

## Zarządzane atrybuty

Niniejszy rozdział rozszerza informacje na temat zaprezentowanych wcześniej technik *przechwytywania atrybutów*, wprowadza kolejne i wykorzystuje je w kilku większych przykładach. Tak jak wszystkie pozostałe rozdziały z tej części książki, został on zaklasyfikowany do tematów bardziej zaawansowanych i opcjonalnych, ponieważ większość programistów aplikacji nie musi się przejmować omawianymi tutaj zagadnieniami — mogą oni pobierać i ustawiać atrybuty obiektów bez martwienia się o samą implementację atrybutów.

Zarządzanie dostępem do atrybutów może jednak być szczególnie istotne dla twórców narzędzi — jako część elastycznego API. Co więcej, znajomość opisanego tutaj modelu deskryptorów pozwoli lepiej zrozumieć inne narzędzia, takie jak sloty i właściwości, a nawet może okazać się niezbędna, jeżeli narzędzia te zostaną użyte w wykorzystywanym kodzie.

### Po co zarządza się atrybutami?

Atrybuty obiektów są kluczowym elementem większości programów napisanych w Pythonie — to w nich często przechowujemy informacje o jednostkach przetwarzanych przez nasz skrypt. Normalnie atrybuty są po prostu zmiennymi obiektów — atrybut `name` osoby może na przykład być prostym łańcuchem znaków, pobieranym i ustawianym za pomocą podstawowej składni atrybutów:

```
person.name           # Pobranie wartości atrybutu
person.name = wartość # Modyfikacja wartości atrybutu
```

W większości przypadków atrybut znajduje się w samym obiekcie lub jest dziedziczony po klasie, z której obiekt ten pochodzi. Ten podstawowy model będzie wystarczający na potrzeby większości programów pisanych w trakcie naszej kariery programisty Pythona.

Czasami jednak wymagana jest większa elastyczność. Przypuśćmy, że napisaliśmy program korzystający z atrybutu `name` w sposób bezpośredni, jednak później wymagania się zmieniają — na przykład decydujemy o tym, że dane te powinny być w jakiś sposób sprawdzane bądź modyfikowane przy pobraniu. Napisanie kodu metod zarządzającego dostępem do wartości atrybutów jest stosunkowo proste (`valid` i `transform` są tutaj wartościami abstrakcyjnymi).

```

class Person:
    def getName(self):
        if not valid():
            raise TypeError('nie można pobrać danych')
        else:
            return self.name.transform()
    def setName(self, value):
        if not valid(value):
            raise TypeError('nie można zmienić danych')
        else:
            self.name = transform(value)

person = Person()
person.getName()
person.setName('value')

```

Taka modyfikacja wymaga jednak wprowadzenia zmian w całym programie — we wszystkich miejscach, w których wykorzystywane są dane osobowe ze zmiennej `name`, co może nie być trywialnym zadaniem. Co więcej, takie rozwiązanie wymaga, by program był świadom sposobu eksportowania wartości — w postaci prostych zmiennych lub wywoływanych metod. Jeśli zaczniemy od interfejsu danych opartego na metodach, klient jest odporny na zmiany. Jeśli tak jednak nie będzie, może się to stać problematyczne.

Takie problemy pojawiają się częściej, niż można by tego oczekiwać. Wartość komórki w programie przypominającym arkusz kalkulacyjny może na przykład rozpocząć swój żywot jako prosta, niezależna wartość, ale później może ona przemienić się w dowolne obliczenia. Ponieważ interfejs obiektu powinien być na tyle elastyczny, by obsługiwać tego typu przyszłe zmiany bez zakłócania działania istniejącego kodu, późniejsze przełączenie się na metody jest dalekie od ideału.

## Wstawianie kodu wykonywanego w momencie dostępu do atrybutów

Lepszym rozwiązaniem byłoby pozwolenie na automatyczne wykonywanie kodu w momencie dostępu do atrybutu, jeśli jest to potrzebne. Jest to jedna z głównych cech zarządzanego atrybutu — umożliwia on definiowanie kodu, który jest wykonywany po uzyskaniu dostępu. Uogólniając, taki atrybut działa w trybie daleko wykraczającym poza proste przechowywanie danych.

W różnych miejscach książki spotkaliśmy narzędzia Pythona pozwalające skryptom automatycznie obliczać wartości atrybutów przy ich pobieraniu i sprawdzające poprawność lub modyfikujące wartości atrybutów przy przechowywaniu. W niniejszym rozdziale rozszerzymy informacje na temat wprowadzonych już narzędzi, poznamy inne dostępne, a także przyjrzymy się większym przykładom przypadków użycia z tej dziedziny. W szczególności rozdział ten prezentuje *cztery* techniki:

1. Wbudowaną funkcję property służącą do przekierowywania dostępu do uszczegółowionych atrybutów.
2. Metody deskryptora `__get__` i `__set__` służące do przekierowywania dostępu do uszczegółowionych atrybutów oraz stanowią bazę dla innych narzędzi, takich jak właściwości i sloty.

3. Metody `__getattr__` oraz `__setattr__` służące do przekierowywania niezdefiniowanych pobrań atrybutów, a także wszystkich przypisań atrybutów.
4. Metodę `__getattribute__` służącą do przekierowywania wszystkich pobrań.

Narzędzia wymienione w punktach zostały krótko opisane w rozdziałach 30. i 32. Jak zobaczymy, wszystkie cztery techniki do pewnego stopnia mają wspólne cele i zazwyczaj można rozwiązać określony problem w kodzie za pomocą dowolnej z nich.

Istnieją jednak między nimi pewne istotne różnice. Przykładowo dwie ostatnie z wymienionych technik mają zastosowanie do *uszczegółowionych* atrybutów, natomiast dwie pierwsze są na tyle ogólne, by móc je wykorzystywać w klasach opartych na delegacji, które muszą przekierowywać dowolne *atrybuty* do opakowanych obiektów. Jak będziemy mogli się przekonać, wszystkie cztery rozwiązania różnią się także stopniem skomplikowania oraz estetyką, co trzeba zobaczyć w praktyce, by móc dokonać ich samodzielnej oceny.

Poza omówieniem szczegółów leżących u podstaw czterech wymienionych wyżej technik przechwytywania atrybutów w niniejszym rozdziale będziemy mieli okazję zapoznać się z programami większymi od widzianych wcześniej w książce. Studium przypadku `CardHolder` z końca rozdziału powinno służyć jako przykład działania większej klasy do samodzielnego studiowania. Części technik wykorzystanych tutaj użyjemy również w kolejnym rozdziale, w kodzie dekoratorów, dlatego przed przejściem dalej należy upewnić się, że omawiane tutaj zagadnienia zrozumiało się przynajmniej w ogólnym zarysie.

## Właściwości

Protokół właściwości pozwala przekierowywać operacje pobierania, ustawiania i usuwania określonego atrybutu do udostępnianych przez nas funkcji lub metod, pozwalając nam wstawiać kod, który zostanie wykonany automatycznie w momencie dostępu do atrybutów, a także przechwytywać usuwanie atrybutów i udostępniać ich dokumentację, jeśli jest to pożądane.

Jak już powiedzieliśmy w rozdziale 32., właściwości tworzone są za pomocą funkcji wbudowanej `property` i przypisywane są do atrybutów klas, podobnie jak funkcje metod. Tym samym są także dziedziczone przez klasy podrzędne oraz instancje, podobnie jak wszystkie inne atrybuty klas. Do ich przechwytywania dostęp funkcji przekazywany jest argument instancji `self`, który umożliwia dostęp do informacji o stanie i do atrybutów klas dostępnych w podmiotowej instancji.

Właściwość zarządza pojedynczym, określonym atrybutem. Choć nie jest w stanie przechwytywać wszystkich dostępu do atrybutów w sposób ogólny, pozwala na kontrolowanie dostępu związanego z pobieraniem i przypisaniem, a także umożliwia zmianę atrybutu z prostych danych na swobodnie obliczany bez zakłócania działania istniejącego kodu. Jak zobaczymy, właściwości są mocno powiązane z deskryptorami — są właściwie ich ograniczoną postacią.

## Podstawy

Właściwość tworzona jest za pomocą przypisania wyniku wbudowanej funkcji do atrybutu klasy:

```
attribute = property(fget, fset, fdel, doc)
```

Żaden z argumentów funkcji wbudowanej nie jest wymagany i w razie ich nieprzekazania wszystkie otrzymują wartość domyślną `None`. Takie operacje nie będą wtedy obsługiwane, a próba ich wykonania spowoduje automatyczne zwrócenie wyjątku `AttributeError`.

Przy użyciu `do fget` przekazujemy funkcję służącą do przechwytywania pobrań atrybutów, `do fset` — funkcję do przypisań, `do fdel` — funkcję do usuwania atrybutów. Z technicznego punktu widzenia we wszystkich trzech argumentach można umieszczać dowolne funkcje i metody, przy czym pierwszym argumentem takiej funkcji musi być referencja do instancji klasy posiadającej dany atrybut. Funkcja umieszczona w argumencie `fget` musi zwracać wartość atrybutu, natomiast funkcje umieszczone w argumentach `fset` i `fdel` nie mogą zwracać żadnego wyniku (albo zwracać wartość `None`). Wszystkie trzy rodzaje funkcji mogą zgłaszać wyjątki oznaczające odmowę dostępu do atrybutu.

Argument `doc` otrzymuje łańcuch znaków dokumentacji dla atrybutu, jeśli jest to pożądane (w przeciwnym razie właściwość kopiuje łańcuch znaków dokumentacji `fget`, o ile jest on podany, który ma wartość domyślną `None`).

Wbudowana funkcja `property` zwraca obiekt właściwości przypisywany do nazwy atrybutu, którym chcemy zarządzać w zakresie klasy, gdzie zostanie on odziedziczony przez wszystkie instancje. Jak się wkrótce dowiesz, to przypisanie można zautomatyzować za pomocą składni dekoratora `@`, chociaż jego rozproszona użyteczność może wydawać się nieporęczna w przypadku metod ustawiania i usuwania. Niezależnie od sposobu przypisania późniejsze odwołania do atrybutu automatycznie wywołują metody obsługujące właściwość.

## Pierwszy przykład

W celu zademonstrowania, jak przekłada się to na działający kod, klasa z przykładu 38.1 wykorzystuje właściwość do śledzenia dostępu do atrybutu o nazwie `name`. Same przechowywane dane noszą nazwę `_name` w celu uniknięcia konfliktu nazw z właściwością.

Przykład 38.1. *prop-person.py*

```
class Person:
    def __init__(self, name):
        self._name = name

    def getName(self):
        print('pobieranie...')
        return self._name

    def setName(self, value):
        print('modyfikacja...')
        self._name = value
```

```

def delName(self):
    print('usunięcie...')
    del self._name
name = property(getName, setName, delName, 'Dokumentacja właściwości name')

bob = Person('Robert Zielony')           # bob ma zarządzany atrybut
print(bob.name)                          # Wykonuje getName
bob.name = 'Robert A. Zielony'           # Wykonuje setName
print(bob.name)
del bob.name                             # Wykonuje delName

print('-'*20)
anna = Person('Anna Czerwona')           # anna także dziedziczy właściwość
print(anna.name)
print(Person.name.__doc__)                # Lub help(Person.name)

```

Ta akurat właściwość niewiele robi — po prostu przechwytuje i śledzi atrybut — jednak służy do zademonstrowania protokołu. Kiedy powyższy kod zostanie wykonany, dwie instancje dziedziczą właściwość — w ten sam sposób jak odziedziczyłyby dowolny inny atrybut dołączony do swojej klasy. Jednak dostęp do ich atrybutu name jest przechwytywany i zarządzany przez nasz kod:

```

$ python3 prop-person.py
pobieranie...
Robert Zielony
modyfikacja...
pobieranie...
Robert Zielony
usunięcie...
-----
pobieranie...
Anna Czerwona
Dokumentacja właściwości name

```

Tak jak wszystkie atrybuty klas, właściwości są *dziedziczone* zarówno przez instancje, jak i klasy podrzędne znajdujące się niżej w hierarchii. Jeśli zmodyfikujemy nasz przykład w następujący sposób:

```

class Super:
    ...kod oryginalnej klasy Person...
    name = property(getName, setName, delName, 'Dokumentacja właściwości name')

class Person(Super):
    pass                                     # Właściwości są dziedziczone (atrybuty klasy)

bob = Person('Robert Zielony')
...reszta bez zmian...

```

Wynik będzie taki sam — klasa podrzędna Person dziedziczy właściwość name po klasie Super, natomiast instancja bob otrzymuje ją po Person. Jeśli chodzi o dziedziczenie, właściwości działają tak samo jak normalne metody; ponieważ mają dostęp do argumentów instancji self, mogą uzyskiwać dostęp do informacji o stanie w instancji, podobnie jak metody bez względu na głębokość klasy podrzędnej, co zobaczymy dalej.

## Obliczanie atrybutów

Przykład powyżej śledzi po prostu dostęp do atrybutów. Zazwyczaj jednak właściwości robią o wiele więcej — na przykład obliczają wartość atrybutu w sposób dynamiczny w momencie pobierania. Przykład 38.2 ilustruje takie zastosowanie.

Przykład 38.2. *prop-computed.py*

```
class PropSquare:
    def __init__(self, start):
        self.value = start

    def getX(self):
        return self.value ** 2

    def setX(self, value):
        self.value = value

X = property(getX, setX)

P = PropSquare(3)
Q = PropSquare(32)

print(P.x)
P.x = 4
print(P.x)
print(Q.x)
```

# Przy pobraniu atrybutów

# Przy przypisaniu atrybutów

# Brak usuwania i dokumentacji

# 2 instancje klasy z właściwością

# Każda ma inne informacje o stanie

# 3 \*\* 2

# 4 \*\* 2

# 32 \*\* 2

Powyższa klasa definiuje atrybut `X`, do którego dostęp odbywa się tak, jakby był on danymi statycznymi, jednak tak naprawdę wykonuje kod obliczający jego wartość przy pobraniu. Rezultat przypomina niejawne wywołanie metody. Kiedy kod jest wykonywany, wartość przechowywana jest w instancji w postaci informacji o stanie, jednak za każdym razem, gdy pobieramy ją za pomocą zarządzanego atrybutu, wartość ta jest automatycznie podnoszona do kwadratu:

```
$ python3 prop-computed.py
9
16
1024
```

Warto zwrócić uwagę na to, że utworzyliśmy dwie różne instancje. Ponieważ metody właściwości automatycznie otrzymują argument `self`, mają dostęp do informacji o stanie przechowywanych w instancjach. W naszym przypadku oznacza to, że operacja pobrania oblicza kwadrat danych podmiotowej instancji.

## Zapisywanie właściwości w kodzie za pomocą dekoratorów

Choć dodatkowe szczegóły zachowamy na kolejny rozdział, podstawy dekoratorów funkcji wprowadziliśmy wcześniej, w rozdziale 32. Przypomnijmy, że składnia dekoratorów:

```
@dekorator
def funkcja(argumenty): ...
```

automatycznie przekładana jest przez Pythona na poniższy odpowiednik w celu ponownego dowiązania nazwy funkcji do wyniku obiektu wywołalnego *dekorator*:

```
def funkcja(argumenty): ...  
    funkcja = dekorator(funkcja)
```

Dzięki temu odwzorowaniu okazuje się, że wbudowana funkcja *property* może służyć jako dekorator, definiujący funkcję, która zostanie wykonana automatycznie, kiedy atrybut zostaje pobrany:

```
class Person:  
    @property  
    def name(self): ...           # Ponownie dowiązuje name = property(name)
```

Po wykonaniu do udekorowanej metody automatycznie przekazywany jest pierwszy argument funkcji wbudowanej *property*. Jest to tak naprawdę składnia alternatywna dla tworzenia właściwości i ręcznego, ponownego dowiązania nazwy atrybutu, ale może być postrzegany jako bardziej jawny:

```
class Person:  
    def name(self): ...  
    name = property(name)       # Ręczny odpowiednik @property
```

## Dekoratory setter i deleter

Kod powyżej działa dla funkcji pobierających właściwości, ale co z innymi rodzajami dostępu? Obiekty właściwości mogą zawierać metody *getter*, *setter* i *deleter*, które przypisują odpowiednie metody akcesora właściwości i zwracają kopię samej właściwości. Możemy je wykorzystać do określenia komponentów właściwości, dekorując także normalne metody, choć komponent *getter* zazwyczaj wypełniany jest automatycznie przez sam fakt tworzenia właściwości. Przykład 38.3 pokazuje podstawy.

Przykład 38.3. *prop-person-deco.py*

```
class Person:  
    def __init__(self, name):  
        self._name = name  
  
    @property  
    def name(self):           # name = property(name)  
        'Dokumentacja właściwości name'  
        print('pobieranie...')  
        return self._name  
  
    @name.setter  
    def name(self, value):    # name = name.setter(name)  
        print('modyfikacja...')  
        self._name = value  
  
    @name.deleter  
    def name(self):           # name = name.deleter(name)  
        print('usunięcie...')  
        del self._name
```

```

bob = Person('Robert Zielony')      # bob ma zarządzany atrybut
print(bob.name)                     # Wykonuje komponent getter dla name
                                    # (pierwszy dostęp do name)
bob.name = 'Robert A. Zielony'      # Wykonuje komponent setter dla name
                                    # (drugi dostęp do name)

print(bob.name)
del bob.name                         # Wykonuje komponent deleter dla name
                                    # (trzeci dostęp do name)

print('-'*20)
anna = Person('Anna Czerwona')      # anna także dziedziczy właściwość
print(anna.name)
print(Person.name.__doc__)           # Lub help(Person.name)

```

Tak naprawdę powyższy kod jest odpowiednikiem pierwszego przykładu z tego podrozdziału — dekoracja to w tym przypadku po prostu alternatywny sposób zapisania w kodzie właściwości. Po wykonaniu wyniki są takie same:

```

$ python3 prop-person-deco.py
pobieranie...
Robert Zielony
modyfikacja...
pobieranie...
Robert A. Zielony
usunięcie...
-----
pobieranie...
Anna Czerwona
Dokumentacja właściwości name

```

W porównaniu z ręcznym przypisywaniem wyników za pomocą property w tym przypadku użycie dekoratorów do tworzenia właściwości wymaga jedynie trzech dodatkowych wierszy kodu (różnica jest do pominięcia). Jak to często bywa w przypadku narzędzi alternatywnych, wybór pomiędzy dwoma technikami jest w dużej mierze kwestią subiektywną.

## Deskryptory

Omówione krótko w rozdziale 32. *deskryptory* stanowią alternatywny sposób przechwytywania dostępu do atrybutów i są mocno powiązane z właściwościami omówionymi w poprzednim podrozdziale. Tak naprawdę właściwość *jest* rodzajem deskryptora — funkcja wbudowana property jest uproszczonym sposobem tworzenia określonego typu deskryptora, który wykonuje funkcje metod w momencie dostępu do atrybutów. Deskryptory implementują mechanizm wykorzystywany przez różne narzędzia w klasie, w tym właściwości i sloty, oraz pełnią inne wewnętrzne funkcje w Pythonie, które możemy spokojnie tutaj pominąć.

Z punktu widzenia funkcjonalności protokół deskryptora pozwala nam przekierować operacje pobierania i ustawiania określonego atrybutu do metod osobnego obiektu klasy, który udostępnimy. Umożliwiają one wstawienie kodu wykonywanego automatycznie w momencie dostępu do atrybutu i pozwalają przechwytywać operacje usuwania atrybutów, a także udostępniać dokumentację, jeśli jest to pożądane.



Deskryptory tworzone są jako niezależne klasy i są przypisywane do atrybutów *klas* tak samo jak funkcje metod. Tak jak wszystkie inne atrybuty klas, są one dziedziczone przez klasy podrzędne oraz instancje. Do ich metod przechwytyjących operacje dostępu przekazywane są zarówno sam deskryptor (w postaci argumentu `self`), jak i instancje klasy klienta. Z tego powodu zachowują i wykorzystują własne informacje o stanie, a także informacje o stanie podmiotowej instancji. Przykładowo deskryptor może wywoływać metody dostępne w klasie klienta, a także definiowane przez niego metody specyficzne dla deskryptora.

Podobnie jak właściwość, deskryptor zarządza pojedynczym, określonym atrybutem. Choć nie jest w stanie przechwytywać wszystkich dostępuów do atrybutów w sposób uniwersalny, udostępnia kontrolę nad dostępem związanym zarówno z pobieraniem, jak i z przypisywaniem, a także pozwala dowolnie modyfikować atrybut z prostych danych na obliczenia bez zakłócania działania istniejącego kodu. Właściwości tak naprawdę są po prostu wygodnym sposobem tworzenia określonego typu deskryptora i, jak zobaczymy, można je tworzyć w kodzie bezpośrednio w postaci deskryptorów.

Podczas gdy właściwości mają stosunkowo wąski zakres, deskryptory są rozwiązaniem bardziej uniwersalnym. Przykładowo, ponieważ tworzone są w postaci normalnych klas, deskryptory mają swój własny stan, mogą brać udział w hierarchiach dziedziczenia deskryptorów, wykorzystują kompozycję do agregacji obiektów i stanowią naturalną strukturę dla tworzenia w kodzie metod wewnętrznych oraz łańcuchów znaków dokumentacji atrybutów.

## Podstawy

Jak wspomniano wcześniej, deskryptory zapisywane są w kodzie jako odrębne klasy i udostępniają metody akcesorów o specjalnych nazwach, służące do operacji dostępu do atrybutów, które mają przechwytywać. Metody pobierania (`__get__`), ustawiania (`__set__`) oraz usuwania (`__delete__`) w klasie deskryptora są wykonywane automatycznie w momencie wystąpienia odpowiedniego typu dostępu do atrybutu przypisanego do instancji klasy deskryptora:

```
class Descriptor:
    "miejsce na łańcuch znaków dokumentacji"
    def __get__(self, instancja, właściciel): ...      # Zwraca wartość atrybutu
    def __set__(self, instancja, właściciel): ...      # Nic nie zwraca (None)
    def __delete__(self, instancja): ...               # Nic nie zwraca (None)
```

Klasy zawierające dowolną z powyższych metod uznawane są za deskryptory, a ich metody są specjalne, jeśli jedna z ich instancji zostanie przypisana do atrybutu innej klasy — przy dostępie do atrybutu są one wywoływane automatycznie.

Jeśli któraś z metod jest nieobecna, oznacza to zazwyczaj, że odpowiadający jej typ dostępu nie jest obsługiwany. W przeciwieństwie do właściwości pominięcie metody `__set__` pozwala na ponowne zdefiniowanie nazwy w instancji, tym samym *ukrywając* deskryptor. By atrybut był tylko do odczytu, należy zdefiniować metodę `__set__` w taki sposób, aby przechwytywała przypisanie i zgłaszała wyjątek.

Deskryptor zawierający metodę `__set__` wywołuje pewne specyficzne implikacje podczas dziedziczenia klas. Ten temat został odłożony do rozdziału 40., w którym szczegółowo zostały

opisane metaklasy i proces dziedziczenia. W skrócie: deskryptor zawierający metodę `__set__` jest nazywany **deskryptorem danych** i w regułach zwykłego dziedziczenia opatrzona nim nazwa jest traktowana priorytetowo. Na przykład odziedziczony deskryptor nazwy `__class__` zastępuje nazwę instancji. Ponadto deskryptory kodowane we własnych klasach mają pierwszeństwo przed innymi deskryptorami.

## Argumenty metod deskryptorów

Zanim zajmiemy się utworzeniem jakiegoś realistycznego przykładu, przyjrzyjmy się podstawom. Do wszystkich trzech opisanych powyżej metod deskryptorów przekazywana jest zarówno instancja klasy deskryptora (`self`), jak i instancja klasy klienta (*instancja*), do której dołączana jest instancja deskryptora.

Metoda dostępu `__get__` dodatkowo otrzymuje argument *właściciel* określający klasę, do której dołączana jest instancja deskryptora. Jej argument *instancja* jest albo instancją, przez którą odbył się dostęp do atrybutu (dla *instancja.atrybut*), lub `None`, jeśli dostęp do atrybutu odbywa się bezpośrednio za pomocą klasy właściciela (dla *klasa.atrybut*). Pierwsza forma zazwyczaj oblicza wartość dla dostępu do instancji, natomiast druga najczęściej zwraca `self`, jeśli obsługiwany jest dostęp do obiektu deskryptora.

Przykładowo w poniższym kodzie uruchomionym w sesji REPL po pobraniu `X.attr` automatycznie jest wykonywana metoda `__get__` klasy `Descriptor`, do której przypisany jest atrybut klasy `Subject.attr`:

```
>>> class Descriptor:
    def __get__(self, instance, owner):
        print(self, instance, owner, sep='\n')

>>> class Subject:
    attr = Descriptor()                                # Instancja klasy Descriptor jest atrybutem klasy

>>> X = Subject()
>>> X.attr
<__main__.Descriptor object at 0x104bc9b20>
<__main__.Subject object at 0x104b8a570>
<class '__main__.Subject'>

>>> Subject.attr
<__main__.Descriptor object at 0x104bc9b20>
None
<class '__main__.Subject'>
```

Warto zwrócić uwagę na argumenty przekazane automatycznie do metody `__get__` przy pierwszym pobraniu atrybutu. Gdy pobierane jest `X.attr`, działa to tak, jakby nastąpił poniższy przykład (choć `Subject.attr` nie wywołuje tutaj ponownie metody `__get__`):

```
X.attr -> Descriptor.__get__(Subject.attr, X, Subject)
```

Deskryptor wie o tym, że odbywa się do niego dostęp bezpośredni, jeśli argument instancji równy jest `None`.

## Deskryptory tylko do odczytu

Jak wspomniano wcześniej, w przeciwieństwie do właściwości, w przypadku deskryptorów pominięcie metody `__set__` nie wystarczy, by atrybut stał się tylko do odczytu, ponieważ zmienną deskryptora można przypisać do instancji. W poniższym kodzie przypisanie atrybutu do `X.a` powoduje przechowanie `a` w instancji obiektu `X`, tym samym ukrywając deskryptor przechowywany w klasie `C`.

```
>>> class D:
...     def __get__(*args): print('pobranie')
...
>>> class C:
...     a = D()                                # Atrybut a jest instancją deskryptora
...
>>> X = C()
>>> X.a                                         # Wykonuje metodę __get__ odziedziczonego deskryptora
pobranie
>>> C.a
pobranie
>>> X.a = 99                                  # Przechowane w X, ukrywa C.a
>>> X.a
99
>>> list(X.__dict__.keys())
['a']
>>> Y = C()
>>> Y.a                                         # Y nadal dziedziczy deskryptor
pobranie
>>> C.a
pobranie
```



*Trio metod usuwających.* Należy także uważać, by nie pomylić metody deskryptora `__delete__` z ogólną metodą `__del__`. Ta pierwsza wywoływana jest przy próbach usunięcia nazwy zarządzanego atrybutu w instancji klasy właściciela. Ta druga jest ogólną metodą destruktoru instancji, wykonywaną gdy instancja klasy dowolnego typu ma być wyczyszczona z pamięci. Metoda `__delete__` jest bliżej związana z ogólną metodą usuwania atrybutu `__delattr__`, z którą spotkamy się w dalszej części rozdziału. Więcej informacji na temat metod przeciążania operatorów można znaleźć w rozdziale 30.

W ten sposób działa przypisywanie atrybutów instancji w Pythonie, co pozwala klasom na selektywne nadpisywanie wartości domyślnych z poziomu klasy w ich instancjach. By uczynić atrybut oparty na deskrypcie atrybutem tylko do odczytu, należy przechwycić przypisanie w klasie deskryptora i zgłosić wyjątek, tak by zapobiec przypisaniu atrybutu. Przy przypisywaniu atrybutu będącego deskryptorem Python obchodzi normalne zachowanie na poziomie instancji i przekierowuje operację do obiektu deskryptora.

```
>>> class D:
...     def __get__(*args): print('pobranie')
...     def __set__(*args): raise AttributeError('nie można ustawić')
...
>>> class C:
...     a = D()
```

```
>>> X = C()
>>> X.a                                     # Przekierowane do C.a.__get__
pobranie
>>> X.a = 99                               # Przekierowane do C.a.__set__
AttributeError: nie można ustawić
```

## Pierwszy przykład

By zobaczyć, jak to wszystko łączy się ze sobą w bardziej realistycznym kodzie, zacznijmy od tego samego pierwszego przykładu, jaki napisaliśmy dla właściwości. Przykład 38.4 definiuje deskryptor przechwytyjący dostęp do atrybutu o nazwie `name` w swoich klientach. Jego metody wykorzystują argument instancji w celu uzyskania dostępu do informacji o stanie z podmiotowej instancji, w której przechowywany jest łańcuch znaków imienia i nazwiska.

Przykład 38.4. *desc-person.py*

```
class Name:
    'Dokumentacja deskryptora name'

    def __get__(self, instance, owner):
        print('pobieranie...')
        return instance._name

    def __set__(self, instance, value):
        print('modyfikacja...')
        instance._name = value

    def __delete__(self, instance):
        print('usunięcie...')
        del instance._name

class Person:
    def __init__(self, name):
        self._name = name
        name = Name()                                     # Przypisanie deskryptora do atrybutu

bob = Person('Robert Zielony')                          # bob ma zarządzany atrybut
print(bob.name)                                         # Wykonuje Name.__get__
bob.name = 'Robert A. Zielony'                         # Wykonuje Name.__set__
print(bob.name)
del bob.name                                             # Wykonuje Name.__delete__

print('-'*20)
anna = Person('Anna Czerwona')                         # anna także dziedziczy deskryptor
print(anna.name)
print(Name.__doc__)                                     # Lub help(Name)
```

Warto zwrócić uwagę na to, jak w powyższym kodzie przypisujemy instancję klasy deskryptora do *atrybutu klasy* w klasie klienta. Z tego powodu jest on dziedziczony przez wszystkie instancje klasy, tak samo jak jej metody. Tak naprawdę *musimy* przypisać deskryptor do atrybutu klasy w taki właśnie sposób — nie będzie on działał, jeśli zamiast tego przypiszemy go do atrybutu instancji `self`. Po wykonaniu metody `__get__` deskryptora przekazywane są do niej trzy obiekty w celu zdefiniowania kontekstu:

- self jest instancją klasy Name,
- instance jest instancją klasy Person,
- owner to klasa Person.

Po wykonaniu powyższego kodu metody deskryptora przechwytyują próby uzyskania dostępu do atrybutów, podobnie jak wersja kodu z właściwościami. Tak naprawdę wynik będzie znowu taki sam:

```
$ python3 desc-person.py
pobieranie...
Robert Zielony
modyfikacja...
pobieranie...
Robert A. Zielony
usunięcie...
-----
pobieranie...
Anna Czerwona
Dokumentacja deskryptora name
```

Podobnie jak w przykładzie z właściwościami, instancja klasy deskryptora jest atrybutem klasy i tym samym *dziedziczona* jest przez wszystkie instancje klasy klienta oraz klasy podrzędne. Jeśli w naszym przykładzie zmienimy klasę Person w następujący sposób, wynik skryptu będzie taki sam:

```
...
class Super:
    def __init__(self, name):
        self._name = name
        name = Name()

class Person(Super):
    pass
...
# Deskryptory są dziedziczone
```

Warto również zauważyć, że kiedy klasa deskryptora nie jest przydatna poza klasą klienta, zupełnie rozsądne jest składniowe osadzenie definicji deskryptora wewnątrz klienta. Oto jak wyglądać będzie nasz przykład w przypadku zastosowania *klasy osadzonej*:

```
class Person:
    def __init__(self, name):
        self._name = name

class Name:
    'Dokumentacja deskryptora name'
    def __get__(self, instance, owner):
        ...to samo...
    def __set__(self, instance, value):
        ...to samo...
    def __delete__(self, instance):
        ...to samo...
    name = Name()
# Zastosowanie klasy osadzonej
```

W takim kodzie Name staje się zmienną lokalną w zakresie instrukcji klasy Person i tym samym nie będzie wchodziła w konflikt z żadnymi nazwami spoza klasy. Powyższa wersja działa tak

samo jak oryginalna — przenieśliśmy po prostu definicję klasy deskryptora do zakresu klasy klienta — jednak ostatni wiersz kodu sprawdzającego musi się zmienić, by pobierać łańcuch znaków dokumentacji z nowej lokalizacji (zawarta w pliku *desc-person-nested.py* w materiałach do pobrania dla tej książki):

```
...
print(Person.Name.__doc__)           # Różnica: już nie Name.__doc__ poza klasą
```

## Obliczone atrybuty

Tak jak było w przypadku zastosowania właściwości, nasz pierwszy przykład deskryptora z powyższego podrozdziału nie robił nic specjalnego — wyświetlał po prostu komunikaty śledzenia dla dostępu do atrybutów. W praktyce deskryptory można także wykorzystać do obliczania wartości atrybutów za każdym razem, gdy są one pobierane. Ilustruje to przykład 38.5 — jest on inną wersją tego samego przykładu, jaki utworzyliśmy dla właściwości. Wykorzystuje on deskryptor do automatycznego obliczenia kwadratu wartości atrybutu z każdym pobraniem.

Przykład 38.5. *desc-computed.py*

```
class DescSquare:
    def __init__(self, start):           # Każdy deskryptor ma własny stan
        self.value = start

    def __get__(self, instance, owner):  # Przy pobieraniu atrybutów
        return self.value ** 2

    def __set__(self, instance, value):  # Przy przypisywaniu atrybutów
        self.value = value              # Brak usuwania i dokumentacji

class Client1:
    X = DescSquare(3)                   # Przypisanie instancji deskryptora do atrybutu klasy

class Client2:
    X = DescSquare(32)                  # Inna instancja w innej klasie klienta
                                        # Można także utworzyć kod dwóch instancji tej samej klasy

c1 = Client1()
c2 = Client2()

print(c1.x)                            # 3 ** 2
c1.x = 4
print(c1.x)                             # 4 ** 2
print(c2.x)                             # 32 ** 2
```

Po wykonaniu wynik powyższego przykładu będzie taki sam jak oryginalnej wersji opartej na właściwościach, jednak tym razem próby dostępu do atrybutów przechwytywane są przez obiekt deskryptora klas:

```
$ python3 desc-computed.py
9
16
1024
```

## Wykorzystywanie informacji o stanie w deskryptorach

Jeśli zastanowimy się chwilę nad dwoma utworzonymi dotychczas przykładami deskryptorów, możemy zauważyć, że swoje informacje pobierają one z różnych miejsc. Pierwszy (przykład z atrybutem `name`) wykorzystuje dane przechowywane w *instancji* klienta, natomiast drugi (przykład z kwadratem atrybutu) wykorzystuje dane dołączone do samego obiektu *deskryptora* (`self`). Tak naprawdę deskryptory mogą wykorzystywać stan *zarówno* instancji, jak i deskryptora lub też dowolną ich kombinację:

- **Stan deskryptora** wykorzystywany jest do zarządzania danymi wewnętrznymi z punktu widzenia działania deskryptora lub danymi ze wszystkich instancji. Może być inny w każdej klasie klienckiej.
- **Stan instancji** zapisuje informacje powiązane z klasą klienta i prawdopodobnie przez nią utworzone. Może być inny w każdej klasie klienckiej (tj. w każdym obiekcie aplikacyjnym).

Innymi słowy, stan deskryptora to dane charakterystyczne dla deskryptora, a stan instancji to dane charakterystyczne dla instancji klienckiej. Jak zawsze w programowaniu obiektowym należy starannie wybierać stan. Na przykład nie można wykorzystywać stanu *deskryptora* do przechowywania nazwiska pracownika, ponieważ w każdej instancji klienckiej wymagana jest inna wartość. Jeżeli nazwisko zostanie zapisane w stanie deskryptora, wtedy będzie współdzielone przez wszystkie instancje klasy klienckiej. Analogicznie: stanu *instancji* nie można wykorzystywać do przechowywania wewnętrznych danych deskryptora, ponieważ zostaną utworzone ich różne kopie.

Metody deskryptorów mogą wykorzystywać oba rozwiązania, jednak stan deskryptora często sprawia, że używanie specjalnych konwencji nazewnictwa w celu uniknięcia konfliktów między nazwami danych deskryptora przechowywanych w instancji nie jest konieczne. Poniższy deskryptor z przykładu 38.6. dołącza informacje do własnej instancji, dzięki czemu nie wchodzi one w konflikt z informacjami instancji klasy klienta.

Przykład 38.6. *desc-state-desc.py*

```
class DescState:                                # Wykorzystanie stanu deskryptora
    def __init__(self, value):
        self.value = value

    def __get__(self, instance, owner):          # Przy pobieraniu atrybutów
        print('pobranie DescState')
        return self.value * 10

    def __set__(self, instance, value):          # Przy przypisywaniu atrybutów
        print('ustawienie DescState')
        self.value = value

# Klasa klienta
class CalcAttrs:
    X = DescState(2)                            # Atrybut klasy deskryptora
    Y = 3                                         # Atrybut klasy
    def __init__(self):
        self.Z = 4                             # Atrybut instancji
```

```

obj = CalcAttrs()
print(obj.x, obj.Y, obj.Z)           # X jest obliczane, pozostałe nie są
obj.x = 5                             # Przypisanie X jest przechwytywane
obj.Y = 6                             # Wartość przypisana ponownie atrybutowi Y w klasie
obj.Z = 7                             # Wartość przypisana ponownie atrybutowi Z w instancji
print(obj.x, obj.Y, obj.Z)

obj2 = CalcAttrs()                   # Atrybut X wykorzystuje współdzielone dane, podobnie jak Y!
print(obj2.x, obj2.Y, obj2.Z)

```

Wartość `value` znajduje się w powyższym kodzie jedynie w *deskrytorze*, dzięki czemu nie wystąpi konflikt, jeśli ta sama nazwa zostanie użyta w instancji klienta. Warto zauważyć, że zarządzamy tutaj jedynie atrybutem deskrytora — przechwytywane są próby pobrania oraz ustawienia zmiennej `X`, jednak dostęp do `Y` oraz `Z` nie (zmienna `Y` dołączona jest do klasy klienta, natomiast `Z` — do instancji). Po wykonaniu powyższego kodu zmienna `X` jest po pobraniu obliczana, ale jej wartość jest taka sama we wszystkich instancjach klienckich, ponieważ wykorzystuje ona stan deskrytora:

```

$ python3 desc-state-desc.py
pobranie DescState
20 3 4
ustawienie DescState
pobranie DescState
50 6 7
pobranie DescState
50 6 4

```

Deskrytor może także przechowywać lub wykorzystywać atrybut dołączony do *instancji* klasy klienta zamiast do siebie. W takim wypadku dane mogą być inne w każdej instancji klienckiej w przeciwieństwie do danych przechowywanych w samym deskrytorze. Deskrytor z przykładu 38.7. zakłada, że instancja ma atrybut `_X` dołączony do klasy klienta, i wykorzystuje go do obliczenia reprezentowanej przez niego wartości.

Przykład 38.7. *desc-state-inst.py*

```

class InstState:                       # Wykorzystanie stanu instancji
    def __get__(self, instance, owner):
        print('pobranie InstState')    # Założenie ustawienia przez klasę klienta
        return instance._X * 10

    def __set__(self, instance, value):
        print('ustawienie InstState')
        instance._X = value

# Klasa klienta
class CalcAttrs:
    X = InstState()                    # Atrybut deskrytora klasy
    Y = 3                              # Atrybut deskrytora klasy

    def __init__(self):
        self._X = 2                    # Atrybut instancji
        self._Z = 4                    # Atrybut instancji

obj = CalcAttrs()

```



```

print(obj.x, obj.Y, obj.Z)
obj.X = 5
CalcAttrs.Y = 6
obj.Z = 7
print(obj.X, obj.Y, obj.Z)

obj2 = CalcAttrs()
print(obj2.X, obj2.Y, obj2.Z)

```

*# X jest obliczane, reszta nie*  
*# Przypisanie X jest przechwytywane*  
*# Wartość przypisana ponownie atrybutowi Y w klasie*  
*# Wartość przypisana atrybutowi Z w instancji*

*# Tym razem wartość X jest inna, podobnie jak Z!*

Tym razem `X` przypisywane jest do deskryptora, który zarządza dostępem, tak jak wcześniej. Nowy deskryptor z powyższego przykładu nie ma własnych informacji, jednak wykorzystuje atrybut, który zakłada, że istnieje w instancji. Atrybut ten nosi nazwę `_X` w celu uniknięcia konfliktu z nazwą z samego deskryptora. Po wykonaniu tej wersji kodu wyniki będą podobne, jednak wartość atrybutu deskryptora może się różnić w zależności od instancji klienta z powodu różnych polityk stanu:

```

$ python3 desc-state-inst.py
pobranie InstState
20 3 4
ustawienie InstState
pobranie InstState
50 6 7
pobranie InstState
20 6 4

```

Zarówno stan z deskryptora, jak i stan z instancji pełnią swoje role. Tak naprawdę na tym właśnie polega przewaga deskryptorów nad właściwościami — ponieważ mają one własny stan, mogą z łatwością przechowywać dane wewnętrznie, bez dodawania ich do przestrzeni nazw obiektu instancji klienta. Poniższy przykład stanowi podsumowanie. Wykorzystane są w nim *oba* źródła informacji o stanie. Atrybut `self.data` zawiera informacje właściwe dla deskryptora, a `instance.data` informacje właściwe dla instancji klienckiej:

```

>>> class DescBoth:
    def __init__(self, data):
        self.data = data
    def __get__(self, instance, owner):
        return f'{self.data}, {instance.data}'
    def __set__(self, instance, value):
        instance.data = value

>>> class Client:
    def __init__(self, data):
        self.data = data
    managed = DescBoth('hakować')

>>> I = Client('kod')
>>> I.managed
'hakować, kod' >>> I.managed = 'HAKOWAĆ'
>>> I.managed
'hakować, HAKOWAĆ'

```

*# Wyświetlenie obu źródeł danych*  
*# Zmiana danych instancji*

Skutki wyboru miejsca przechowywania danych zostaną szerzej opisane w przykładzie w dalszej części rozdziału. Zanim przejdziemy dalej, przypomnijmy sobie z opisu slotów w rozdziale 32., że do „wirtualnych” atrybutów, takich jak właściwości i deskryptory, nawet jeżeli

nie ma ich w słowniku przestrzeni nazw instancji, można odwoływać się za pomocą metod `dir` i `getattr`. Odwoływanie się do nich w ten czy inny sposób zależy od programu. W przypadku właściwości i deskryptorów można wykonywać dowolne operacje. Są to mniej oczywiste instancje „danych” niż sloty:

```
>>> I.__dict__
{'dane': 'HAKOWAĆ'}
>>> [x for x in dir(I) if not x.startswith('__')]
['dane', 'zarządzane']

>>> getattr(I, 'dane')
'HAKOWAĆ'
>>> getattr(I, 'zarządzane')
'hakować, HAKOWAĆ'

>>> for attr in (x for x in dir(I) if not x.startswith('__')):
    print(f'{attr} => {getattr(I, attr)}')
dane => HAKOWAĆ
zarządzane => hakować, HAKOWAĆ
```

Bardziej ogólne narzędzia `__getattr__` i `__getattribute__`, opisane w dalszej części rozdziału, nie są przeznaczone do powyższych zastosowań. Ponieważ nie mają atrybutów na poziomie klasy, wyniki zwracane przez metodę `dir` nie zawierają nazwy ich „wirtualnych” atrybutów (zgodnie z tym, co zostało powiedziane w rozdziale 31., metoda `__dir__` może zwrócić wynik `dir`, ale jest to opcjonalne i rzadko stosowane). Nie są za to ograniczone do określonych nazw atrybutów zakodowanych jako właściwości lub deskryptory, które oferują znacznie więcej możliwości, o czym mowa w następnym podrozdziale.

## Powiązania pomiędzy właściwościami a deskryptorami

Jak wspomniano wcześniej, właściwości i deskryptory są ze sobą silnie powiązane — wbudowana funkcja `property` jest po prostu wygodnym sposobem tworzenia deskryptora. Skoro już wiemy, jak działają oba rozwiązania, powinniśmy być w stanie zobaczyć, że możliwe jest symulowanie funkcji wbudowanej `property` za pomocą klasy deskryptora, jak w przykładzie 38.8.

Przykład 38.8. *prop-desc-equiv.py*

```
class Property:
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc
        # Zapisanie metod bez wiązania
        # lub innych obiektów wywoływalnych

    def __get__(self, instance, instancetype=None):
        if instance is None:
            return self
        if self.fget is None:
            raise AttributeError("nie można pobrać atrybutu")
        return self.fget(instance)
        # Przekazanie instancji do self
        # w akcesorach właściwości

    def __set__(self, instance, value):
```

```

        if self.fset is None:
            raise AttributeError("nie można ustawić atrybutu")
        self.fset(instance, value)

    def __delete__(self, instance):
        if self.fdel is None:
            raise AttributeError("nie można usunąć atrybutu")
        self.fdel(instance)

class Person:
    def getName(self): ...
        print('getName...')
    def setName(self, value): ...
        print('setName...')
    name = Property(getName, setName)    # Użyć jak property()

x = Person()
x.name
x.name = 'Robert'
del x.name

```

Powyższa klasa Property przechwytyje dostęp do atrybutów za pomocą protokołu deskryptora i przekierowuje żądania do funkcji lub metod przekazanych i zapisanych w stanie deskryptora, kiedy klasa jest tworzona. Pobranie atrybutu jest na przykład przekierowywane z klasy Person do metody `__get__` klasy Property i z powrotem do metody `getName` klasy Person. W przypadku deskryptorów wszystko „po prostu działa”:

```

$ python3 prop-desc-equiv.py
getName...
setName...
AttributeError: nie można usunąć atrybutu

```

Warto zauważyć, że przykład odpowiednika klasy deskryptora obsługuje jedynie proste zastosowania właściwości. By skorzystać ze *składni dekoratorów* z `@` w celu określenia operacji ustawiania i usuwania, nasza klasa Property musiałaby zostać rozszerzona za pomocą metod setter oraz deleter, zapisujących udekorowaną funkcję akcesora i zwracających obiekt właściwości (powinno tu wystarczyć `self`). Ponieważ funkcja wbudowana property robi to za nas, pominiemy tutaj kod takiego rozszerzenia.

## Deskryptory, sloty i nie tylko

Teraz możemy sobie wyobrazić, jak można wykorzystywać deskryptory do implementowania *slotów*. Można zrezygnować ze słowników atrybutów instancji i stworzyć deskryptory na poziomie klasy, przechwytyjące dostęp do nazw slotów i wiążące je z przestrzenią wykorzystywaną przez instancję. Jednak w odróżnieniu od jawnego wywołania funkcji property większość magii slotów dzieje się w chwili automatycznego lub jawnego tworzenia klasy, gdy pojawia się atrybut `__slots__`.

Rozdział 32. zawiera więcej informacji o slotach (wraz z uzasadnieniem, dlaczego nie należy ich używać, z wyjątkiem patologicznych sytuacji). Deskryptory są również używane do innych narzędzi klasowych, ale tutaj pominiemy dalsze szczegóły. Więcej informacji znajdziesz w dokumentacji Pythona i jego kodzie źródłowym.



*Niedokończone wątki o deskryptorach.* W rozdziale 39. będziemy także wykorzystywać deskryptory do implementowania dekoratorów funkcji mających zastosowanie zarówno do funkcji, jak i metod. Jak zobaczymy, ponieważ deskryptory otrzymują zarówno instancje deskryptora, jak i klasy podmiotowej, w tej roli dobrze się sprawdzają, choć funkcje zagnieżdżone są zazwyczaj prostszym rozwiązaniem. W tym rozdziale wykorzystamy również deskryptory do przechwycenia wywołań wbudowanych metod, a w rozdziale 40., w którym dokładnie opisane jest pierwszeństwo deskryptorów danych we wspomnianym wcześniej modelu *dziedziczenia* klas. Jeżeli użyje się metody `__set__`, wtedy deskryptory zastępują inne nazwy, przez co są wiążące i nie mogą być zastąpione nazwami znajdującymi się w słownikach instancji.

## Metody `__getattr__` i `__getattribute__`

Dotychczas omówiliśmy właściwości oraz deskryptory — narzędzia służące do zarządzania określonymi atrybutami. Metody przeciążania operatorów `__getattr__` oraz `__getattribute__` udostępniają jeszcze inne sposoby przechwytywania pobrań atrybutów dla instancji klas. Tak jak właściwości oraz deskryptory, pozwalają one wstawiać kod, który będzie wykonywany w momencie dostępu do atrybutów. Jak jednak zobaczymy, te dwie metody mogą być wykorzystywane w sposób bardziej uniwersalny. Ponieważ przechwytyują dowolne nazwy, ich rola jest znacznie szersza — umożliwiają m.in. delegowanie wywołań. W niektórych sytuacjach mogą być dodatkowo wywoływane, są jednak zbyt dynamiczne, aby mogły być umieszczone w wyniku metody `dir`.

Przechwytywanie pobierania atrybutów ma dwie odmiany, których kod tworzy się za pomocą dwóch różnych metod:

- Metoda `__getattr__` wykonywana jest dla atrybutów *niezdefiniowanych*, to znaczy atrybutów nieprzechowywanych w instancji lub dziedziczonych po jednej z jej klas.
- Metoda `__getattribute__` wykonywana jest dla *każdego* atrybutu, dlatego wykorzystując ją, trzeba uważać i unikać pętli rekurencyjnych przy przekazywaniu dostępu do atrybutów do klasy nadrzędnej.

Z pierwszą z powyższych metod spotkaliśmy się w rozdziale 30. Dwie powyższe metody są reprezentantami zbioru metod przechwytywania atrybutów, do którego należą także `__setattr__` i `__delattr__`. Ponieważ metody te pełnią podobne role, potraktujemy je tutaj jako jedno zagadnienie.

W przeciwieństwie do właściwości oraz deskryptorów metody te są częścią protokołu *przeciążania operatorów* Pythona — metod klas o specjalnych nazwach, dziedziczonych przez klasy podrzędne i wykonywanych automatycznie, gdy instancje użyte zostają w domniemanej wbudowanej operacji. Tak jak wszystkie metody klasy, każda z nich po wywołaniu otrzymuje pierwszy argument `self`, co daje dostęp do wszelkich wymaganych informacji o stanie instancji lub innych metodach klasy.

Metody `__getattr__` i `__getattribute__` są także bardziej *ogólne* od właściwości i deskryptorów — można je wykorzystywać do przechwytywania dostępu do dowolnych (nawet wszystkich)

pobrań atrybutów instancji, a nie tylko określonych nazw, do których zostały one przypisane. Z tego powodu metody te są dobrze przystosowane do ogólnych wzorców kodu opartego na *delegacji* — można je wykorzystywać do implementowania obiektów opakowujących, zarządzających wszystkimi dostęпами do atrybutów dla obiektów osadzonych. Z kolei w przypadku właściwości bądź deskryptorów musimy zdefiniować po jednym z nich dla każdego atrybutu, który chcemy przechwytywać. Jak się za chwilę przekonamy, w nowego rodzaju klasach rola ta jest nieco ograniczona w przypadku wbudowanych operacji, jednak obejmuje wszystkie nazywane metody opakowanego obiektu.

Wreszcie te dwie metody są o wiele węższe i *bardziej skoncentrowane* w swoim działaniu od wcześniej rozważanych alternatyw — przechwytyują jedynie pobrania atrybutów, a nie przypisania. By przechwytywać również zmianę atrybutu przez jego przypisanie, musimy utworzyć kod `__setattr__` — metody przeciążania atrybutów wykonywanej dla każdego ich pobrania, w przypadku której należy uważać, by uniknąć pętli rekurencyjnych poprzez przekierowanie przypisania atrybutów za pośrednictwem słownika przestrzeni nazw instancji. Choć zdarza się to o wiele rzadziej, możemy także utworzyć kod metody przeciążania operatorów `__delattr__` (która w ten sam sposób musi unikać pętli) w celu przechwycenia operacji usunięcia atrybutów. W przeciwieństwie do tego rozwiązania właściwości i deskryptory z założenia przechwytyują operacje pobierania, ustawiania oraz usuwania.

Metody `__getattr__` i `__setattr__` wprowadzone zostały w rozdziałach 30. oraz 32.; o `__getattribute__` wspomnieliśmy krótko w rozdziale 32. Tutaj rozszerzymy jedynie informacje o ich zastosowaniach i omówimy ich rolę w szerszym kontekście.

## Podstawy

W skrócie, jeśli klasa definiuje lub dziedziczy poniższe metody, zostaną one wykonane automatycznie, jeżeli instancja zostanie użyta w kontekście opisanym za pomocą komentarzy znajdujących się z prawej strony:

```
def __getattr__(self, nazwa):          # Przy pobraniu niezdefiniowanego atrybutu [obiekt.nazwa]
def __getattribute__(self, nazwa):     # Przy pobraniu wszystkich atrybutów [obiekt.nazwa]
def __setattr__(self, nazwa, wartość): # Przy przypisaniu wszystkich atrybutów
                                         # [obiekt.nazwa=wartość]
def __delattr__(self, nazwa):          # Przy usunięciu wszystkich atrybutów[del obiekt.nazwa]
```

W każdej z metod `self` jest jak zwykle obiektem instancji docelowej, `nazwa` to nazwa atrybutu, do którego dostęp ma miejsce, a `wartość` to obiekt przypisywany do atrybutu. Dwie metody pobierania normalnie zwracają wartość atrybutu, natomiast pozostałe dwie nie zwracają niczego (`None`). Wszystkie metody mogą zgłaszać wyjątki oznaczające odmowę dostępu.

Przykładowo w celu przechwycenia operacji pobrania każdego atrybutu możemy użyć dowolnej z dwóch pierwszych metod wymienionych wyżej, natomiast by przechwycić przypisanie każdego atrybutu, możemy skorzystać z trzeciej. Poniższy kod do pobierania wykorzystuje metodę `__getattr__`:

```
class Catcher:
    def __getattr__(self, name):
        print('Pobranie:', name)
```

```
def __setattr__(self, name, value):
    print('Ustawienie:', name, value)

X = Catcher()
X.job                # Wyświetla "Pobranie: job"
X.pay                # Wyświetla "Pobranie: pay"
X.pay = 'chleb'      # Wyświetla "Ustawienie: pay na chleb"
```

W tym konkretnym przypadku metodę `__getattribute__` można wykorzystać w taki sam sposób. Pojawia się potencjalne, trudno uchwytnie niebezpieczeństwo zapętlenia kodu, o czym będzie mowa w następnym podrozdziale:

```
class Catcher(object):
    def __getattribute__(self, name):
        print('Pobranie: name')
        ...pozostała część bez zmian...

# W wersji 2.x wymagane użycie (object)
# Metoda działa tak samo jak getattr
# Niebezpieczeństwo zapętlenia
```

Takie struktury kodu można wykorzystać do implementowania wzorca kodu delegacji, z którym spotkaliśmy się wcześniej, w rozdziale 31. Ponieważ wszystkie atrybuty przekierowywane są do naszych metod przechwytyjących w sposób ogólny, możemy sprawdzać ich poprawność i przekazywać je do osadzonych, zarządzanych obiektów. Poniższa klasa (zapożyczona z rozdziału 31.) śledzi na przykład *każdą* operację pobrania atrybutu wykonaną dla innego obiektu przekazanego do klasy opakowującej:

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object
    def __getattr__(self, attrname):
        print('Śledzenie:', attrname)
        return getattr(self.wrapped, attrname)

# Zapisanie obiektu
# Śledzenie pobrania
# Delegacja pobrania

X = Wrapper([1, 2, 3])
X.append(4)
print(X.wrapped)

# Wyświetlenie "Śledzenie: append"
# Wyświetlenie "[1, 2, 3, 4]"
```

Taka analogia nie występuje w przypadku właściwości oraz deskryptorów, z wyjątkiem pisania kodu akcesorów dla *każdego możliwego* atrybutu w *każdym możliwym* opakowanym obiekcie. Z drugiej strony, jeżeli taka ogólność nie jest wymagana, podstawowe metody dostępne mogą być niepotrzebnie dodatkowo wywoływane podczas przypisywania wartości atrybutom. Jest to kompromis opisany w rozdziale 30. i wspomniany w przykładzie opisanym na końcu tego rozdziału.

## Unikanie pętli w metodach przechwytyjących atrybuty

Metody te są stosunkowo proste w użyciu. Jedynym skomplikowanym elementem jest w nich potencjalna możliwość *zapętlenia* (czyli rekurencyjności). Ponieważ metoda `__getattr__` wywoływana jest jedynie dla atrybutów niezdefiniowanych, może swobodnie pobierać inne atrybuty w ramach swojego kodu. Ponieważ jednak metody `__getattribute__` oraz `__setattr__` wykonywane są dla wszystkich atrybutów, musimy być ostrożni przy dostępie do innych atrybutów, by uniknąć ponownego wywoływania ich wzajemnie i uruchomienia rekurencyjnej pętli.

Przykładowo kolejne pobranie atrybutu wewnątrz kodu metody `__getattr__` spowoduje ponowne wywołanie tej metody, a kod zapętlę się, dopóki nie wyczerpie się pamięć:

```
def __getattr__(self, name):
    x = self.other # PĘTLA!
```

Powyższa metoda jest bardziej narażona na zapętlenie, niż się na pierwszy rzut oka wydaje. Odwołanie do atrybutu `self` w dowolnym miejscu klasy zawierającej powyższą metodę skutkuje wywołaniem metody `__getattr__` i potencjalnym zapętleniem, w zależności od wpisanego kodu klasy. Zadaniem tej metody jest przechwytywanie *wszystkich* odczytów wartości atrybutu, ale należy jej używać ostrożnie, ponieważ obejmuje swoim działaniem wszystkie atrybuty. Jeżeli w niej samej zakoduje się odczytywanie atrybutu, wtedy zapętlenie jest niemal pewne.

Aby temu zapobiec, należy przekierować operację pobrania za pośrednictwem klasy nadrzędnej wyżej w hierarchii, zamiast przechodzić do wersji z klasy z tego poziomu — klasa `object` zawsze będzie klasą nadrzędną i świetnie się sprawdza w tej roli:

```
def __getattr__(self, name):
    x = object.__getattr__(self, 'other') # Wymuszenie klasy wyżej w celu uniknięcia siebie
```

W przypadku metody `__setattr__` sytuacja jest podobna (krótko zostało to opisane w rozdziale 30.). Przypisanie dowolnego atrybutu wewnątrz tej metody wywołuje ponownie `__setattr__` i tworzy podobną pętlę:

```
def __setattr__(self, name, value):
    self.other = value # Rekurencja (może się ZAPĘTLIĆ!)
```

Tutaj sytuacja jest podobna. Przypisanie wartości atrybutowi `self` w dowolnym miejscu klasy zawierającej powyższą metodę skutkuje wywołaniem metody `__setattr__`. Jednak prawdopodobieństwo zapętlenia jest znacznie większe, jeżeli przypisanie ma miejsce w samej metodzie `__setattr__`. By obejść ten problem, należy zamiast tego przypisać atrybut jako klucz do słownika przestrzeni nazw instancji `__dict__`. Pozwala to uniknąć bezpośredniego przypisania atrybutu:

```
def __setattr__(self, name, value):
    self.__dict__['other'] = value # Użycie słownika atrybutów w celu uniknięcia siebie
```

Choć jest to rzadziej stosowane rozwiązanie, metoda `__setattr__` może także przekazywać własne przypisania atrybutów do klasy nadrzędnej wyżej w hierarchii w celu uniknięcia pętli, tak samo jak metoda `__getattr__`. Ten schemat jest czasami wybierany, gdy klasy opakowane używają *slotów*, *właściwości* lub innych „wirtualnych” atrybutów, które istnieją na poziomie klas zamiast instancji — a w przypadku slotów mogą wykluczać użycie `__dict__`:

```
def __setattr__(self, name, value):
    object.__setattr__(self, 'other', value) # Wymuszenie wyższej klasy nadrzędnej
                                              # w celu uniknięcia siebie
```

Krótsze przykłady z `__setattr__` w tej książce często używają `__dict__`, ponieważ ich parametry są znane. Inaczej sytuacja przedstawia się w przypadku metody `__getattr__` — w celu uniknięcia pętli nie możemy użyć sztuczki ze słownikiem `__dict__`:

```
def __getattr__(self, name):
    x = self.__dict__['other'] # PĘTLA!
```

Pobranie samego atrybutu `__dict__` powoduje ponowne wywołanie metody `__getattr__`, co wywołuje pętlę rekurencyjną. Dziwne, ale prawdziwe!

Metoda `__delattr__` w praktyce wykorzystywana jest rzadko, jeśli jednak tak jest, wywoływana jest dla *każdego* usunięcia atrybutu (tak samo jak `__setattr__` wywoływana jest dla każdego przypisania atrybutu). Tym samym należy uważać, by unikać pętli przy usuwaniu atrybutów, używając do tego tych samych technik — słowników przestrzeni nazw lub wywołań metod z klasy nadrzędnej.

## Pierwszy przykład

Wszystko to nie jest wcale aż tak skomplikowane, jak mogłoby wynikać z powyższych informacji. By pokazać, jak wykorzystać te koncepcje w praktyce, zamieszczamy przykład 38.9, który jest taki sam jak przykład dla właściwości i deskryptorów — tym razem zaimplementowany za pomocą metod przeciążania operatorów atrybutów. Ponieważ metody te są tak ogólne, sprawdzamy tutaj nazwy atrybutów, by wiedzieć, że odbywa się dostęp do atrybutu zarządzanego. Pozostałe mogą być przekazywane normalnie.

Przykład 38.9. *getattr-person.py*

```
class Person:
    def __init__(self, name):           # Dla [Person()]
        self._name = name              # Wywołuje __setattr__!

    def __getattr__(self, attr):        # Dla [obiekt.niezdefiniowany]
        if attr == 'name':             # Przechwycenie name: nie przechowano
            print('pobieranie...')
            return self._name          # Nie tworzy pętli: prawdziwy atrybut
        else:                           # Pozostałe są błędami
            raise AttributeError(attr)

    def __setattr__(self, attr, value): # Dla [obiekt.dowolny = wartość]
        if attr == 'name':
            print('modyfikacja...')
            attr = '_name'              # Ustawienie nazw wewnętrznych
        self.__dict__[attr] = value    # Tutaj unikanie pętli

    def __delattr__(self, attr):        # Dla [del obiekt.dowolny]
        if attr == 'name':
            print('usunięcie..')
            attr = '_name'              # Tutaj także unikanie pętli
        del self.__dict__[attr]        # jednak jest to o wiele radsze

bob = Person('Robert Zielony')         # bob ma zarządzany atrybut
print(bob.name)                        # Wykonuje __getattr__
bob.name = 'Robert A. Zielony'         # Wykonuje __setattr__
print(bob.name)
del bob.name                           # Wykonuje __delattr__

print('-'*20)
anna = Person('Anna Czerwona')        # anna także dziedziczy właściwość
print(anna.name)
#print(Person.name.__doc__)            # Tutaj brak odpowiednika
```



Po wykonaniu kodu zwracany jest ten sam wynik, jednak tym razem jest to efekt normalnego mechanizmu przeciążania operatorów Pythona oraz naszych metod przechwytywania atrybutów:

```
$ python3 getattr-person.py
pobieranie...
Robert Zielony
modyfikacja...
pobieranie...
Robert A. Zielony
usunięcie...
-----
pobieranie...
Anna Czerwona
```

Warto zauważyć, że przypisanie atrybutu w konstruktorze `__init__` także wywołuje metodę `__setattr__` — metoda ta przechwytuje *wszystkie* przypisania atrybutów, nawet te wewnątrz samej klasy oraz te, które dotyczą ukrytych atrybutów, takich jak `_name`. Warto również zauważyć, że w przeciwieństwie do właściwości i deskryptorów nie istnieje bezpośredni sposób podawania tutaj *dokumentacji* naszego atrybutu — zarządzane atrybuty istnieją wewnątrz kodu metod przechwytyjących, a nie jako odrębne obiekty.

## Metoda `__getattribute__`

By uzyskać dokładnie taki sam rezultat za pomocą metody `__getattribute__`, należy zastąpić `__getattr__` w przykładzie 38.9 kodem z przykładu 38.10. Ponieważ przechwytuje on pobranie *wszystkich* atrybutów, wersja ta musi uważać na unikanie pętli, przekazując nowe operacje pobierania do klasy nadrzędnej. Nie może także zakładać, że wszystkie nieznane nazwy są błędami.

Przykład 38.10. `getattribute-person.py` (różniąca się część)

# Należy zastąpić metodę `__getattr__` za pomocą poniższej

```
def __getattribute__(self, attr):
    print('pobieranie: ' + attr)
    if attr == 'name':
        attr = '_name'
    return object.__getattribute__(self, attr)
```

# Dla [obiekt.dowolny]  
# Przechwycenie wszystkich nazw  
# Odzworowanie na nazwę wewnętrzną  
# Tutaj unikanie pętli

Po uruchomieniu powyższego kodu uzyskamy podobny wynik jak poprzednio. Różnica będzie polegała jedynie na tym, że metoda `__getattribute__` zostanie wywołana więcej razy w wyniku odczytania wartości atrybutu wewnątrz metody `__setattr__` (zainicjowanego w metodzie `__init__`):

```
$ python3 getattribute-person.py
pobieranie: __dict__
pobieranie: name
Robert Zielony
modyfikacja...
pobieranie: __dict__
pobieranie: name
Robert A. Zielony
```

```

usuniecie..
pobieranie: __dict__
-----
pobieranie: __dict__
pobieranie: name
Anna Czerwona

```

Powyższy przykład jest odpowiednikiem rozwiązania z właściwościami i deskryptorami, jednak jest nieco sztuczny i tak naprawdę nie przedstawia działania tych narzędzi w praktyce. Metody `__getattr__` oraz `__getattribute__`, ponieważ są tak bardzo ogólne, częściej wykorzystywane są w kodzie opartym na mechanizmie delegacji (zgodnie z informacjami przedstawionymi wcześniej), gdzie dostęp do atrybutów jest sprawdzany i przekierowywany do osadzonego obiektu. Tam, gdzie musimy zarządzać *pojedynczym* atrybutem, właściwości i deskryptory sprawdzają się równie dobrze lub nawet lepiej i pozwalają unikać dodatkowych wywołań dla atrybutów niezarządzanych.

## Obliczanie atrybutów

Tak jak wcześniej, nasz poprzedni przykład nie robi nic specjalnego poza śledzeniem operacji pobrania atrybutów. Obliczenie wartości atrybutu przy jego pobraniu nie stanowi szczególnie większego wyzwania. Tak jak w przypadku właściwości i deskryptorów, przykład 38.11 tworzy wirtualny atrybut `x`, który po pobraniu wykonuje obliczenia.

Przykład 38.11. *getattr-computed.py*

```

class AttrSquare:
    def __init__(self, start):
        self.value = start                # Wywołuje __setattr__!

    def __getattr__(self, attr):          # Przy pobraniu niezdefiniowanego atrybutu
        if attr == 'x':
            return self.value ** 2       # Wartość nie jest niezdefiniowana
        else:
            raise AttributeError(attr)

    def __setattr__(self, attr, value):  # Przy przypisaniu wszystkich atrybutów
        if attr == 'x':
            attr = 'value'
        self.__dict__[attr] = value

A = AttrSquare(3)                       # 2 instancje klasy z przeciążaniem operatorów
B = AttrSquare(32)                      # Każda ma inne informacje o stanie

print(A.x)                             # 3 ** 2
A.x = 4
print(A.x)                             # 4 ** 2
print(B.x)                             # 32 ** 2

```

Wykonanie powyższego kodu powoduje zwrócenie tego samego wyniku, jaki otrzymaliśmy wcześniej za pomocą właściwości i deskryptorów, jednak mechanizm tego skryptu oparty jest na metodach przechwytywania ogólnych atrybutów:

```
$ python3 getattr-computed.py
9
16
1024
```

## Metoda `__getattribute__`

Tak jak wcześniej, ten sam rezultat możemy uzyskać za pomocą użycia metody `__getattribute__` w miejsce `__getattr__`. Przykład 38.12 zastępuje metodę pobrania za pomocą `__getattribute__` i zmienia metodę przypisania `__setattr__` w taki sposób, by uniknąć pętli, wykorzystując do tego bezpośrednio wywołania metod klasy nadrzędnej zamiast kluczy słownika `__dict__`.

Przykład 38.12. *getattrattribute-computed.py*

```
class AttrSquare:
    def __init__(self, start):
        self.value = start          # Wywołuje __setattr__!

    def __getattribute__(self, attr): # Przy pobraniu wszystkich atrybutów
        if attr == 'X':
            return self.value ** 2   # Znowu wywołuje __getattribute__!
        else:
            return object.__getattribute__(self, attr)

    def __setattr__(self, attr, value): # Przy przypisaniu wszystkich atrybutów
        if attr == 'X':
            attr = 'value'
            object.__setattr__(self, attr, value)
```

*...kod testujący taki sam jak w przykładzie 38.11...*

Po wykonaniu powyższej wersji kodu otrzymany wynik będzie ten sam. Warto zwrócić uwagę na niejawne przekierowanie, które występuje wewnątrz metod tej klasy:

- `self.value=start` wewnątrz konstruktora wywołuje metodę `__setattr__`,
- `self.value` wewnątrz metody `__getattribute__` wywołuje ponownie metodę `__getattribute__`.

Tak naprawdę metoda `__getattribute__` za każdym razem, gdy pobieramy atrybut `X`, wywoływana jest *dwa* razy. Takie zjawisko nie występuje w wersji z metodą `__getattr__`, ponieważ atrybut `value` nie jest niezdefiniowany. Jeśli szybkość kodu ma dla nas znaczenie i chcemy tego uniknąć, należy zmodyfikować metodę `__getattribute__` w taki sposób, by do pobrania `value` również wykorzystywała klasę nadrzędną:

```
def __getattribute__(self, attr):
    if attr == 'X':
        return object.__getattribute__(self, 'value') ** 2
```

Oczywiście to rozwiązanie nadal powoduje wywołanie metody z klasy nadrzędnej, jednak bez dodatkowego wywołania rekurencyjnego, zanim tam dotrzemy. W celu prześledzenia tego, kiedy i jak wywoływane są te metody, warto dodać wywołania `print`.

## Porównanie metod `__getattr__` i `__getattribute__`

W celu podsumowania różnic w kodzie metod `__getattr__` oraz `__getattribute__` przykład 38.13 wykorzystuje obydwie z nich w celu zaimplementowania trzech atrybutów: `attr1` jest atrybutem klasy, `attr2` jest atrybutem instancji, natomiast `attr3` jest wirtualnym atrybutem zarządzanym, obliczanym przy pobraniu.

Przykład 38.13. *getattr-v-getattribute.py*

```
class GetAttr:
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattr__(self, attr):
        print('pobieranie: ' + attr)
        if attr == 'attr3':
            return 3
        else:
            raise AttributeError(attr)
    # Tylko dla niezdefiniowanych atrybutów
    # Nie attr1: dziedziczony z klasy
    # Nie attr2: przechowany w instancji

X = GetAttr()
print(X.attr1)
print(X.attr2)
print(X.attr3)
print('-'*40)

class GetAttribute:
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattribute__(self, attr):
        print('pobieranie: ' + attr)
        if attr == 'attr3':
            return 3
        else:
            return object.__getattribute__(self, attr)
    # Dla pobrań wszystkich atrybutów
    # Tutaj użycie klasy nadrzędnej w celu uniknięcia pętli

X = GetAttribute()
print(X.attr1)
print(X.attr2)
print(X.attr3)
```

Po wykonaniu wersja z metodą `__getattr__` przechwytuje jedynie dostęp do atrybutu `attr3`, ponieważ jest on niezdefiniowany. Wersja z metodą `__getattribute__` przechwytuje z kolei operacje pobierania wszystkich atrybutów i musi przekierowywać te, którymi nie zarządza, do klasy nadrzędnej w celu uniknięcia pętli:

```
$ python3 py -3 getattr-v-getattr.py
1
2
pobieranie: attr3
3
-----
pobieranie: attr1
1
pobieranie: attr2
```

```
2
pobieranie: attr3
3
```

Choć metoda `__getattr__` może przechwytywać więcej operacji pobierania atrybutów od `__getattribute__`, w praktyce często są one stosowane wymiennie — jeśli atrybuty nie są fizycznie przechowywane, obie metody dają ten sam efekt.

## Porównanie technik zarządzania atrybutami

W celu podsumowania różnic w kodzie wszystkich czterech zaprezentowanych w niniejszym rozdziale technik zarządzania atrybutami przejdźmy szybko przez bardziej rozbudowany przykład z obliczaniem atrybutów z wykorzystaniem każdej z technik. Pierwsza wersja (przykład 38.14) wykorzystuje *właściwości* do przechwytywania i obliczania atrybutów o nazwie `square` oraz `cube`. Warto zwrócić uwagę na to, jak ich wartości bazowe zostają przechowane w zmiennych rozpoczynających się od znaku `_`, tak by nie wchodziły one w konflikt z samymi nazwami właściwości.

Przykład 38.14. *all\_four\_props.py*

*"Dwa dynamicznie obliczane atrybuty z właściwościami"*

```
class Powers:
    def __init__(self, square, cube):
        self._square = square          # _square to wartość bazowa
        self._cube = cube              # square to nazwa właściwości

    def getSquare(self):
        return self._square ** 2
    def setSquare(self, value):
        self._square = value
    square = property(getSquare, setSquare)  # lub dekorator @property

    def getCube(self):
        return self._cube ** 3
    cube = property(getCube)               # Podobnie
```

By uzyskać to samo za pomocą *deskryptorów*, w przykładzie 38.15. definiujemy atrybuty za pomocą pełnych klas. Warto zwrócić uwagę na to, że poniższe deskryptory przechowują wartości bazowe jako stan instancji, przez co znowu muszą korzystać z początkowych znaków `_` w celu uniknięcia konfliktu z nazwami deskryptorów. Jak zobaczymy w ostatnim przykładzie rozdziału, moglibyśmy uniknąć konieczności zmiany nazwy, przechowując zamiast tego wartości bazowe w postaci stanu deskryptora. W takim przypadku jednak nie można byłoby przechowywać danych, które różniłyby się w poszczególnych instancjach.

Przykład 38.15. *all\_four\_desc.py*

*"To samo, ale z wykorzystaniem deskryptorów (na stan instancji)"*

```
class DescSquare:
    def __get__(self, instance, owner):
        return instance._square ** 2
```

```

def __set__(self, instance, value):
    instance._square = value

class DescCube:
    def __get__(self, instance, owner):
        return instance._cube ** 3

class Powers:
    square = DescSquare()
    cube = DescCube()
    def __init__(self, square, cube):
        self._square = square          # "self.square = square" także działa,
        self._cube = cube              # ponieważ wywołuje metodę __set__ deskryptora!

```

By uzyskać ten sam rezultat za pomocą metody przechwytywania pobierania `__getattr__`, przykład 38.16 przechowuje wartości bazowe za pomocą zmiennych poprzedzonych znakiem `_`, tak by operacje dostępu do zarządzanych zmiennych pozostały niezdefiniowane i tym samym wywołały naszą metodę. Trzeba również utworzyć kod metody `__setattr__` w celu przechwylenia operacji przypisania, a także uważać na jej potencjalne zapętlenie.

Przykład 38.16. *all\_four\_getattr.py*

*"To samo, ale z ogólnym przechwytywaniem niezdefiniowanego atrybutu \_\_getattr\_\_"*

```

class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube = cube

    def __getattr__(self, name):
        if name == 'square':
            return self._square ** 2
        elif name == 'cube':
            return self._cube ** 3
        else:
            raise TypeError('nieznany atrybut:' + name)

    def __setattr__(self, name, value):
        if name == 'square':
            self.__dict__['_square'] = value      # Lub użyj object
        else:
            self.__dict__[name] = value

```

Ostatnie rozwiązanie — przykład 38.17 wykorzystujący metodę `__getattribute__` — jest podobne do wersji poprzedniej. Ponieważ jednak przechwytywamy teraz każdy atrybut, musimy przekierować pobrania wartości bazowych do klasy nadrzędnej w celu uniknięcia zapętlenia. Więcej razy jest jednak wywoływana metoda `__getattribute__`.

Przykład 38.17. *all\_four\_getattribute.py*

*"To samo, ale z ogólnym przechwytywaniem wszystkich atrybutów za pomocą \_\_getattribute\_\_"*

```

class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube = cube

```

```

def __getattr__(self, name):
    if name == 'square':
        return object.__getattr__(self, '_square') ** 2
    elif name == 'cube':
        return object.__getattr__(self, '_cube') ** 3
    else:
        return object.__getattr__(self, name)

def __setattr__(self, name, value):
    if name == 'square':
        object.__setattr__(self, '_square', value)           # Lub użyj __dict__
    else:
        object.__setattr__(self, name, value)

```

W celu przetestowania uruchomiono kod pokazany poniżej w sesji REPL. Kod ten przechodzi przez listę łańcuchów znaków nazw wszystkich czterech modułów, importuje je i pobiera klasy. Każda technika przybiera inną formę w kodzie, ale wszystkie cztery dają ten sam wynik po uruchomieniu:

```

>>> from importlib import import_module
>>> mods = [f'all_four_{M}' for M in ('props', 'desc', 'getattr', 'getattrattribute')]
>>> for modname in mods:
    module = import_module(modname)           # Import poprzez nazwę jako łańcuch znaków
    X = module.Powers(3, 4)                   # Klasa tego modułu (wyświetl w celu podglądu)
    print(X.square)                           # 3 ** 2 = 9
    print(X.cube)                             # 4 ** 3 = 64
    X.square = 5
    print(X.square)                           # 5 ** 2 = 25

```

9  
64  
25  
...powtórzone cztery razy...

Więcej informacji na temat porównania tych alternatyw, a także innych opcji, znajdzie się w bardziej realistycznym ich zastosowaniu w przykładzie ze sprawdzaniem poprawności atrybutów w podrozdziale „Przykład — sprawdzanie poprawności atrybutów”. Najpierw jednak musimy omówić pewną pułapkę związaną z dwoma z powyższych narzędzi — ogólne przechwytywanie atrybutów.

## Przechwytywanie atrybutów wbudowanych operacji

Jeżeli czytasz po kolei wszystkie rozdziały tej książki, w niektórych fragmentach znajdziesz podsumowanie materiału opisanego wcześniej, głównie w ramce „Delegować czy nie atrybuty wbudowane?” z rozdziału 28. Kiedy wprowadzono metody `__getattr__` i `__getattrattribute__`, stwierdzono, że one odpowiednio przechwytyują pobieranie niezdefiniowanych atrybutów oraz wszystkich atrybutów, co sprawia, że idealnie sprawdzają się we wzorcach kodu opartych na delegacji.

Choć jest to prawdą dla atrybutów o *normalnych nazwach*, ich działanie zasługuje na dodatkowe wyjaśnienie. W przypadku atrybutów o nazwach metod pobieranych w sposób niejawny

przez *operacje wbudowane* metody te mogą w ogóle nie być wykonane. Oznacza to, że wywołania metod przeciążających operatory nie mogą być delegowane do opakowanych obiektów, o ile klasy opakowujące w jakiś sposób same nie zdefiniują tych metod ponownie.

Przykładowo pobranie atrybutów dla metod `__str__`, `__add__` oraz `__getitem__` wykonywane w sposób *niejawny* za pomocą, odpowiednio, wyświetlania, wyrażeń ze znakiem `+` oraz indeksowania nie jest przekierowywane do ogólnych metod przechwytywania atrybutów. W związku z tym nie ma bezpośredniego sposobu na ogólne przechwytywanie i delegowanie wbudowanych operacji tego typu.

To rozróżnienie było obecne w Pythonie 3.X, którego domniemaną racją były metaklasy i optymalizacja wbudowanych operacji. Niezależnie od powodów wszystkie atrybuty — zarówno `__X__`, jak i inne — są nadal przekazywane przez metody przechwytywania instancji, gdy są wywoływane bezpośrednio po nazwie, co kwalifikuje się jako rażąca niespójność: `X.__add__` uruchamia `__getattr__`, ale `X+Y`, które korzysta z `X.__add__`, już nie. W efekcie wpływa to na komplikację kodu opartego na delegacji.

Dobra wiadomość jest taka, że klasy opakowujące mogą obejść to ograniczenie, redefiniując wszystkie niezbędne metody przeciążania operatorów w samej klasie opakowującej w celu delegowania wywołań. Te dodatkowe metody można dodawać albo ręcznie, za pomocą narzędzi, albo za pomocą definicji we wspólnych klasach nadrzędnych i dziedziczeniu po nich. Takie rozwiązanie sprawia jednak, że klasy opakowujące wymagają więcej pracy, jeśli metody przeciążania operatorów są częścią interfejsu opakowanego obiektu.

Rozważmy przykład 38.18, który sprawdza różne typy atrybutów i operacji wbudowanych na instancjach klas zawierających metody `__getattr__` oraz `__getattribute__`.

Przykład 38.18. *getattr-builtins.py*

```
class GetAttr:
    cattr = 88                                # Atrybuty przechowywane w klasie i instancji
    def __init__(self):                       # To pomija metodę getattr, ale nie getattribute
        self.iattr = 77

    def __len__(self):                        # Ponownie zdefiniowane len(), nie wywołuje getattr
        print('__len__: 66')
        return 66

    def __getattr__(self, attr):              # Dostarczenie __str__ po żądaniu, inaczej pusta funkcja
        print('getattr: ' + attr)            # Nigdy nie uruchamiane dla __str__ — dziedziczone z object
        if attr == '__str__':
            return lambda *args: '[Getattr str]'
        else:
            return lambda *args: None

class GetAttribute:
    cattr = 88                                # Podobnie, ale przechwytytuje wszystkie atrybuty
    def __init__(self):                       # Z wyjątkiem niejawnych pobrań dla wbudowanych operacji
        self.iattr = 77

    def __len__(self):                        # Ponownie zdefiniowane len(), nie wywołuje getattribute
        print('__len__: 66')                 # Ale jawne pobrania dziedziczonego __str__ tak
```



```

        return 66

    def __getattr__(self, attr):
        print('getattr: ' + attr)
        if attr == '__str__':
            return lambda *args: '[GetAttribute str]'
        else:
            return lambda *args: None

for Class in GetAttr, GetAttribute:
    print('\n' + Class.__name__.ljust(50, '='))

X = Class()

# Zdefiniowane atrybuty wywołują getattr, ale nie getattr

X.cattr                # Atrybut klasy (zdefiniowany, pomija getattr)
X.iattr                # Atrybut instancji (zdefiniowany, pomija getattr)
X.other                # Brakujący atrybut
len(X)                 # __len__ definiowane w sposób bezpośredni

# Operacje wbudowane nie wywołują ani getattr, ani getattr
# Żadne domyślne atrybuty nie są dziedziczone po obiekcie klasy nadrzędnej

try:                    # Próbuje wywołać __getitem__
    X[0]
except:
    print('niepowodzenie []')

try:
    X + 99
except:                  # Jak wyżej, tylko że __add__
    print('niepowodzenie +')

try:
    X()
except:                  # Jak wyżej, tylko że __call__
    print('niepowodzenie ()')

# Ale jawne wywołania uruchamiają oba przechwytywania

X.__getitem__(0)
X.__add__(99)
X.__call__()

# Domyślna klasa nadrzędna obiektu definiuje metodę __str__, która wyklucza getattr
# Jednak bezwzględna metoda getattr również nie jest wywoływana dla niejawnych pobrań

print(X.__str__())      # __str__: jawne wywołanie => tylko __getattr__ jest pomijane
print(X)                # __str__: niejawne poprzez wbudowaną funkcję => oba pomijane

```

Ten plik kolejno uruchamia ten sam zestaw testów na każdej z jego klas. Dopasuj poniższe wyniki do testów i komentarzy, aby zobaczyć, jak to działa. W skrócie: ani `__getattr__`, ani `__getattr__` nie są uruchamiane dla żadnych nazw przeciążonych operatorów wywoływanych przez wbudowane operacje, ponieważ takie nazwy są szukane tylko w klasach:

```
$ python3 getattr-builtins.py
```

```
GetAttr=====
```

```

getattr: other
__len__: 66
niepowodzenie []
niepowodzenie +
niepowodzenie ()
getattr: __getitem__
getattr: __add__
getattr: __call__
<__main__.GetAttr object at 0x025D17F0>
<__main__.GetAttr object at 0x025D17F0>

GetAttribute=====
getattr: cattr
getattr: iattr
getattr: other
__len__: 66
niepowodzenie []
niepowodzenie +
niepowodzenie ()
getattr: __getitem__
getattr: __add__
getattr: __call__
getattr: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x025D1870>

```

Bardziej ogólnie, wszystkie *jawne* pobrania atrybutów o nazwach metod są zawsze kierowane do obu metod przechwytywania atrybutów, ale żadne z *niejawnych* metod przeciążających operatory nie uruchamiają tych metod przechwytywania atrybutów, gdy ich atrybuty są pobierane przez wbudowane operacje. Oto kluczowe punkty tego przykładu, które warto wyróżnić:

- Próba przechwycenia `__str__` przez metodę `__getattr__` kończy się niepowodzeniem — raz dla wbudowanego wyświetlania, drugi raz dla jawnego pobrania, ponieważ zachowanie domyślne dziedziczone jest po wbudowanej klasie `object`, będącej klasą nadrzędną dla wszystkich.
- Próba przechwycenia `__str__` przez metodę przechwytyjącą wszystko `__getattribute__` kończy się niepowodzeniem podczas wbudowanej operacji wyświetlania. Pobranie w sposób jawny pomija odziedziczone `__str__` i uruchamia `__getattribute__`.
- Próba przechwycenia `__call__` kończy się niepowodzeniem dla wbudowanych wyrażeń wywołania, jednak przechwycenie jest możliwe w obu metodach w przypadku pobrania w sposób jawny. W przeciwieństwie do `__str__` nie istnieje żadna domyślna odziedziczona metoda `__call__`, która miałaby przewagę nad `__getattr__` w jawnych pobraniach. To samo dotyczy `__add__` i operacji dodawania.
- Obie klasy przechwytyją `__len__` — dlatego, że jest to metoda zdefiniowana w sposób jawny w samych klasach. Jej nazwa nie jest przekierowywana ani do `__getattr__`, ani do `__getattribute__`, jeśli usuniemy metody `__len__` klasy, ponieważ wbudowana funkcja `len` standardowo pomija je wszystkie.

I znów, w rezultacie metody przeciążania operatorów wykonywane w sposób niejawni przez operacje wbudowane nigdy nie są przekierowywane za pośrednictwem żadnej z metod przechwytywania atrybutów. Python szuka takich atrybutów w *klasach* i całkowicie pomija wyszukiwanie w instancjach. Zazwyczaj nazwy atrybutów i jawne pobrania zaczynają się od in-szacji.

By zobaczyć bardziej realistyczny przykład tego zjawiska wraz z obchodzącym ten problem rozwiązaniem, warto zajrzeć do przykładu z dekoratorem `Private` z kolejnego rozdziału wraz z omówieniem wielu *obejść* możliwych do ponownego użycia.

## Powrót przykładu delegacji z rozdziału 28.

Teraz powinienś być w stanie zrozumieć, dlaczego klasa `Manager` w przykładzie 28.11 z rozdziału 28. musiała zdefiniować metodę `__repr__`, aby przekierować żądania wyświetlania do opakowanego obiektu. Podobnie jak `__str__` w naszym przykładzie, klasa `object` dostarcza domyślną metodę `__repr__`, która uniemożliwiłaby operacjom `print` wywołanie `__getattr__`. Technicznie rzecz biorąc, klasa `object` definiuje zarówno `__str__`, jak i `__repr__`, ale jej `__str__` po prostu wywołuje `__repr__`.

Domyślne ustawienia klasy `object` są w dużej mierze bez znaczenia, podobnie jak wszystkie wbudowane operacje. Funkcja `print` omija zarówno `__getattr__`, jak i `__getattribute__`, tak jak to miało miejsce w naszym przykładzie dla `__getattribute__`. Z tego powodu metoda `__repr__` jest wymagana *zarówno* przez domyślną implementację `object`, jak i przez sposób działania wbudowanej funkcji.

Rozwiązania dotyczące delegowania operacji wbudowanych zostały omówione w rozdziale 39.

## Przykład: sprawdzanie poprawności atrybutów

Na zakończenie rozdziału przejdźmy do bardziej realistycznego przykładu, wykorzystującego w kodzie wszystkie cztery rozwiązania z zakresu zarządzania atrybutami. Przykład ten definiuje obiekt `CardHolder` z czterema atrybutami, z których trzy są zarządzane. Zarządzane atrybuty sprawdzają poprawność lub przekształcają dane po pobraniu albo przechowaniu. Wszystkie cztery wersje zwracają te same wyniki dla tego samego kodu testowego, jednak implementują atrybuty na bardzo różne sposoby. Przykłady są tutaj zamieszczone głównie z myślą o samodzielnym przestudiowaniu. Choć nie będę szczegółowo omawiał ich kodu, wykorzystują one koncepcje omawiane już w niniejszym rozdziale.

## Wykorzystywanie właściwości do sprawdzania poprawności

Nasz pierwszy kod w przykładzie 38.19 do zarządzania trzema atrybutami wykorzystuje właściwości. Jak zwykle moglibyśmy zamiast atrybutów zarządzanych zastosować proste metody, jednak właściwości są pomocne, jeśli w istniejącym kodzie wykorzystywaliśmy już atrybuty. Właściwości wykonują kod automatycznie w momencie dostępu do atrybutów, jednak skoncentrowane są na ściśle określonym ich zbiorze. Nie można ich wykorzystywać do przechwytywania wszystkich atrybutów w sposób uniwersalny.

By zrozumieć poniższy kod, kluczowe jest zwrócenie uwagi na to, że przypisania do atrybutów wewnątrz metody konstruktora `__init__` także wywołują metodę ustawiającą właściwości. Kiedy metoda ta przypisuje na przykład wartość do `self.name`, automatycznie wywołuje metodę `setName`, przekształcającą wartość i przypisującą ją do atrybutu instancji o nazwie `__name`, tak by nie wchodził on w konflikt z nazwą właściwości.

Taka *zmiana nazwy* jest niezbędna, ponieważ właściwości wykorzystują wspólny stan instancji i nie mają własnego. Dane przechowane są w atrybucie o nazwie `__name`, a atrybut o nazwie `name` jest zawsze właściwością, nie danymi. Jak widzieliśmy w rozdziale 31., nazwy takie jak `__name` oznaczają tzw. atrybuty *pseudoprywatne*. Gdy Python umieszcza je w przestrzeni nazw instancji, dołącza do nich nazwę klasy. Dzięki temu można odróżniać atrybuty charakterystyczne dla danej implementacji od innych atrybutów, m.in. od właściwości, które nimi zarządzają.

Podsumowując, klasa z przykładu 38.19 zarządza atrybutami o nazwach `name`, `age` oraz `acct`. Pozwala na bezpośredni dostęp do atrybutu `addr` i udostępnia atrybut tylko do odczytu o nazwie `remain`, który jest w pełni wirtualny i obliczany na żądanie. Na potrzeby porównania: kod przykładu oparty na właściwościach składa się z 39 wierszy (uwzględniając puste wiersze).

Przykład 38.19. *validate\_properties.py*

```
class CardHolder:
    acctlen = 8                               # Dane klasy
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                       # Dane instancji
        self.name = name                       # Te także wywołują metody ustawiające właściwości
        self.age = age                         # Zmiana nazwy __X w celu uzyskania nazwy klasy
        self.addr = addr                       # Atrybut addr nie jest zarządzany
                                                # Atrybut remain nie ma danych

    def getName(self):
        return self.__name
    def setName(self, value):
        value = value.lower().replace(' ', '_')
        self.__name = value
    name = property(getName, setName)           # Lub dekoratory @ dla obu

    def getAge(self):
        return self.__age
    def setAge(self, value):
        if value < 0 or value > 150:
            raise ValueError('niepoprawny wiek')
        else:
            self.__age = value
    age = property(getAge, setAge)

    def getAcct(self):
        return self.__acct[:-3] + '***'
    def setAcct(self, value):
        value = value.replace('-', '')
        if len(value) != self.acctlen:
```

```

        raise TypeError('niepoprawny numer konta')
    else:
        self.__acct = value
acct = property(getAcct, setAcct)

def remainGet(self):
    return self.retireage - self.age
remain = property(remainGet)

```

*# Mógłby być metodą, nie atrybutem*  
*# O ile niewykorzystywany jeszcze jako atrybut*

## Testowanie kodu

Aby przetestować klasę, należy uruchomić skrypt z przykładu 38.20 jednym poleceniem, podając w argumentcie nazwę modułu (bez *.py*). Możesz również zaimportować klasę w sesji REPL, ale staramy się tutaj nie powtarzać kodu. Dla wszystkich czterech wersji tego przykładu wykorzystamy ten sam kod sprawdzający, więc jego wyniki będą takie same. Po wykonaniu tworzymy dwie instancje klasy z zarządzanymi atrybutami, a także pobieramy i modyfikujemy jej różne atrybuty. Operacje, które mogą się nie powieść, opakowujemy w instrukcje try.

Przykład 38.20. *validate\_tester.py*

```

def loadclass():
    import sys, importlib
    modulename = sys.argv[1]
    module = importlib.import_module(modulename)
    print (f'[Using: {module.CardHolder}]')
    return module.CardHolder

```

*# Nazwa modułu w wierszu poleceń*  
*# Import modułu o podanej nazwie*  
*# Ta metoda getattr() nie jest potrzebna*

```

def printholder(who):
    print(who.acct, who.name, who.age, who.remain, who.addr, sep=' / ')

```

```

if __name__ == '__main__':
    CardHolder = loadclass()
    bob = CardHolder('1234-5678', 'Robert Zielony', 40, 'ul. Poziomkowa 15')
    print(bob.acct, bob.name, bob.age, bob.remain, bob.addr, sep=' / ')
    bob.name = 'Robert A. Zielony'
    bob.age = 50
    bob.acct = '23-45-67-89'
    printholder(bob)

    anna = CardHolder('5678-12-34', 'Anna Czerwona', 35, 'ul. Poziomkowa 16')
    printholder(anna)
    try:
        anna.age = 200
    except:
        print('Niepoprawny wiek dla Anny')

    try:
        anna.remain = 5
    except: print("Nie można ustawić anna.remain")

    try:
        anna.acct = '1234567'
    except: print('Niepoprawne konto dla Anny')

```

Poniżej znajdują się wyniki dla naszego kodu samosprawdzającego. I znów będą one takie same dla wszystkich wersji tego przykładu, inna będzie tylko nazwa testowanej klasy. Warto prześledzić kod w celu przekonania się, w jaki sposób wywoływane są metody klasy. Konta wyświetlane są z ukryciem niektórych znaków, imiona i nazwiska przekształcane są na ustandaryzowany format, a czas pozostały do emerytury obliczany jest przy pobraniu za pomocą atrybutu klasy:

```
$ python3 validate_tester.py validate_properties
[Using: <class 'validate_properties.CardHolder'>]
12345*** / robert_zielony / 40 / 19.5 / ul. Poziomkowa 15
23456*** / robert_a._zielony / 50 / 9.5 / ul. Poziomkowa 16
56781*** / anna_czerwona / 35 / 24.5 / ul. Poziomkowa 16
Niepoprawny wiek dla Anny
Nie można ustawić anna.remain
Niepoprawne konto dla Anny
```

## Wykorzystywanie deskryptorów do sprawdzania poprawności

Napiszmy teraz nową wersję kodu naszego przykładu z wykorzystaniem *deskryptorów* zamiast właściwości. Jak widzieliśmy, deskryptory są bardzo podobne do właściwości, jeśli chodzi o funkcjonalność i pełnione role. Tak naprawdę właściwości są ograniczoną formą deskryptora. Tak jak właściwości, deskryptory zaprojektowano z myślą o obsłudze określonych atrybutów, a nie uniwersalnego dostępu do atrybutów. W przeciwieństwie do właściwości deskryptory mają własny stan i są rozwiązaniem bardziej ogólnym.

### Opcja 1: sprawdzanie z wykorzystaniem współdzielonego stanu deskryptora instancji (niepoprawnie)

By zrozumieć kod z przykładu 38.21, istotne jest zauważenie, że przypisania atrybutów wewnątrz metody konstruktora `__init__` wywołują metody `__set__` deskryptora. Kiedy metoda konstruktora przypisuje na przykład wartość do `self.name`, automatycznie wywołuje metodę `Name.__set__()`, przekształcającą wartość i przypisującą ją do atrybutu deskryptora o nazwie `name`.

Przykład 38.21. *validate\_descriptors1.py*

```
class CardHolder:
    acctlen = 8
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct
        self.name = name
        self.age = age
        self.addr = addr

class Name:
    def __get__(self, instance, owner):
        return self.name
    def __set__(self, instance, value):
```

# Użycie dzielonego stanu deskryptora  
# Dane klasy  
  
# Dane instancji  
# Te także wywołują metodę \_\_set\_\_  
# Zmiana nazwy \_\_X nie jest konieczna: w deskryptorze  
# Atrybut addr nie jest zarządzany  
# Atrybut remain nie ma danych  
  
# Zmienne klasy: lokalne dla CardHolder

```

        value = value.lower().replace(' ', '_')
        self.name = value
name = Name()

class Age:
    def __get__(self, instance, owner):
        return self.age # Użycie danych deskryptora
    def __set__(self, instance, value):
        if value < 0 or value > 150:
            raise ValueError('niepoprawny wiek')
        else:
            self.age = value
age = Age()

class Acct:
    def __get__(self, instance, owner):
        return self.acct[:-3] + '***'
    def __set__(self, instance, value):
        value = value.replace('-', '')
        if len(value) != instance.acctlen: # Użycie danych instancji klasy
            raise TypeError('niepoprawny numer konta')
        else:
            self.acct = value
acct = Acct()

class Remain:
    def __get__(self, instance, owner):
        return instance.retireage - instance.age # Wywołuje Age.__get__
    def __set__(self, instance, value):
        raise TypeError('nie można ustawić remain') # Inaczej zezwolilibyśmy na ustawienie
remain = Remain()

```

Wreszcie klasa ta implementuje te same atrybuty co wersja poprzednia — zarządza atrybutami o nazwach `name`, `age` oraz `acct`. Pozwala także na bezpośredni dostęp do atrybutu `addr` i udostępnia atrybut tylko do odczytu o nazwie `remain`, który jest w pełni wirtualny i obliczany na żądanie. Warto zwrócić uwagę na to, w jaki sposób musimy przechwytywać operacje przypisania do zmiennej `remain` w deskrytorze i zgłaszać wyjątek. Jak dowiedzieliśmy się wcześniej, gdybyśmy tego nie zrobili, przypisanie do tego atrybutu instancji po cichu utworzyłoby atrybut instancji ukrywający deskryptor atrybutu klasy.

Na potrzeby porównania: kod oparty na deskrytorze składa się z 45 wierszy.

Powyższy kod uruchomiony wraz z opisanym wcześniej skryptem testującym zwraca bardzo podobne wyniki jak wcześniej. Inna jest tylko nazwa klasy w pierwszym wierszu:

```

$ python3 validate_tester.py validate_descriptors1
[Using: <class 'validate_descriptors1.CardHolder'>]
...takie same wyniki jak poprzednio dla właściwości...

```

## Opcja 2: sprawdzanie z wykorzystaniem indywidualnego stanu instancji (poprawnie)

W przykładzie 38.21, inaczej niż w poprzednim, opartym na właściwości, wartość atrybutu `name` jest przypisana do obiektu *deskryptora*, a nie do instancji klasy klienckiej. Choć wartość tę można zapisać zarówno w stanie instancji, jak i deskryptora, w tym drugim przypadku nie

trzeba przed nazwami umieszczać znaków podkreślenia, aby uniknąć konfliktów. W klasie `CardHolder` atrybut `name` jest zawsze obiektem deskryptora, a nie danych.

Powyższe podejście ma ten mankament, że stan zapisany w samym deskrytorze stanowi dane klasy, które są *współdzielone* przez wszystkie instancje klienckie, a więc nie mogą być różne w poszczególnych instancjach. Przechowywanie stanu w instancji *deskryptora* zamiast w instancji klasy właściciela (klienta) oznacza, że stan będzie taki sam we wszystkich instancjach klasy właściciela. Stan deskryptora może się zmieniać tylko w poszczególnych atrybutach.

Aby przekonać się, na czym to polega, z nowym skryptem z przykładu 38.22 spróbuj wyświetlić atrybuty instancji `bob` po utworzeniu drugiej instancji `anna`. Wartości zarządzanych atrybutów tej instancji (`name`, `age` i `acct`) *zastępują* atrybuty poprzedniej instancji o nazwie `bob`, ponieważ obie instancje współdzieli ten sam deskryptor przypisany ich klasie.

Przykład 38.22. *validate\_tester\_plus.py*

```
from validate_tester import loadclass
CardHolder = loadclass()

bob = CardHolder('1234-5678', 'Robert Zielony', 40, 'ul. Poziomkowa 15')
print('bob:', bob.name, bob.acct, bob.age, bob.addr)

anna = CardHolder('5678-12-34', 'Anna Czerwona', 35, 'ul. Poziomkowa 16')
print('anna:', anna.name, anna.acct, anna.age, anna.addr)  # Adres jest inny, dane klienckie
print('bob:', bob.name, bob.acct, bob.age, bob.addr)        # name, acct, age nadpisane?
```

Po uruchomieniu tego skryptu z deskrytorem stanu `CardHolder` z przykładu 38.21 wyniki potwierdzają podejrzenia: instancja `bob` przekształciła się w instancję `anna`!

```
$ python3 validate_tester2.py validate_descriptors1
[Using: <class 'validate_descriptors1.CardHolder'>]
bob: robert_zielony 12345*** 40 ul. Poziomkowa 15
anna: anna_czerwona 56781*** 35 ul. Poziomkowa 16
bob: anna_czerwona 56781*** 35 ul. Poziomkowa 16
```

To nie jest problem dla właściwości, ponieważ nie mają one własnego stanu, a istnieją uzasadnione zastosowania dla stanu deskryptora. Taki stan można wykorzystywać do zarządzania implementacją deskryptora i danymi wspólnymi dla wszystkich instancji. Powyższy kod został napisany w celu zilustrowania tej techniki. Tym mniej więcej różnią się implikacje wynikające z użycia stanu klasy i instancji.

Jednak w tej opcji atrybuty obiektu `CardHolder` prawdopodobnie lepiej jest przechowywać w danych *instancji*, a nie danych deskryptora. Stosowanie tej samej konwencji nazewnictwa `__x` pozwala uniknąć konfliktu nazw w instancji, co jest w tym przypadku ważniejsze, ponieważ klient jest osobną klasą zawierającą własne atrybuty stanu. W przykładzie 38.23 wprowadzono niezbędne zmiany w kodzie. Liczba wierszy jest taka sama (wciąż równa 45).

Przykład 38.23. *validate\_descriptors2.py*

```
class CardHolder:                                # Użycie stanu dla klienta-instancji
    acctlen = 8                                   # Dane klasy
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
```



```

self.acct = acct          # Dane instancji klienckiej
self.name = name          # Te także wywołują metodę __set__
self.age = age            # Zmiana nazwy __X nie jest konieczna: w deskrytorze
self.addr = addr          # Atrybut addr nie jest zarządzany
                           # Atrybut remain nie ma danych

class Name:
    def __get__(self, instance, owner):          # Zmienne klasy: lokalne dla CardHolder
        return instance.__name
    def __set__(self, instance, value):
        value = value.lower().replace(' ', '_')
        instance.__name = value
name = Name()                                     # class.name i zmieniony atrybut

class Age:
    def __get__(self, instance, owner):
        return instance.__age                    # Użycie danych *instancji*
    def __set__(self, instance, value):
        if value < 0 or value > 150:
            raise ValueError('niepoprawny wiek')
        else:
            instance.__age = value
age = Age()                                       # class.age i zmieniony atrybut

class Acct:
    def __get__(self, instance, owner):
        return instance.__acct[:-3] + '***'
    def __set__(self, instance, value):
        value = value.replace('-', '')
        if len(value) != instance.acctlen:       # Użycie danych instancji klasy
            raise TypeError('niepoprawny numer konta')
        else:
            instance.__acct = value
acct = Acct()                                    # class.acct i zmieniona nazwa

class Remain:
    def __get__(self, instance, owner):
        return instance.retireage - instance.age # Wywołuje Age.__get__
    def __set__(self, instance, value):
        raise TypeError('nie można ustawić remain') # Inaczej zezwolilibyśmy na ustawienie
remain = Remain()

```

Teraz zgodnie z oczekiwaniami do przechowywania atrybutów name, age i acct wykorzystywane są dane instancji (instancja bob nie ulega zmianom). Inne testy dają ten sam wynik:

```
$ python3 validate_tester2.py validate_descriptors2
```

```
[Using: <class 'validate_descriptors2.CardHolder'>]
```

```
bob: robert_zielony 12345*** 40 ul. Poziomkowa 15
```

```
anna: anna_czerwona 56781*** 35 ul. Poziomkowa 16
```

```
bob: robert_zielony 12345*** 40 ul. Poziomkowa 15
```

```
$ python3 validate_tester.py validate_descriptors2
```

```
...takie same wyniki jak poprzednio z wyjątkiem nazwy klasy...
```

Drobna uwaga: w tej wersji nie jest możliwy dostęp do deskrytora klasy, ponieważ w argumencie instancji jest umieszczana wartość None (należy również zwrócić uwagę, że nazwa atrybutu `__X` zmieniła się na `__Name__name`, co widać w komunikacie o błędzie pojawiającym się przy próbie odczytania wartości atrybutu):

```
>>> from validate_descriptors1 import CardHolder
>>> bob = CardHolder('1234-5678', 'Robert Zielony', 40, 'ul. Poziomkowa 15')
>>> bob.name
'robert_zielony'
>>> CardHolder.name
'robert_zielony'

>>> from validate_descriptors2 import CardHolder
>>> bob = CardHolder('1234-5678', 'Robert Zielony', 40, 'ul. Poziomkowa 15')
>>> bob.name
'robert_zielony'
>>> CardHolder.name
AttributeError: 'NoneType' object has no attribute '_Name__name'
```

Problem można rozwiązać, dopisując niewielki kod wyświetlający bardziej szczegółowy komunikat, ale raczej nie ma takiej potrzeby. Ponieważ program w tej wersji zapisuje dane w *instancji klientckiej*, deskryptory nie mają znaczenia, chyba że są powiązane z instancją klientką (podobnie jak zwykła metoda instancji). W rzeczywistości jest to zasadnicza zmiana wprowadzona w tej wersji kodu.

Ponieważ deskryptory są klasami, stanowią bardzo użyteczne narzędzie. Jednak oferowane przez nie funkcjonalności mogą mieć znaczny wpływ na działanie programu. Dlatego jak to zwykle bywa w programowaniu obiektowym, należy rozważnie wybierać sposoby przechowywania informacji o stanie.

## Wykorzystywanie metody `__getattr__` do sprawdzania poprawności

Jak widzieliśmy, metoda `__getattr__` przechwytuje wszystkie niezdefiniowane atrybuty, dzięki czemu może działać w sposób bardziej ogólny od właściwości czy deskryptorów. Na potrzeby przykładu sprawdzamy atrybut `name` w celu przekonania się, kiedy atrybut zarządzany jest pobierany. Pozostałe przechowywane są fizycznie w instancji, dzięki czemu nigdy nie docierają do metody `__getattr__`. Choć takie rozwiązanie jest bardziej uniwersalne od zastosowania właściwości czy deskryptorów, wymagana może być dodatkowa praca imitująca koncentrację innych narzędzi na określonych atrybutach. Musimy sprawdzać nazwy w czasie wykonywania (wybór wielokrotny, czyli główna rola funkcji `match`), a także napisać kod metody `__setattr__` w celu przechwycenia oraz sprawdzenia poprawności operacji przypisania do atrybutów.

Przykład 38.24 to wersja z `__getattr__` naszego kodu sprawdzającego. W rezultacie powyższa klasa, jak dwie poprzednie, zarządza atrybutami o nazwach `name`, `age` oraz `acct`. Pozwala także na bezpośredni dostęp do atrybutu `addr` i udostępnia atrybut tylko do odczytu o nazwie `rema` i, który jest w pełni wirtualny i obliczany na żądanie.

Przykład 38.24. *validate\_getattr.py*

```
class CardHolder:
    acctlen = 8                               # Dane klasy
    retireage = 62.5
```

```

def __init__(self, acct, name, age, addr):
    self.acct = acct          # Dane instancji
    self.name = name          # Te także wywołują metodę __setattr__
    self.age = age            # Zmiana nazwy __acct nie jest konieczna:
                                # nazwa jest sprawdzana
    self.addr = addr          # Atrybut addr nie jest zarządzany
                                # Atrybut remain nie ma danych

def __getattr__(self, name):
    match name:
        case 'acct':          # Przy pobraniu niezdefiniowanego atrybutu
            return self._acct[:-3] + '***'    # Atrybuty name, age, addr są zdefiniowane
        case 'remain':
            return self.retireage - self.age    # Nie wywołuje __getattr__
        else:
            raise AttributeError(name)

def __setattr__(self, name, value):
    match name:
        case 'name':          # Dla wszystkich przypisań do atrybutów
            value = value.lower().replace(' ', '_')    # Atrybut addr przechowywany
                                                        # w sposób bezpośredni
        case 'age':           # Zmiana nazwy acct na _acct
            if value < 0 or value > 150:
                raise ValueError('niepoprawny wiek')
        case 'acct':
            name = '_acct'
            value = value.replace('-', '')
            if len(value) != self.acctlen:
                raise TypeError('niepoprawny numer konta')
        case 'remain':
            raise TypeError('nie można ustawić remain')
    self.__dict__[name] = value    # Uniknięcie zapętlenia

```

Tak jak w wersjach tego przykładu z właściwością i deskryptorem, kluczowe jest zauważenie, że przypisania atrybutów wewnątrz metody konstruktora `__init__` wywołują także metodę `__setattr__` klasy. Kiedy metoda konstruktora przypisuje na przykład wartość do `self.name`, automatycznie wywołuje metodę `__setattr__`, przekształcającą wartość i przypisującą ją do atrybutu instancji o nazwie `name`. Dzięki przechowaniu `name` w instancji upewniamy się, że przyszłe próby dostępu nie wywołają metody `__getattr__`. Atrybut `acct` jest z kolei przechowywany jako `_acct`, tak by późniejsze próby dostępu do `acct` wywoływały metodę `__getattr__`.

Na potrzeby porównania: to rozwiązanie składa się z 34 wierszy kodu — 5 mniej od wersji opartej na właściwościach i 11 mniej od wersji wykorzystującej deskryptory (jednak tutaj zastąpiliśmy instrukcję `if` instrukcją `match` i tym samym dodaliśmy dwa wiersze i wcięcie). Jasność kodu ma oczywiście większe znaczenie od jego rozmiaru, jednak dodatkowy kod może czasami wiązać się z dodatkową pracą przy programowaniu oraz utrzymywaniu. Ważniejsze są tutaj chyba pełnione *role* — narzędzie uniwersalne, takie jak `__getattr__`, może się lepiej sprawdzać w ogólnej delegacji, natomiast właściwości i deskryptory są zaprojektowane z myślą o zarządzaniu określonymi atrybutami.

Warto również zauważyć, że poniższy kod powoduje *dotatkowe wywołania* przy ustawianiu atrybutów niezarządzanych (na przykład `addr`), natomiast żadne dodatkowe wywołania nie

występują dla pobierania atrybutów niezarządzanych, gdyż są one zdefiniowane. Choć dla większości programów będzie się to wiązało z niewielkim wzrostem nakładu pracy, właściwości i deskryptory powodują dodatkowe wywołanie jedynie przy dostępie do atrybutów zarządzanych. Ponadto znajdują się w wyniku zwracanym przez metodę `dir` wywoływaną przez podstawowe narzędzia.

Powyższy kod uruchomiony z dowolnym skryptem testowym zwraca takie same wyniki jak poprzednio (inna jest tylko nazwa klasy):

```
$ python3 validate_tester.py validate_getattr
... taki sam wynik jak w przykładzie z właściwościami, inna jest tylko nazwa klasy...
```

```
$ python3 validate_tester2.py validate_getattr
... taki sam wynik jak w przykładzie z deskryptorami, inna jest tylko nazwa klasy...
```

## Wykorzystywanie metody `__getattr__` do sprawdzania poprawności

Nasz ostatni wariant wykorzystuje przechwytyjącą wszystko metodę `__getattr__` w celu przechwycenia operacji pobierania atrybutów i zarządzania nimi zgodnie z potrzebami. Przechwycone zostaje każde pobranie atrybutu, dlatego musimy sprawdzać nazwy atrybutów w celu wykrycia tych zarządzanych i przekierować wszystkie pozostałe do klasy nadrzędnej, tak by pobranie zostało tam przetworzone w normalny sposób. Ta wersja wykorzystuje do przechwytywania operacji przypisania tę samą metodę `__setattr__` co wariant wcześniejszy (nie ma odpowiadającej metody `__setattribute__` w Pythonie, ale może to tylko kwestia czasu?).

Przykład 38.25 jest ostatnią wersją tego kodu. Działa bardzo podobnie do wersji z metodą `__getattr__`, dlatego nie będę tutaj powtarzał pełnego opisu. Warto jednak zauważyć, że ponieważ *każde* pobranie atrybutu przekierowywane jest do metody `__getattr__`, nie musimy zmieniać nazw w celu ich przechwycenia (atrybut `acct` przechowywany zostaje jako `acct`). Z drugiej strony, kod ten musi uwzględniać przekierowanie operacji pobrania niezarządzanych atrybutów do klasy nadrzędnej w celu uniknięcia zapętlenia lub dodatkowych wywołań.

Przykład 38.25. *validate\_getattribute.py*

```
class CardHolder:
    acctlen = 8                               # Dane klasy
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                       # Dane instancji
        self.name = name                       # Te także wywołują metodę __setattr__
        self.age = age                         # Zmiana nazwy __acct nie jest konieczna:
                                                # nazwa jest sprawdzana
        self.addr = addr                       # Atrybut addr nie jest zarządzany
                                                # Atrybut remain nie ma danych

    def __getattr__(self, name):
        superget = object.__getattr__(self, name)  # Bez pętli — jeden poziom w górę
        match name:
```

```

    case 'acct':
        # Dla wszystkich pobrań atrybutów
        return superget(self, 'acct')[:-3] + '***'
    case 'remain':
        return superget(self, 'retireage') - superget(self, 'age')
    case _:
        return superget(self, name)
        # name, age, addr: przechowane

def __setattr__(self, name, value):
    match name:
        case 'name':
            # Dla wszystkich przypisań atrybutów
            value = value.lower().replace(' ', '_')
            # Atrybut addr przechowany
            # w sposób bezpośredni

        case 'age':
            if value < 0 or value > 150:
                raise ValueError('niepoprawny wiek')
        case 'acct':
            value = value.replace('-', '')
            if len(value) != self.acctlen:
                raise TypeError('niepoprawny numer konta')
        case 'remain':
            raise TypeError('nie można ustawić remain')
    self.__dict__[name] = value
    # Unikanie pętli, oryginalne nazwy

```

Warto również zauważyć, że poniższa wersja powoduje wykonanie dodatkowych wywołań zarówno dla ustawiania, jak i pobierania atrybutów niezarządzanych (na przykład `addr`). Jeśli szybkość działania kodu ma duże znaczenie, to rozwiązanie może być najwolniejsze ze wszystkich. Na potrzeby porównania: poniższa wersja składa się z 34 wierszy kodu, tak samo jak poprzednia (znowu dwa dodatkowe wiersze z powodu instrukcji `match`).

Skrypty wykorzystujące metody `__getattr__` i `__getattribute__` z obydwoma skryptami testowymi działają tak samo jak wersje wykorzystujące właściwości i deskryptory. *Wszystkie cztery drogi prowadzą do tego samego celu*, mimo że skrypty różnią się strukturą, jak również nie-które ich role mogą być zbędne:

```
$ python3 validate_tester.py validate_getattribute
... taki sam wynik jak w przykładzie z właściwościami, inna jest tylko nazwa klasy...
```

```
$ python3 validate_tester_plus.py validate_getattribute
... taki sam wynik jak w przykładzie z właściwościami, inna jest tylko nazwa klasy...
```

By dowiedzieć się nieco więcej na temat technik zarządzania atrybutami, warto samodzielnie przestudiować i wykonać kody przykładów z tego rozdziału.

## Podsumowanie rozdziału

Niniejszy rozdział omawiał różne techniki zarządzania dostępem do atrybutów w Pythonie, w tym metody przeciążania operatorów `__getattr__` oraz `__getattribute__`, właściwości klas oraz deskryptory atrybutów. Porównaliśmy te narzędzia, przedstawiliśmy różnice między nimi oraz zaprezentowaliśmy kilka przypadków użycia demonstrujących ich działanie.

W rozdziale 39. będziemy kontynuowali omawianie elementów służących do budowy narzędzi, przyglądając się *dekoratorom* — kodowi wykonywanemu automatycznie w czasie tworzenia

funkcji oraz klas, a nie w momencie dostępu do atrybutów. Zanim jednak przejdziemy dalej, zapoznajmy się ze zbiorem pytań podsumowujących informacje zaprezentowane w tym rozdziale.

## Sprawdź swoją wiedzę — quiz

1. Czym różnią się od siebie metody `__getattr__` oraz `__getattribute__`?
2. Czym różnią się od siebie właściwości oraz deskryptory?
3. W jaki sposób są ze sobą powiązane właściwości oraz dekoratory?
4. Jakie są główne różnice w funkcjonalności metod `__getattr__` oraz `__getattribute__` w porównaniu z funkcjonalnością właściwości oraz deskryptorów?

## Sprawdź swoją wiedzę — odpowiedzi

1. Metoda `__getattr__` wykonywana jest jedynie dla operacji pobierania atrybutów *niezdefiniowanych* — to znaczy tych, które nie są obecne w instancji i nie są dziedziczone po żadnej z jej klas. Metoda `__getattribute__` jest z kolei wywoływana dla *każdej* operacji pobrania atrybutu, bez względu na to, czy atrybut ten jest zdefiniowany, czy też nie. Z tego powodu kod znajdujący się wewnątrz metody `__getattr__` może swobodnie pobierać inne atrybuty, jeśli są one zdefiniowane, natomiast `__getattribute__` musi do pobrania takich atrybutów wykorzystywać specjalny kod umożliwiający uniknięcie zapętlenia (musi przekierować operacje pobrania do klasy nadrzędnej, tak by pominąć samą siebie). Żadna z metod nie jest uruchamiana dla niejawnych pobrań wbudowanych operacji (bez heroicznych działań z następnego rozdziału).
2. Właściwości pełnią rolę szczegółową, natomiast deskryptory są bardziej ogólne. Właściwości definiują funkcje pobierania, ustawiania i usuwania dla określonego atrybutu. Deskryptory także udostępniają klasę z metodami przeznaczonymi dla tych operacji, jednak dają dodatkową elastyczność umożliwiającą obsługę bardziej dowolnych działań. Tak naprawdę właściwości są prostym sposobem tworzenia szczególnego rodzaju deskryptora — takiego, który wykonuje funkcje w momencie dostępu do atrybutu. Ich kod także się od siebie różni — właściwości tworzy się za pomocą funkcji wbudowanej, natomiast deskryptor za pomocą klasy. Tym samym deskryptory mogą korzystać ze wszystkich zwykłych opcji klas wynikających z programowania zorientowanego obiektowo, takich jak dziedziczenie. Co więcej, poza informacjami o stanie instancji deskryptory mają własny lokalny stan, dzięki czemu są w stanie unikać konfliktów między nazwami w instancji.
3. Właściwości można tworzyć za pomocą składni dekoratorów. Ponieważ funkcja wbudowana `property` przyjmuje pojedynczy argument funkcji, można ją wykorzystać bezpośrednio jako dekorator funkcji w celu zdefiniowania właściwości dostępu przez pobranie. Z uwagi na ponowne dowiązywanie nazw dekoratorów nazwa udekorowanej funkcji przypisywana jest do właściwości, której akcesor pobierania ustawiany jest na udekorowaną oryginalną funkcję (`name=property(name)`). Atrybuty `setter` i `deleter` właściwości pozwalają nam na dodanie akcesorów ustawiania oraz usuwania za pomocą składni dekoratorów — ustawiają one akcesor na udekorowaną funkcję i zwracają rozszerzoną właściwość.

4. Metody `__getattr__` i `__getattribute__` są bardziej ogólne — można je wykorzystać do przechwycenia dowolnie wielu atrybutów. Z kolei każda właściwość lub deskryptor umożliwia przechwytywanie dostępu jedynie dla *określonego* atrybutu — nie jesteśmy w stanie przechwycić operacji pobrania każdego atrybutu za pomocą pojedynczej właściwości czy deskryptora. Z drugiej strony, właściwości oraz deskryptory z założenia obsługują nie tylko pobieranie atrybutów, ale także *przypisanie*. Metody `__getattr__` i `__getattribute__` obsługują jedynie pobieranie. By przechwytywać także przypisania, niezbędne jest utworzenie metody `__setattr__`. Ich implementacja również się różni — `__getattr__` i `__getattribute__` są metodami przeciążania operatorów, podczas gdy właściwości i deskryptory są obiektami ręcznie przypisanymi do atrybutów klas. Właściwości i deskryptory pozwalają niekiedy uniknąć dodatkowych wywołań podczas przypisywania wartości niezarządzanym atrybutom, jak również są automatycznie umieszczane w wynikach metody `dir`. Jednak ich zakres działania jest węższy i nie można przy ich użyciu osiągnąć niektórych podstawowych celów. Nowe funkcjonalności pojawiające się w miarę rozwoju języka Python oferują nowe możliwości, jednak nie w pełni uwzględniają to, co było wcześniej.