nowa metoda w klasie Czas
    return (self.godziny == other.godziny
            and self.minuty == other.minuty
            and self.sekundy == other.sekundy)

def __lt__(self, other): # nowa metoda w klasie Czas
    # wykorzystamy porządek leksykograficzny dla krotek
    return ((self.godziny, self.minuty, self.sekundy)
            < (other.godziny, other.minuty, other.sekundy))
```

Przetestujmy powyższe:

```
>>> lista = [ Czas(1, 0, 0), Czas(1, 30, 0), Czas(2, 30, 0),
...             Czas(0, 30, 0), Czas(0, 30, 50), Czas(0, 0, 55) ]
...
>>> min(lista) # wymaga jedynie __lt__()
Czas(0, 0, 55)
>>> lista.sort() # wymaga jedynie __lt__()
>>> lista
[Czas(0, 0, 55), Czas(0, 30, 0), Czas(0, 30, 50), Czas(1, 0, 0),
Czas(1, 30, 0), Czas(2, 30, 0)]
>>> lista[0] < lista[1] < lista[2]
True
>>> lista[0] == lista[1]
False
>>> lista[0] != lista[1] # na podstawie __eq__()
True
>>> lista[0] <= lista[1] # niestety
Traceback (most recent call last):
TypeError: unorderable types: Czas() <= Czas()
>>> lista[1] > lista[0] # na podstawie __lt__()
True
```

Operatory łączne. Metodzie `__iadd__()` odpowiada operator „`+=`”.

```
def __iadd__(self, other): # nowa metoda w klasie Czas
    self.godziny += other.godziny
    self.minuty += other.minuty
    self.sekundy += other.sekundy
    if self.sekundy >= 60:
        self.minuty += 1
        self.sekundy -= 60
    if self.minuty >= 60:
        self.godziny += 1
        self.minuty -= 60
    return self
```

Zwrócmy uwagę, że obiekt `self` modyfikujemy w miejscu i zwracamy go jako wynik.

```
>>> t3 = Czas(0, 30, 0)
>>> t4 = Czas(1, 30, 0)
>>> t3 += t4
>>> t3
Czas(2, 0, 0)
```

Tworzenie obiektów iterowalnych. W podrozdziale 3.3 powiedzieliśmy, jak konstruować z obiektów iterowalnych. Aby nasza klasa zapewniała interfejs tego typu, musimy zaimplementować w niej metodę `__iter__()`. Jej zadaniem jest zwracać tzw. iterator – obiekt dowolnej klasy zawierającej definicje dwóch metod:

- `__iter__()` (zwraca `self`);
- `__next__()` (służy do generowania „kolejnych wartości z obiektu, po którym iterujemy”).

Utwórzmy przykładową klasę reprezentującą iteratory odliczające czas:

```
class IterCzas:
    def __init__(self, ile):
        self.ile = ile
        self.aktualny = self.ile

    def __iter__(self):
        self.aktualny = self.ile
        return self

    def __next__(self):
        if self.aktualny <= 0:
            raise StopIteration
        self.aktualny -= 1
        return "Pozostało: %d s." % self.aktualny
```

Zwróćmy uwagę na mechanizm informowania, że skończyła się możliwość generowania kolejnych elementów: zgłaszamy wówczas wyjątek `StopIteration`. Przetestujmy działanie iteratora:

```
>>> list(IterCzas(3))
['Pozostało: 2 s.', 'Pozostało: 1 s.', 'Pozostało: 0 s.']
```

Uzupełnijmy teraz klasę `Czas` o metodę `__iter__()`:

```
def __iter__(self): # nowa metoda w klasie Czas
    ile = self.godziny*60*60 + self.minuty*60 + self.sekundy
    return IterCzas(ile)
```

Teraz:

```
>>> t = Czas(0, 0, 3)
>>> tuple(t)
('Pozostało: 2 s.', 'Pozostało: 1 s.', 'Pozostało: 0 s.')
```

```
>>> tuple(iter(t))
('Pozostało: 2 s.', 'Pozostało: 1 s.', 'Pozostało: 0 s.')
>>> for x in t:
...     print(x)
...
Pozostało: 2 s.
Pozostało: 1 s.
Pozostało: 0 s.
```

16.3. Pola

Przyjrzyjmy się nieco dokładniej polom w klasie. Zauważmy przede wszystkim, że dostęp do nich mamy nie tylko przy użyciu zapisu `obiekt.pole`, ale i następujących funkcji:

```
>>> David.przedstaw_sie() # przypominamy
Nazywam się David Thomas i mam 23 lat(a).
>>> getattr(David, "imie")
'David'
>>> setattr(David, "self.wiek", 41)
>>> hasattr(David, "nazwisko")
True
```

16.3.1. Definiowanie z góry ustalonych pól w klasie

Do tej pory pokazywaliśmy klasy, których pola były tworzone „dynamicznie”, np. w konstruktorze za pośrednictwem operatora przypisania i argumentu `self`. W takim przypadku wszystkie atrybuty są przechowywane w zwykłym słowniku. Zapewnia to dużą elastyczność.

```
>>> David.__dict__
{'kobieta': False, 'imie': 'David', 'nazwisko': 'Thomas', 'wiek': 23,
 'self.wiek': 41}
```

Okazuje się jednak, że w języku Python możemy także tworzyć klasy o z góry ustalonych polach (ang. *slots*). Zmniejsza to zużycie pamięci, przyspiesza działanie programu i może być bezpieczniejsze z punktu widzenia programisty. Na przykład:

```
class Osoba:
    # tutaj definiujemy nazwy pól:
    __slots__ = ["imie", "nazwisko", "wiek"]

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

```
def przedstaw_sie(self):
    print("Nazywam się %s %s i mam %d lat(a)." %
          (self.imie, self.nazwisko, self.wiek))
```

Utwórzmy nową osobę i pokażmy, że możemy na niej działać tak, jak do tej pory:

```
>>> Kandydat = Osoba("David", "Napewno", 30)
>>> Kandydat.nazwisko
'Napewno'
>>> Kandydat.imie = "Thomas"
>>> Kandydat.przedstaw_sie()
Nazywam się Thomas Napewno i mam 30 lat(a).
```

Jednakże:

```
>>> Kandydat.nowe_pole = 7
Traceback (most recent call last):
AttributeError: 'Osoba' object has no attribute 'nowe_pole'
>>> Kandydat.__dict__
Traceback (most recent call last):
AttributeError: 'Osoba' object has no attribute '__dict__'
>>> Kandydat.__slots__
['imie', 'nazwisko', 'wiek']
```

16.3.2. Pola prywatne, chronione i publiczne

Tak zwana *hermetyzacja* to mechanizm, który umożliwia ukrycie szczegółów implementacji danej klasy przed użytkownikiem końcowym. W szczególności dotyczy to pól i niektórych metod. Najczęściej wyróżnia się trzy następujące typy widoczności atrybutów:

- publiczny (ang. *public*) – każdy ma do nich dostęp;
- chroniony (ang. *protected*) – dostęp możliwy tylko z poziomu tej samej klasy i klas, które po niej dziedziczą;
- prywatny (ang. *private*) – dostęp możliwy tylko z poziomu metod w tej samej klasie.

Już wcześniej wspomnieliśmy, że język Python tak naprawdę „nie ma nie do ukrycia”. Możemy jednak imitować różne poziomy dostępności, dodając odpowiedni przedrostek do nazw atrybutów: jeśli poprzedzimy nazwę pola jednym znakiem podkreślenia, oznajmiamy tym samym użytkownikom, że dany element należy uznać za szczegół implementacyjny, który nie jest dostępny publicznie i może ulec zmianie w każdej chwili bez wcześniejszego ostrzeżenia. Możemy powiedzieć, że jest to jakiś odpowiednik widoczności typu chronionego.

Gdy z kolei użyjemy przedrostka nazwy złożonego z dwóch znaków podkreślenia, atrybut taki nadal będzie dostępny z zewnątrz, jednak jego nazwa zostanie automatycznie zmieniona na `_nazwaklasy__element`. Jest to swego rodzaju odpowiednik elemen-

trywiatnego – pozwala to np. uniknąć konfliktu nazw między klasą bazową a jej pochodną.

```
class Test:
    __slots__ = ["publiczne", "_chronione", "__prywatne"]

    def __init__(self):
        self.publiczne, self._chronione, self.__prywatne = 1, 2, 3

    def __str__(self):
        return str((self.publiczne, self._chronione, self.__prywatne))
```

Przetestujmy najpierw dostęp do pola „publicznego”.

```
>>> x = Test()
>>> print(x)
(1, 2, 3)
>>> x.publiczne
1
```

Odwołanie się do atrybutu „chronionego” także nie sprawia problemów:

```
>>> x._chronione
2
```

Warto przypomnieć, że takie nazwy nie są automatycznie podpowiadane np. przez serwer Jupyter. Ciekawy jest jednak przypadek pól prywatnych:

```
>>> x.__prywatne
Traceback (most recent call last):
AttributeError: 'Test' object has no attribute '__prywatne'
```

Ta nazwa została bowiem przetłumaczona na:

```
>>> x._Test__prywatne
3
```

Podobnie będzie zresztą w przypadku dziedziczenia po takiej klasie.

```
class Test2(Test):
    def zmien_chronione(self):
        self._chronione += 1
    def zmien_prywatne(self):
        self.__prywatne += 1
```

Automatyczne „zakrywanie” nazw pól poprzedzonych dwoma znakami `__` pozwala zapobiec konfliktowi nazw między klasą bazową a jej pochodnymi.

```
>>> y = Test2()
>>> y.zmien_chronione()
>>> y.zmien_prywatne() # zachowanie poprawne
```

```

Traceback (most recent call last):
AttributeError: 'Test2' object has no attribute '_Test2__prywate'
>>> print(y)
(1, 3, 3)

```

ZADANIE 16.1. Utwórz klasę **Klasyfikator**, której konstruktor przyjmuje jako argument próbę uczącą (punkty i odpowiadające im etykiety). Niech udostępnia ona dwie „wirtualne” metody **ucz_sie()** (konstrukcja modelu klasyfikacji na podstawie próby uczącej) i **klasyfikuj()** (klasyfikacja wszystkich punktów z próby testowej podanej jako argument), obydwie zgłaszając wyjątek **NotImplementedError**.

Następnie utwórz kilka „konkretnych” klas dziedziczących po powyższej, np. **KlasyfikatorKnn**, **KlasyfikatorDrzewa**, **KlasyfikatorLas** (por. podrozdz. 14.3), implementujących **ucz_sie()** i **klasyfikuj()** w sobie właściwy sposób (możesz oprzeć się na wywoływaniu odpowiednich funkcji i metod z pakietu scikit-learn).

Postaraj się rozszerzyć implementacje tych klas o różne dodatkowe operacje: możliwość testowania jakości uzyskanego modelu (m.in. przy użyciu kroswalidacji), rysowanie itd.

ZADANIE 16.2. Zaimplementuj klasę **DisjointSets**, reprezentującą podziały (jak w analizie skupień) danego zbioru liczb całkowitych $\{0, 1, \dots, k - 1\}$ dla pewnego $k > 1$. Powinna ona udostępniać co najmniej dwie metody: **union()** (łącząca w jeden dwa podzbiory zawierające jako elementy podane liczby) oraz **find()** (zwraca unikatowy identyfikator podzbioru zawierającego jako element podaną liczbę).

Następnie zaimplementuj algorytm hierarchicznej analizy skupień z odmiennością najbliższego sąsiada (zob. p. 14.4.2). Funkcja **single_linkage()** przyjmuje jako argument macierz $\mathbb{R}^{n \times d}$, która reprezentuje n punktów w \mathbb{R}^d . Następnie oblicza ona macierz odległości między wszystkimi parami punktów (metryka euklidesowa) – możemy traktować ją jako pełny graf ważony. Poszukiwane rozwiązanie wyznaczamy za pomocą algorytmu Kruskala, który znajduje minimalne drzewo rozpinające. Postaraj się, by zwracany wynik był w postaci, którą akceptuje funkcja **scipy.cluster.hierarchy.dendrogram()**.

ZADANIE 16.3. Zaimplementuj klasę, która dziedziczy po **pandas.DataFrame**. Przeciąż jej operator indeksowania w taki sposób, by wybór wierszy był dokonywany na podstawie indeksów wierszy, a nie ich etykiet. Dodaj do niej także według uznania inne, często wykonywane operacje.

Życzymy powodzenia w codziennej pracy ze środowiskiem Python!

BIBLIOGRAFIA

- [1] Beazley D., Jones B.K., *Python. Receptury*. Helion, Gliwice 2014.
- [2] Beliakov G., Bustince H., Calvo T., *A practical guide to averaging functions*. Springer, 2010.
- [3] Biecek P., *Odkrywać! Ujawniać! Objaśniać! Zbiór esejów o sztuce prezentowania danych*. PWN, Warszawa 2016.
- [4] Bolwes M., *Machine learning in Python: Essential techniques for predictive analysis*. Wiley, 2014.
- [5] Bressert E., *SciPy and NumPy*. O'Reilly, 2012.
- [6] Cichosz P., *Systemy uczące się*. WNT, Warszawa 2007.
- [7] Cortez P., Cerdeira A., Almeida F., Matos T., Reis J., Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, **47**: 547–553, 2009.
- [8] Dawson M., *Python dla każdego. Podstawy programowania*. Helion, Gliwice 2014.
- [9] Downey A.B., *Think stats*. O'Reilly, 2014.
- [10] Foley J.D. i in., *Wprowadzenie do grafiki komputerowej*. WNT, Warszawa 2001.
- [11] Fortuna Z., Macukow B., Wąsowski J., *Metody numeryczne*. WNT, Warszawa 2015.
- [12] Friedl J., *Wyrażenia regularne*. Helion, Gliwice 2001.
- [13] Gentle J.E., *Random number generation and Monte Carlo methods*. Springer, 2003.
- [14] Gentle J.E., *Computational statistics*. Springer, 2009.
- [15] Gagolewski M., *Data fusion: Theory, methods, and applications*. Instytut Podstaw Informatyki, Warszawa 2015.
- [16] Gagolewski M., *Programowanie w języku R. Analiza danych. Obliczenia. Symulacje*. Wydawnictwo Naukowe PWN, Warszawa 2016.
- [17] Goldberg D., What every computer scientist should know about floating-point arithmetic. *ACM SIGART Computing Surveys*, **21**(1):5–48, 1991.
- [18] Gorelick M., Ozsváld I., *Python. Programuj szybko i wydajnie*. Helion, Gliwice 2015.
- [19] Grabisch M., Marichal J.-L., Mesiar R., Pap E., *Aggregation functions*. Cambridge University Press, 2009.
- [20] Grus J., *Data science from scratch: First principles with Python*. O'Reilly, 2015.
- [21] Hastie T., Tibshirani R., Friedman J., *The elements of statistical learning: Data mining, inference, and prediction*. Springer, 2009.
- [22] Higham N., *Accuracy and stability of numerical algorithms*. SIAM, 2002.
- [23] Idris I., *NumPy beginner's guide*. Packt, 2013.
- [24] Jakubowski J., Sztencel R., *Wstęp do teorii prawdopodobieństwa*. Script, Warszawa 2010.
- [25] Jaworski P., Durante F., Härdle W.K., Rychlik T., *Copula theory and its applications*. Springer, 2010.
- [26] Kazil J., Jarmul K., *Data wrangling with Python: Tips and tools to make your life easier*. O'Reilly, 2016.
- [27] Knowlton J.O., *Python. Projekty do wykorzystania*. Helion, Gliwice 2010.
- [28] Knuth D.E., *Sztuka Programowania. Tom III: Sortowanie i wyszukiwanie*. WNT, Warszawa 2005.
- [29] Knuth D.E., *TeX. Podręcznik użytkownika*. WNT, Warszawa 2005.