

# Dekoratory

W rozdziale poświęconym zaawansowanym aspektom klas (rozdział 32.) poznaliśmy metody statyczne oraz metody klas, a także krótko przyjrzeliliśmy się składni dekoratora @, którą Python oferuje w celu zadeklarowania metod tego rodzaju. Z dekoratorami funkcji spotkaliśmy się z kolei w poprzednim rozdziale (rozdział 38.), przy okazji omawiania możliwości wykorzystania funkcji wbudowanej property w roli dekoratora, a także w rozdziale 29., przy omawianiu pojęcia abstrakcyjnych klas nadrzędnych oraz bardzo krótko w rozdziale 19.

Niniejszy rozdział stanowi kontynuację zagadnień, na których poprzednio zakończyliśmy omawianie dekoratorów. Tym razem zagłębimy się w mechanizmy wewnętrzne dekoratorów i omówimy bardziej zaawansowane sposoby tworzenia własnych, nowych dekoratorów. Jak zobaczymy, wiele z koncepcji omawianych we wcześniejszych rozdziałach, takich jak przechowywanie stanu, regularnie pojawia się w połączeniu z dekoratorami.

Zagadnienie to jest stosunkowo zaawansowane, a budowanie dekoratorów zazwyczaj jest bardziej interesujące dla twórców narzędzi niż dla programistów aplikacji. Mimo to, biorąc pod uwagę, że dekoratory są coraz częściej spotykane w popularnych platformach opartych na Pythonie, ich podstawowe zrozumienie może pomóc lepiej pojąć ich rolę, nawet z punktu widzenia użytkownika.

Poza omówieniem szczegółów konstruowania dekoratorów niniejszy rozdział pełni również rolę bardziej realistycznego *studium przypadku* działania Pythona. Ponieważ przykłady są tu nieco bardziej rozbudowane od innych, spotykanych dotychczas w książce, w lepszy sposób pozwalają zilustrować łączenie kodu w bardziej rozbudowane systemy oraz narzędzia. Dodatkową zaletą jest to, że większość kodu pisanego w tym rozdziale będzie można wykorzystać jako narzędzia ogólnego przeznaczenia w programach tworzonych na co dzień.

## Czym jest dekorator?

**Dekoracja** jest sposobem określania kodu zarządzającego dla funkcji oraz klas. Same dekoratory przybierają postać obiektów wywoływalnych (to znaczy funkcji), które przetwarzają inne obiekty wywoływalne. Jak widzieliśmy wcześniej w książce, dekoratory Pythona mają dwie, powiązane ze sobą odmiany:

- **Dekoratory funkcji**, dodane jako pierwsze, wykonują ponowne dowiązania nazw w momencie definicji funkcji, udostępniając warstwę logiki, która jest w stanie zarządzać funkcjami oraz metodami bądź ich późniejszymi wywołaniami.
- **Dekoratory klas**, dodane później, wykonują ponowne dowiązania nazw w momencie definicji klasy, udostępniając warstwę logiki, która jest w stanie zarządzać klasami bądź instancjami utworzonymi za pomocą ich późniejszego wywołania.

Mówiąc w skrócie, dekoratory udostępniają sposób wstawiania *automatycznie wykonywanego kodu* na końcu instrukcji definicji funkcji oraz klas — na końcu instrukcji `def` w przypadku dekoratorów funkcji oraz na końcu instrukcji `class` w przypadku dekoratorów klas. Taki kod może pełnić wiele różnych ról, opisanych w kolejnych podrozdziałach.

## Zarządzanie wywołaniami i instancjami

Przykładowo w typowym zastosowaniu taki wykonywany automatycznie kod można wykorzystać do rozszerzenia wywołań funkcji oraz klas. Dzieje się tak dzięki zainstalowaniu *obiektów opakowujących* (zwanych też *pośrednikami*), które zostaną wywołane później:

### Pośrednicy funkcji

Dekoratory funkcji instalują obiekty opakowujące, przechwytyjące późniejsze *wywołania funkcji* i odpowiednio je przetwarzające. Zazwyczaj wywołanie jest przekazywane do oryginalnej funkcji w celu wykonania zarządzanej operacji.

### Pośrednicy interfejsu

Dekoratory klas instalują obiekty opakowujące, przechwytyjące późniejsze *wywołania tworzące instancje* i odpowiednio je przetwarzające. Zazwyczaj wywołanie jest przekazywane do oryginalnej klasy w celu utworzenia zarządzanej instancji.

Dekoratory uzyskują taki efekt dzięki automatycznemu ponownemu dowiązaniu nazw funkcji oraz klas do innych obiektów wywoływalnych na końcu instrukcji `def` oraz `class`. W momencie późniejszego wywołania te obiekty mogą wykonywać zadania, takie jak śledzenie i pomiar czasu dla wywoływania funkcji czy zarządzanie dostępem do atrybutów instancji klas.

## Zarządzanie funkcjami i klasami

Choć większość przykładów w niniejszym rozdziale poświęcona jest obiektom opakującym przechwytyującym późniejsze wywołania funkcji oraz klas, nie jest to jedyna możliwość wykorzystania dekoratorów:

### Menedżery funkcji

Dekoratory funkcji można także wykorzystać do zarządzania *obiektami funkcji* zamiast ich późniejszymi wywołaniami — na przykład w celu zarejestrowania funkcji w API. W naszym przypadku nacisk położony zostanie na częściej wykorzystywane zastosowanie w opakowywaniu wywołań.

Dekoratory klas można również wykorzystać do bezpośredniego zarządzania *obiektami klas* zamiast wywołaniami tworzącymi instancje — na przykład w celu rozszerzenia klasy za pomocą nowych metod. Ponieważ ta rola w dużej mierze pokrywa się z zastosowaniem *metaklas* dodatkowe przypadki użycia omówimy jeszcze w kolejnym rozdziale. Jak się przekonamy, oba narzędzia są uruchamiane na koniec procesu tworzenia klasy, jednak dekorator często jest rozwiązaniem mniej obciążającym system.

Innymi słowy, dekoratory funkcji można wykorzystać do zarządzania zarówno wywołaniami funkcji, jak i obiektami funkcji, natomiast dekoratory klas — do zarządzania zarówno instancjami klas, jak i samymi klasami. Dzięki zwracaniu samego udekorowanego obiektu zamiast obiektu opakowującego dekoratory stają się prostym krokiem wykonywanym po utworzeniu funkcji oraz klas.

Bez względu na pełnioną rolę dekoratory udostępniają wygodny i jawny sposób tworzenia narzędzi przydatnych zarówno w trakcie tworzenia programu, jak i w działających systemach produkcyjnych.

## Wykorzystywanie i definiowanie dekoratorów

W zależności od wykonywanych przez nas zadań z dekoratorami możemy spotkać się jako ich użytkownik bądź jako osoba je udostępniająca. Jak widzieliśmy, sam Python zawiera wbudowane dekoratory o wyspecjalizowanych rolach — deklaracje metod statycznych, tworzenie właściwości i wiele innych. Dodatkowo wiele popularnych zestawów narzędzi Pythona zawiera dekoratory wykonujące takie zadania, jak zarządzanie bazą danych czy logiką interfejsu użytkownika. W takich przypadkach możemy sobie poradzić bez wiedzy o tym, w jaki sposób tworzy się kod dekoratora.

W przypadku zadań bardziej ogólnych programiści mogą samodzielnie pisać dowolny kod własnych dekoratorów. Przykładowo dekoratory funkcji można wykorzystać do rozszerzania funkcji za pomocą kodu dodającego śledzenie wywołań, testującego poprawność argumentów w trakcie debugowania, automatycznie tworzącego i zwalnającego blokady wątków czy mierzącego czas wywołań funkcji w celu optymalizacji. Wszystkie działania, jakich dodanie do wywołania funkcji możemy sobie wyobrazić, są kandydatami na własne dekoratory funkcji.

Z drugiej strony, dekoratory funkcji zaprojektowano w taki sposób, by rozszerzały one jedynie określone *wywołanie* funkcji bądź metody, a nie cały *interfejs obiektu*. Lepiej spełniają tę rolę dekoratory klas — ponieważ mogą one przechwytywać wywołania tworzące instancje, można je wykorzystać do implementowania wszelkich zadań związanych z rozszerzaniem interfejsu obiektów czy zarządzaniem. Przykładowo własne dekoratory klas mogą śledzić lub sprawdzać wszystkie referencje do atrybutów wykonywane dla obiektu. Można je także wykorzystywać do tworzenia obiektów pośredniczących (proxy), klas singletona, a także innych popularnych wzorców projektowych. Tak naprawdę niedługo zobaczymy, że wiele z dekoratorów klas jest bardzo podobnych do wzorca projektowego *delegacji*, z którym spotkaliśmy się w rozdziale 31.

## Do czego służą dekoratory?

Jak większość zaawansowanych narzędzi Pythona, dekoratory nigdy nie są wymagane i niezbędne — ich funkcjonalność można często zaimplementować za pomocą prostych wywołań funkcji pomocniczych lub innych technik (a na podstawowym poziomie możemy zawsze napisać kod dowiązujący ponownie zmienne w ręczny sposób; dekoratory robią to w sposób automatyczny).

Dekoratory udostępniają jawną składnię przeznaczoną do zadań tego typu, dzięki czemu intencje programisty są jasne, możliwe jest ograniczenie powtarzalności kodu, a także zapewnienie poprawnego użycia API.

- Dekoratory mają bardzo czytliwą, jawną składnię, co ułatwia ich dostrzeżenie w porównaniu z wywołaniami funkcji pomocniczych, które mogą być dowolnie oddalone od podmiotowych funkcji bądź klas.
- Dekoratory stosowane są *raz*, przy definicji podmiotowej funkcji bądź klasy. Nie jest konieczne dodawanie dodatkowego kodu (który być może w przyszłości będzie musiał się zmienić) do każdego wywołania klasy bądź funkcji.
- Z uwagi na dwa wcześniejsze punkty dekoratory sprawiają, że mniej prawdopodobne jest to, iż użytkownik API *zapomni* rozszerzyć funkcję bądź klasę zgodnie z wymaganiami API.

### Dekoratory a makra

Stosowanie dekoratorów można porównać do tzw. **programowania aspektowego** typowego dla innych języków, polegającego na tworzeniu kodu automatycznie uruchamianego przed wywołaniem funkcji lub po nim. Składnia dekoratorów jest bardzo podobna do anotacji w Javie (prawdopodobnie została stamtąd zapożyczona), niemniej model w Pythonie jest uważany za bardziej elastyczny i uniwersalny.

Niektórzy programiści porównują dekoratory do *makr*, choć jest to dość nieścisłe, a nawet mylące. Makra (na przykład dyrektywa `#define` w preprocesorze kodu w języku C) służą do generowania kodu, tj. zastępowania tekstu kodu i rozszerzania jego fragmentów. Natomiast dekoratory w Pythonie wykonują *w trakcie działania* programu określone operacje, wykorzystując dowiązywane nazwy, obiekty wywoływalne i opakowujące. Choć oba pojęcia mają wspólne obszary zastosowań, zasadniczo się od siebie różnią zasięgiem działania, implementacją i sposobem kodowania. Porównywanie dekoratora i makra jest równie niewłaściwe, jak instrukcji `import` w Pythonie i dyrektywy `#include` w języku C, dlatego że pierwsze pojęcie oznacza instrukcję wykonywaną w trakcie działania programu, a drugie operację polegającą na wstawieniu tekstu do kodu źródłowego.

Oczywiście znaczenie terminu *makro* bardzo się rozmyło w miarę upływu czasu. Dla niektórych programistów oznacza ono nawet wydzieloną serię operacji wykonywanych przez procedurę, a użytkownicy innych języków widzą w nim analogię do deskryptora. Należy jednak pamiętać, że dekoratory są to *obiekty* wykonywalne, które zarządzają innymi *obiektami* wykonywalnymi, ale nie rozszerzają ich. Pythona należy rozumieć i używać w kategoriach jego idiomów.

Innymi słowy, poza samym modelem funkcjonalnym dekoratory mają pewne zalety z punktu widzenia utrzymywania kodu oraz estetyki. Co więcej, jako narzędzia strukturyzujące kod w naturalny sposób powodują one jego *hermetyzację*, co ogranicza powtarzalność i ułatwia wszelkie późniejsze zmiany.

Dekoratory mają jednak również pewne potencjalne *wady* — kiedy wstawiają logikę opakowującą, mogą zmieniać typy dekorowanych obiektów oraz powodować dodatkowe wywołania. Z drugiej strony, te same zastrzeżenia można mieć do wszystkich technik dodających do obiektów logikę opakowującą.

W niniejszym rozdziale omówimy wszystkie te ograniczenia w kontekście prawdziwego kodu. Choć wybór dekoratorów jest nadal kwestią w dużej mierze subiektywną, ich zalety są na tyle kuszące, że w świecie Pythona stają się one dobrą praktyką. By pomóc podjąć decyzję w tej sprawie, przejdziemy teraz do szczegółów.

## Podstawy

Zacznijmy od pierwszego przyjrzenia się działaniu dekoratorów z nieco symbolicznej perspektywy. Niedługo będziemy także pisać prawdziwy, działający kod, jednak ponieważ magia dekoratorów sprowadza się do operacji automatycznego ponownego dowiązywania, istotne jest, by najpierw zrozumieć ten typ odwzorowania.

## Dekoratory funkcji

Jak widzieliśmy wcześniej w książce, są one w dużej mierze składniowym elementem wykonującym jedną funkcję w drugiej pod koniec instrukcji `def` i dowiązującym oryginalną nazwę funkcji do wyniku.

## Zastosowanie

Dekorator funkcji jest rodzajem *deklaracji w czasie wykonywania* i dotyczy funkcji, której definicja pojawi się później. Dekorator zapisywany jest w kodzie w wierszu tuż przed instrukcją `def` definiującą funkcję bądź metodę i składa się z symbolu `@`, po którym następuje referencja do **metafunkcji** — funkcji (bądź innego obiektu wywoływalnego) zarządzającej inną funkcją. Od Pythona 3.9 kod po `@` może być dowolnym *wyrażeniem* zwracającym metefunkcję, ale zazwyczaj jest to prosta nazwa.

Z punktu widzenia kodu dekoratory funkcji automatycznie odwzorowują poniższą składnię:

```
@decorator                                # Udekorowanie funkcji
def F(arg):
    ...

F(99)                                     # Wywołanie funkcji
```

na jej poniższy odpowiednik, w którym dekorator jest jednoargumentowym obiektem wywoływalnym zwracającym obiekt wywoływalny o tej samej liczbie argumentów co `F`, jeśli nie samą funkcję `F`:

```
def F(arg):
    ...
    F = decorator(F)                # Ponowne dowiązanie nazwy funkcji do wyniku dekoratora

F(99)                               # Wywołuje decorator(F)(99)
```

Takie automatyczne ponowne dowiązywanie nazw działa dla dowolnej instrukcji `def` — bez względu na to, czy jest to prosta *funkcja*, czy też *metoda* wewnątrz klasy. Kiedy później wywołana jest funkcja `F`, tak naprawdę wywoływany jest obiekt *zwracany* przez dekorator — może to być albo inny obiekt implementujący wymaganą logikę opakowującą, albo oryginalna funkcja.

Innymi słowy, dekoracja odwzorowuje pierwszy z poniższych zapisów na drugi (choć dekorator wykonywany jest tak naprawdę raz, w czasie dekoracji):

```
func(6, 7)
decorator(func)(6, 7)
```

Takie automatyczne ponowne dowiązywanie nazw odpowiada składni metod statycznych oraz dekoracji właściwości, z którymi spotkaliśmy się wcześniej w książce:

```
class C:
    @staticmethod
    def meth(...): ...                # meth = staticmethod(meth)

class C:
    @property
    def name(self): ...              # name = property(name)
```

W obu przypadkach nazwa metody jest na końcu instrukcji `def` ponownie dowiązywana do wyniku wbudowanego dekoratora funkcji. Późniejsze wywołanie oryginalnej nazwy wywołuje obiekt zwrócony przez dekorację. W tych konkretnych przypadkach oryginalne nazwy są ponownie dowiązywane do routera statycznej metody i deskryptora właściwości, ale sam proces jest bardziej ogólny, o czym mowa w następnym podrozdziale.

## Implementacja

Sam dekorator jest *obiektem wywoływalnym zwracającym obiekt wywoływalny*. Oznacza to, że zwraca on obiekt, który będzie wywołany później, kiedy udekorowana funkcja wywoływana jest za pomocą oryginalnej nazwy. Zwracany obiekt jest albo obiektem opakowującym przechwytyjącym późniejsze wywołania, albo w jakiś sposób rozszerzoną oryginalną funkcją. Tak naprawdę dekoratory mogą być dowolnymi typami obiektów wywoływalnych i mogą *zwracać* dowolny typ obiektu wywoływalnego — można wykorzystać dowolną kombinację funkcji oraz klas, choć niektóre z nich lepiej sprawdzają się w pewnych kontekstach.

Przykładowo w celu wejścia do protokołu dekoracji i zarządzania funkcją tuż po jej utworzeniu możemy napisać kod dekoratora o poniższej formie:

```
def decorator(F):
    # Przetworzenie funkcji F
    return F

@decorator
def func(): ...                    # func = decorator(func)
```

Ponieważ oryginalna udekorowana funkcja przypisywana jest z powrotem do swojej nazwy, powyższy kod dodaje po prostu krok po utworzeniu do definicji funkcji. Taką strukturę można na przykład wykorzystać do zarejestrowania funkcji w API czy przypisania atrybutów funkcji.

W bardziej typowym zastosowaniu w celu wstawienia logiki przechwytyjącej późniejsze wywołania funkcji możemy napisać kod dekoratora zwracającego obiekt inny od oryginalnej funkcji — *pośrednika* dla późniejszych wywołań:

```
def decorator(F):
    # Zapisanie lub użycie funkcji F
    # Zwrócenie innego obiektu wywoływalnego: zagnieżdżonej instrukcji def, klasy z __call__ i tak dalej

@decorator
def func(): ...                                # func = decorator(func)
```

Powyższy dekorator wywoływany jest w czasie dekoracji, a zwracany przez niego obiekt wywoływany jest przy późniejszym wywołaniu oryginalnej nazwy funkcji. Sam dekorator otrzymuje udekorowaną funkcję — zwracany obiekt wywoływalny otrzymuje wszelkie argumenty przekazane później do nazwy udekorowanej funkcji. Tak samo działa to w przypadku *metod* klas — domniemany obiekt instancji po prostu pokazuje się w pierwszym argumencie zwracanego obiektu wywoływalnego.

Oto popularny wzorzec kodu ujmujący tę kwestię z punktu widzenia szkieletu. Dekorator zwraca obiekt opakowujący, zachowujący oryginalną funkcję w zakresie zawierającym:

```
def decorator(F):                                # W momencie dekoracji @
    def wrapper(*args):                          # W momencie wywołania opakowanej funkcji
        # Użycie F oraz args
        # F(*args) wywołuje oryginalną funkcję
        return wrapper

@decorator
def func(x, y):
    ...

func(6, 7)                                     # 6, 7 przekazywane do *args obiektu opakowującego
```

Kiedy później wywołana zostaje nazwa `func`, tak naprawdę wywoływana jest funkcja opakowująca `wrapper` zwracana przez dekorator `decorator`. Funkcja `wrapper` może wtedy wykonać oryginalną funkcję `func`, ponieważ jest ona nadal dostępna w *zakresie zawierającym*. W kodzie tego typu każda udekorowana funkcja tworzy nowy zakres zachowujący stan.

By uzyskać to samo z użyciem *klas*, możemy przeciążyć operację wywoływania i skorzystać z atrybutów instancji w miejsce zakresów zawierających:

```
class decorator:
    def __init__(self, func):                    # W momencie dekoracji @
        self.func = func
    def __call__(self, *args):                  # W momencie wywołania opakowanej funkcji
        # Użycie self.func oraz args
        # self.func(*args) wywołuje oryginalną funkcję

@decorator
```

```
def func(x, y):
    ...
    func(6, 7)
```

```
# func = decorator(func)
# func przekazywane jest do __init__
# 6, 7 przekazywane do *args dla __call__
```

Gdy w tym przypadku wywołamy później nazwę `func`, tak naprawdę wywołamy metodę przeciążania operatora `__call__` instancji utworzonej za pomocą dekoratora `decorator`. Metoda `__call__` może następnie wywołać oryginalną funkcję `func`, ponieważ jest ona nadal dostępna w atrybucie instancji. W kodzie tego typu każda udekorowana funkcja tworzy nową instancję zachowującą stan.

## Obsługa dekoracji metod

Jedną istotną kwestią dotyczącą powyższego kodu opartego na klasie jest to, że choć działa on dla przechwytywania prostych wywołań *funkcji*, nie do końca tak jest w przypadku zastosowania do funkcji *metod* klas:

```
class decorator:
    def __init__(self, func):
        self.func = func
    def __call__(self, *args):
        # self.func(*args) nie działa!

class C:
    @decorator
    def method(self, x, y):
        ...
```

```
# func to metoda bez instancji
# self jest instancją dekoratora
# Instancja C nie znajduje się w args!
# method = decorator(method)
# Ponowne dowiązanie do instancji dekoratora
```

W przypadku powyższego kodu udekorowana metoda zostaje ponownie dowiązana do *instancji* klasy dekoratora zamiast do prostej funkcji.

Problem polega na tym, że `self` w metodzie `__call__` dekoratora otrzymuje przy późniejszym wywołaniu metody instancję dekoratora klasy, natomiast instancja klasy `C` nigdy nie znajduje się w `*args`. Sprawia to, iż niemożliwe staje się przekazanie wywołania do oryginalnej metody — obiekt dekoratora zachowuje oryginalną funkcję metody, jednak nie ma instancji, którą może do niej przekazać.

By obsłużyć zarówno funkcje, jak i metody, lepiej sprawdzi się alternatywa z funkcją zagnieżdżoną:

```
def decorator(F):
    def wrapper(*args):
        # F(*args) wykonuje funkcję lub metodę
        return wrapper
    @decorator
    def func(x, y):
        ...
        func(6, 7)

class C:
    @decorator
    def method(self, x, y):
        ...

X = C()
X.method(6, 7)
```

```
# F jest funkcją lub metodą bez instancji
# instancja klasy w args[0] w przypadku metody
# func = decorator(func)
# Tak naprawdę wywołuje wrapper(6, 7)
# method = decorator(method)
# Ponowne dowiązanie do prostej funkcji
# Tak naprawdę wywołuje wrapper(X, 6, 7)
```



W przypadku takiego kodu obiekt opakowujący wrapper otrzymuje instancję klasy `C` w pierwszym argumencie, dzięki czemu może przekazać go do oryginalnej metody i uzyskać dostęp do informacji o stanie.

Powyższa alternatywa z funkcją zagnieżdżoną działa, ponieważ Python tworzy *dowiązany obiekt metody* i tym samym przekazuje instancję podmiotowej klasy do argumentu `self` jedynie wtedy, gdy atrybut metody odwołuje się do prostej funkcji. Gdy odwołuje się on natomiast do instancji klasy wywoływalnej, instancja tej klasy przekazywana jest do `self` w celu udostępnienia klasie wywoływalnej jej własnych informacji o stanie. W dalszej części rozdziału zobaczymy, jakie znaczenie ma ta subtelna różnica w praktyce.

Warto również zauważyć, że funkcje zagnieżdżone są chyba najprostszym sposobem obsługi dekoracji zarówno dla funkcji, jak i metod — choć nie są jedynym dostępnym rozwiązaniem. *Deskryptory* z poprzedniego rozdziału otrzymują na przykład przy wywołaniu zarówno deskryptor, jak i instancję podmiotowej klasy. Choć jest to bardziej skomplikowane, w dalszej części rozdziału zobaczymy, w jaki sposób możemy wykorzystać to narzędzie także w tym kontekście.

## Dekoratory klas

Dekoratory funkcji okazały się tak przydatne, że model ten został rozszerzony, tak by pozwolić na dekorację klas. Początkowo były przyjmowane z oporami, ponieważ ich role częściowo pełniły *metaklasy*. Ostatecznie jednak dekoratory klas zostały zaakceptowane, ponieważ dzięki nim ten sam efekt można często osiągnąć w znacznie prostszy sposób.

Dekoratory klas są silnie powiązane z dekoratorami funkcji. Tak naprawdę wykorzystują one tę samą składnię i podobne wzorce kodu. Zamiast opakowywać poszczególne funkcje bądź metody, dekoratory klas są sposobami zarządzania klasami lub opakowywania wywołań tworzących instancje w dodatkową logikę, która zarządza instancjami utworzonymi z klasy bądź je rozszerza. W tym drugim przypadku zarządzają wszystkimi *interfejsami* obiektu.

## Zastosowanie

Z punktu widzenia składni dekoratory klas pojawiają się tuż przed instrukcjami `class` (w podobny sposób jak dekoratory funkcji pojawiają się tuż przed definicjami funkcji). Zakładając, że dekorator jest funkcją jednoargumentową zwracającą obiekt wywoływalny, poniższa składnia dekoratora klasy:

```
@decorator                                # Udekorowanie klasy
class C:
    ...

x = C(99)                                  # Utworzenie instancji
```

jest odpowiednikiem poniższego kodu. Klasa jest automatycznie przekazywana do funkcji dekoratora, natomiast wynik dekoratora przypisywany jest z powrotem do nazwy klasy:

```
class C:
    ...
C = decorator(C)                            # Ponowne dowiązanie nazwy klasy do wyniku dekoratora
x = C(99)                                    # Tak naprawdę wywołuje decorator(C)(99)
```

W rezultacie późniejsze wywołanie nazwy klasy w celu utworzenia instancji wywołuje obiekt zwrócony przez dekorator, a nie oryginalną klasę.

## Implementacja

Nowe dekoratory klas tworzy się za pomocą tych samych technik co w przypadku dekoratorów funkcji, choć czasami modyfikacje odbywają się na dwóch poziomach: konstruktorów klasy i interfejsu dostępu do instancji. Ponieważ dekorator klasy jest także *obiektem wywoływalnym zwracającym obiekt wywoływalny*, większość kombinacji funkcji i klas będzie w zupełności wystarczająca.

Bez względu na sposób zapisu w kodzie wynikiem dekoratora jest to, co zostaje wykonane przy późniejszym utworzeniu instancji. Przykładowo by po prostu zarządzać klasą tuż po jej utworzeniu, należy zwrócić oryginalną klasę:

```
def decorator(C):  
    # Przetworzenie klasy C  
    return C  
  
@decorator  
class C: ...                # C = decorator(C)
```

By zamiast tego wstawić warstwę opakowującą przechwytyującą późniejsze wywołania tworzące instancję, należy zwrócić inny obiekt wywoływalny:

```
def decorator(C):  
    # Zapisanie lub użycie klasy C  
    # Zwrócenie innego obiektu wywoływalnego: zagnieżdżonej instrukcji def, klasy z __call__ i tak dalej  
  
@decorator  
class C: ...                # C = decorator(C)
```

Obiekt wywoływalny zwracany przez tego typu dekorator klasy zazwyczaj tworzy i zwraca nową instancję oryginalnej klasy, rozszerzoną w jakiś sposób umożliwiającą zarządzanie jej interfejsem. Przykładowo poniższy kod wstawia obiekt przechwytyjący niezdefiniowane atrybuty instancji klasy:

```
def decorator(cls):  
    class Wrapper:  
        def __init__(self, *args):  
            self.wrapped = cls(*args)  
        def __getattr__(self, name):  
            return getattr(self.wrapped, name)  
    return Wrapper  
  
@decorator  
class C:  
    def __init__(self, x, y):  
        self.attr = 'hakować'  
  
x = C(6, 7)  
print(x.attr)
```

# W momencie dekoracji @  
# W momencie tworzenia instancji  
# W momencie pobrania atrybutu  
# C = decorator(C)  
# Wykonywane przez Wrapper.\_\_init\_\_  
# Tak naprawdę wywołuje Wrapper(6, 7)  
# Wykonuje Wrapper.\_\_getattr\_\_, wyświetla "hakować"

W powyższym przykładzie dekorator dowiązuje ponownie nazwę klasy do innej klasy, zachowując oryginalną klasę w zakresie zawierającym i tworząc oraz osadzając instancję oryginalnej klasy przy wywołaniu. Kiedy atrybut jest później pobierany z instancji, jest on przechwytywany przez metodę `__getattr__` obiektu opakowującego i delegowany do osadzonej instancji oryginalnej klasy. Co więcej, każda udekorowana klasa tworzy nowy zakres pamiętający oryginalną klasę. W dalszej części rozdziału przekształcimy ten przykład w bardziej przydatny kod.

Podobnie jak dekoratory funkcji, dekoratory klas są często zapisywane w kodzie jako funkcje *domykające* (zwane fabrycznymi), tworzące i zwracające obiekty wywoływalne, lub jako klasy wykorzystujące metody `__init__` albo `__call__` do przechwytywania operacji wywoływania — bądź dowolna kombinacja obu rozwiązań. Funkcje fabryczne zazwyczaj zachowują stan w referencjach do zakresu zawierającego, natomiast klasy — w atrybutach.

## Obsługa większej liczby instancji

Tak jak w przypadku dekoratorów funkcji, w dekoratorach klas pewne typy kombinacji działają lepiej od innych. Rozważmy poniższą *niepoprawną* alternatywę dla dekoratora klasy z poprzedniego przykładu:

```
class Decorator:
    def __init__(self, C):
        self.C = C
    def __call__(self, *args):
        self.wrapped = self.C(*args)
        return self
    def __getattr__(self, attrname):
        return getattr(self.wrapped, attrname)

@Decorator
class C: ...

x = C()
y = C()
```

# W momencie dekoracji @  
# W momencie tworzenia instancji  
# W momencie pobrania atrybutu  
# C = Decorator(C)  
# Nadpisuje x!

Powyższy kod obsługuje kilka udekorowanych klas (każda z nich tworzy nową instancję klasy `Decorator`) i przechwytuje wywołania tworzące instancje (każda wykonuje metodę `__call__`). W przeciwieństwie do poprzedniej wersji ta nie jest w stanie obsłużyć *większej liczby instancji* danej klasy — każde wywołanie tworzące instancję nadpisuje poprzednio zapisaną instancję. Wersja oryginalna obsługuje większą liczbę instancji, ponieważ każde wywołanie tworzące instancję tworzy nowy, niezależny obiekt opakowujący. Mówiąc bardziej ogólnie, każdy z poniższych wzorców obsługuje większą liczbę opakowanych instancji:

```
def decorator(C):
    class Wrapper:
        def __init__(self, *args):
            self.wrapped = C(*args)
        return Wrapper

class Wrapper: ...
def decorator(C):
```

# W momencie dekoracji @  
# W momencie tworzenia instancji  
# W momencie dekoracji @

```
def onCall(*args):
    return Wrapper(C(*args))
return onCall
```

# W momencie tworzenia instancji  
# Osadzenie instancji w instancji

Zjawisko to omówimy w bardziej realistycznym kontekście w dalszej części rozdziału. W praktyce musimy uważać na poprawne łączenie typów wywoływalnych, tak by obsługiwały one wyznaczone przez nas zadania, oraz mądrze ustalać zasady dotyczące stanów.

## Zagnieżdżanie dekoratorów

Czasami jeden dekorator nie wystarczy. Załóżmy na przykład, że przygotowaliśmy *dwa* dekoratory, które będą nam potrzebne podczas kodowania: jeden do sprawdzania typów argumentów przed wywołaniem funkcji, a drugi do sprawdzania typu zwracanego wyniku po wywołaniu funkcji. Każdego z dekoratorów można używać osobno, ale co robić, gdy *oba* trzeba przypisać do tej samej funkcji? Otóż należy je *zagnieżdżyć* tak, aby wynik pierwszego dekoratora był modyfikowany przez drugi. Kolejność zagnieżdżenia nie jest istotna, o ile tylko oba dekoratory są wykonywane podczas późniejszych wywołań.

By obsłużyć kilka kroków rozszerzenia, składnia dekoratorów pozwala na dodawanie większej liczby warstw logiki opakowującej do udekorowanej funkcji lub metody. Przy skorzystaniu z tej możliwości każdy dekorator musi się pojawiać w osobnym wierszu. Następująca postać składni dekoratorów:

```
@A
@B
@C
def f(...):
    ...
```

wykonuje to samo co poniższy kod:

```
def f(...):
    ...
    f = A(B(C(f)))
```

W kodzie oryginalna funkcja przekazywana jest przez trzy różne dekoratory, a wynikowy obiekt wywoływalny przypisywany jest z powrotem do oryginalnej nazwy. Każdy dekorator przetwarza wynik poprzedniego, którym może być oryginalna funkcja lub wstawiony obiekt opakowujący.

Jeśli każdy z dekoratorów wstawia obiekty opakowujące, rezultat będzie taki, że przy wywołaniu nazwy oryginalnej funkcji wywołane zostaną trzy różne warstwy logiki obiektów opakowujących, pozwalające na rozszerzenie oryginalnej funkcji na trzy różne sposoby. Ostatni wymieniony dekorator zostanie zastosowany jako pierwszy — jako zagnieżdżony najgłębiej, kiedy później jest wywoływana pierwotna funkcja.

Tak samo jak w przypadku funkcji, większa liczba dekoratorów klas daje większą liczbę wywołań zagnieżdżonych funkcji i być może większą liczbę poziomów logiki opakowującej wokół wywołań tworzących instancje. Przykładowo poniższy kod:

```
@hack
@code
```

```
class C:
    ...

X = C()
```

jest odpowiednikiem następującego:

```
class C:
    ...
C = hack(code(C))

X = C()
```

I znów, każdy z dekoratorów może zwracać albo oryginalną klasę, albo wstawiony obiekt opakowujący. W przypadku obiektów opakowujących, gdy zostanie zażądana instancja oryginalnej klasy C, wywołanie przekierowywane jest do obiektów warstw opakowujących udostępnianych przez dekoratory spam oraz eggs, które mogą pełnić zupełnie różne role. Na przykład mogą rejestrować działanie funkcji i kontrolować dostęp do atrybutów, przy czym oba kroki będą wykonywane przy następnych żądaniach.

Przykładowo poniższe nic nierobiące dekoratory po prostu zwracają udekorowaną funkcję:

```
def d1(F): return F
def d2(F): return F
def d3(F): return F

@d1
@d2
@d3
def func():
    print('hakować')

func() # func = d1(d2(d3(func)))
```

# Wyświetla "hakować"

Ta sama składnia działa w przypadku klas, podobnie jak te same nic nierobiące dekoratory.

Kiedy dekoratory wstawiają obiekty funkcji opakowujących, mogą one jednak rozszerzać oryginalne funkcje przy wywołaniu. Poniższy kod dokonuje konkatencji wyniku w warstwach dekoratorów, w miarę przechodzenia warstw od najbardziej wewnętrznej do zewnętrznej:

```
def d1(F): return lambda: 'X' + F()
def d2(F): return lambda: 'Y' + F()
def d3(F): return lambda: 'Z' + F()

@d1
@d2
@d3
def func():
    return 'hakować'

print(func()) # func = d1(d2(d3(func)))
```

# Wyświetla "XYZhakować"

Funkcje lambda wykorzystane zostają tutaj do zaimplementowania warstw opakowujących (każda zachowuje opakowaną funkcję w zakresie zawierającym). W praktyce warstwy opakowujące mogą przybierać postać na przykład funkcji czy klas wywoływalnych. Przy dobrym zaprojektowaniu zagnieżdżanie pozwala łączyć ze sobą poszczególne etapy rozszerzania na wiele sposobów.

## Argumenty dekoratorów

Dekoratory zarówno funkcji, jak i klas zdają się móc przyjmować *argumenty*, choć tak naprawdę argumenty te są przekazywane do obiektu wywoływalnego *zwracającego* z kolei dekorator, który natomiast *zwraca* obiekt wywoływalny. Zazwyczaj w takim przypadku definiowanych jest kilka poziomów zachowywania stanu. Poniższy kod:

```
@decorator(A, B)
def F(arg):
    ...
```

```
F(99)
```

automatycznie odwzorowywany jest na następującą postać równoważną, gdzie decorator jest obiektem wywoływalnym *zwracającym* sam dekorator. Zwrócony dekorator zwraca z kolei obiekt wywoływalny wykonywany później dla wywołań nazwy oryginalnej funkcji:

```
def F(arg):
    ...
F = decorator(A, B)(F)      # Ponowne dowiązanie F do wyniku wartości zwracanej przez dekorator

F(99)                      # Tak naprawdę wywołuje decorator(A, B)(F)(99)
```

Argumenty dekoratorów są ustalane *przed* wystąpieniem dekoracji i zazwyczaj wykorzystywane są do przechowania informacji o stanie do użycia w późniejszych wywołaniach. Funkcja dekoratora z poniższego przykładu może na przykład przybrać następującą postać:

```
def decorator(A, B):
    # Zapisanie lub użycie A, B
    def actualDecorator(F):
        # Zapisanie lub użycie funkcji F
        # Zwrócenie obiektu wywoływalnego: zagnieżdżonej instrukcji def, klasy z __call__ i tak dalej
        return callable
    return actualDecorator
```

W tej strukturze funkcja zewnętrzna zapisuje argumenty dekoratora w postaci informacji o stanie, by były one dostępne do użycia w samym dekoratorze, zwracanym przez niego obiekcie wywoływalnym lub w obu tych miejscach. Powyższy fragment kodu zachowuje argument informacji o stanie w referencjach do zakresu funkcji zawierającej, jednak często wykorzystywane są również atrybuty klas.

Innymi słowy, argumenty dekoratora często wymuszają *trzy poziomych obiektów wywoływalnych*: obiekt przyjmujący argumenty dekoratora zwracający obiekt służący jako dekorator, który z kolei zwraca obiekt obsługujący wywołania do oryginalnej funkcji bądź klasy. Każdy z tych trzech poziomów może być funkcją lub klasą i może zachowywać stan w postaci zakresów lub atrybutów klas.

Argumenty dekoratorów można wykorzystywać do przekazywania wartości inicjujących atrybuty, etykiet komunikatów, nazw weryfikowanych atrybutów itp. Można w nich umieszczać wszelkiego rodzaju *parametry* konfiguracyjne obiektów lub ich pośredników. Konkretnie przykłady użycia argumentów dekoratorów zobaczymy w dalszej części rozdziału.

## Dekoratory zarządzają także funkcjami i klasami

Choć większa część reszty niniejszego rozdziału skupia się na opakowywaniu późniejszych wywołań funkcji oraz klas, powinienem podkreślić, że mechanizm dekoratorów jest o wiele bardziej uniwersalny — jest to protokół służący do przekazywania funkcji oraz klas za pośrednictwem obiektów wywoływalnych natychmiast po ich utworzeniu. Tym samym można go również wykorzystać do wywołania dowolnego przetwarzania po utworzeniu:

```
def decorate():
    # Zapisanie lub rozszerzenie funkcji bądź klasy O
    return O

@decorator
def F(): ...                                # F = decorator(F)

@decorator
class C: ...                                # C = decorator(C)
```

Dopóki będziemy zwracać w ten sposób oryginalny udekorowany obiekt, a nie obiekt opakowujący, możemy zarządzać *samymi* funkcjami i klasami, a nie tylko późniejszymi ich wywołaniami. Bardziej realistyczne przykłady takiego działania zobaczymy w dalszej części rozdziału, gdzie technika ta posłuży do rejestrowania obiektów wywoływalnych w API za pomocą dekoracji oraz przypisywania atrybutów do funkcji w czasie ich tworzenia. Zastosowania dekoratorów są różnorodne i tylko od Twojej wyobraźni zależy, jak je wykorzystasz.

## Kod dekoratorów funkcji

Przejdźmy do kodu. W dalszej części niniejszego rozdziału będziemy omawiali działające przykłady demonstrujące zaprezentowane właśnie kwestie związane z dekoratorami. W niniejszym podrozdziale zaprezentujemy kilka przykładów działania dekoratorów funkcji, natomiast w kolejnym — działanie dekoratorów klas. Na koniec zamknijemy rozdział kilkoma większymi studiami przypadku użycia dekoratorów klas oraz funkcji — pełną implementacją prywatności klas i sprawdzaniem zakresów wartości argumentów.

## Śledzenie wywołań

Na początek wrócimy do przykładu śledzenia wywołań z rozdziału 32. Przykład 39.1 definiuje i stosuje dekorator funkcji zliczający liczbę wywołań wykonywanych do udekorowanej funkcji, a także wyświetlający komunikat śledzenia dla każdego wywołania.

Przykład 39.1. *decorator1.py*

```
class tracer:
    def __init__(self, func):                # W momencie dekoracji @: zapisanie oryginalnej funkcji
        self.calls = 0
        self.func = func
    def __call__(self, *args):               # W momencie późniejszego wywołania:
                                                # wykonanie oryginalnej funkcji
        self.calls += 1
```

```

        print(f'wywołanie {self.calls} to {self.func.__name__}')
        self.func(*args)

@tracer
def hack(a, b, c):
    print(a + b + c)

```

*# hack = tracer(hack)*  
*# Opakowuje hack w obiekt dekoratora*

Warto przyjrzeć się temu, jak każda funkcja udekorowana za pomocą tej klasy będzie tworzyła nową instancję, z własnym zapisanym obiektem funkcji oraz licznikiem wywołań. Na uwagę zasługuje także to, w jaki sposób składnia argumentów `*args` wykorzystywana jest do spakowania i rozpakowania dowolnej liczby przekazanych argumentów. Taka uniwersalność pozwala na wykorzystanie dekoratora do opakowania dowolnej funkcji z dowolną liczbą argumentów (powyższa wersja nie działa jeszcze na metodach klas, ale w dalszej części podrozdziału to poprawimy).

Jeśli teraz zaimportujemy funkcję z tego modułu i przetestujemy ją w sesji interaktywnej, otrzymamy następujące działanie. Każde wywołanie generuje początkowo komunikat śledzenia, ponieważ przechwytywane jest przez klasę dekoratora:

```

$ python3
>>> from decorator1 import hack

>>> hack(1, 2, 3)
wywołanie 1 hack
6
# Tak naprawdę wywołuje opakowujący obiekt śledzenia

>>> hack('a', 'b', 'c')
wywołanie 2 hack
abc
# Wywołuje metodę __call__ w klasie

>>> hack.calls
2
# Zliczenie wywołań w informacjach o stanie obiektu opakowującego

>>> hack
<decorator1.tracer object at 0x02D9A730>

```

Po wykonaniu klasa `tracer` zapisuje udekorowaną funkcję i przechwytuje jej późniejsze wywołania w celu dodania warstwy logiki zliczającej i wyświetlającej każde wywołanie. Warto zwrócić uwagę na pokazywanie się całkowitej liczby wywołań jako atrybutu udekorowanej funkcji — `hack` jest tak naprawdę instancją klasy `tracer` po udekorowaniu (odkrycie to może mieć konsekwencje dla programów wykonujących sprawdzanie typów, ale jest ogólnie nieszkodliwe).

W przypadku wywołań funkcji składnia dekoracji ze znakiem `@` może być wygodniejsza od modyfikowania każdego wywołania w celu uzyskania dodatkowego poziomu logiki, gdyż pozwala uniknąć przypadkowego bezpośredniego wywołania oryginalnej funkcji. Rozważmy odpowiednik tego kodu *niezawierający dekoratora*, jak poniższy:

```

>>> calls = 0
>>> def tracer(func, *args):
    global calls
    calls += 1
    print(f'wywołanie {calls} to {func.__name__}')
    func(*args)

```



```
>>> def hack(a, b, c):                                # Nieudekorowana funkcja
    print(a, b, c)

>>> hack(1, 2, 3)                                     # Normalne, niesledzone wywołanie: przypadkowe?
1 2 3

>>> tracer(hack, 1, 2, 3)                             # Specjalne śledzone wywołanie bez dekoratorów
wywołanie 1 spam
1 2 3
```

Powyższą alternatywę można wykorzystać w dowolnej funkcji bez specjalnej składni ze znakiem @, jednak w przeciwieństwie do wersji z dekoratorem wymaga ona dodania dodatkowej składni w każdym miejscu, w którym w kodzie *wywoływana* jest funkcja. Co więcej, jej cel nie jest tak oczywisty i brak jest także zapewnienia, że dodatkowa warstwa zostanie wywołana dla normalnych wywołań. Choć dekoratory nigdy nie są *wymagane* (nazwy zawsze możemy dowiązać ponownie ręcznie), często są najwygodniejszą opcją.

## Możliwości w zakresie zachowania informacji o stanie

Ostatni przykład zwraca uwagę na pewną istotną kwestię. Dekoratory funkcji mają kilka możliwości w zakresie zachowywania informacji o stanie udostępnianych w czasie dekoracji i wykorzystywanych w czasie samego wywołania funkcji. Zazwyczaj muszą one obsługiwać większą liczbę udekorowanych obiektów oraz wywołań, jednak istnieje kilka sposobów implementacji tych celów — do zachowania stanu można wykorzystać atrybuty instancji, zmienne globalne, zmienne nielokalne, a także atrybuty funkcji.

Ten temat jest podobny do omówienia stanu początkowego w rozdziale 17., ale tutaj można go rozwinąć o szczegóły dotyczące klas i jest tak powszechny w przypadku dekoratorów, że kwalifikuje się jako podstawa. Zagadnienie to odnosi się zarówno do dekoratorów funkcji, jak i klas, ale skoncentrujemy się na węższym obszarze dekoratorów funkcji.

### Stan z atrybutami instancji klasy

Przykład 39.2 to rozszerzona wersja poprzedniego przykładu, dodająca obsługę argumentów ze słowami kluczowymi i operatorem \*\*, zwracająca wynik opakowanej funkcji, tak by możliwa była obsługa większej liczby zastosowań. (Dla tych czytelników, którzy nie czytają rozdziałów kolejno jeden po drugim: pierwsze argumenty ze słowami kluczowymi zostały opisane w rozdziale 18.).

Przykład 39.2. *decorator\_state\_classes.py*

```
class tracer:
    def __init__(self, func):                # Stan w atrybutach instancji
        self.calls = 0                     # W momencie dekoracji @
        self.func = func                   # Zapisanie func dla późniejszego wywołania

    def __call__(self, *args, **kwargs):    # W momencie wywołania oryginalnej funkcji
        self.calls += 1
        print(f'wywołanie {self.calls} to {self.func.__name__}')
        return self.func(*args, **kwargs)
```

```

@tracer
def hack(a, b, c):
    print(a + b + c)

# To samo co: hack = tracer(hack)
# Uruchamia tracer.__init__

@tracer
def code(x, y):
    print(x ** y)

# To samo co: code = tracer(code)
# Opakowuje code w obiekt tracer

hack(1, 2, 3)
hack(a=4, b=5, c=6)

# Tak naprawdę wywołuje instancję tracer: wykonuje tracer.__call__
# hack jest atrybutem instancji

code(2, 16)
code(4, y=4)

# Tak naprawdę wywołuje instancję tracer, self.func to code
# self.calls zliczane tutaj per funkcja (potrzebna instrukcja nonlocal z 3.0)

```

Tak jak wersja oryginalna, powyższy kod wykorzystuje *atrybuty instancji klasy* do zapisania stanu w sposób jawny. Zarówno opakowana funkcja, jak i licznik wywołań są informacjami *per instancja* — każda dekoracja otrzymuje ich własną kopię. Po wykonaniu skryptu wynik powyższej wersji kodu będzie następujący. Warto zwrócić uwagę na to, że funkcje `hack` i `code` mają własne liczniki wywołań, ponieważ każda dekoracja tworzy nową instancję klasy:

```

$ python3 decorator_state_classes.py
wywołanie 1 hack
6
wywołanie 2 hack
15
wywołanie 1 code
16
wywołanie 2 code
65536

```

Choć przydaje się przy dekoracji funkcji, taki schemat kodu sprawia pewne problemy po zastosowaniu do *metod* (więcej informacji na ten temat nieco później).

## Stan ze zmiennymi globalnymi

W przypadku prostszych zadań, które nie wymagają osobnych danych na funkcję, przeniesienie zmiennych stanu na *poziom globalny* może wystarczyć, jak to pokazano w przykładzie 39.3. Ten kod nadal używa odwołania do oryginalnej dekorowanej funkcji w zakresie zewnętrznym, ale przenosi licznik wywołań do modułu otaczającego.

Przykład 39.3. *decorator\_state\_globals.py*

```

calls = 0
def tracer(func):
    def wrapper(*args, **kwargs):
        global calls
        calls += 1
        print(f'wywołanie {calls} to {func.__name__}')
        return func(*args, **kwargs)
    return wrapper

# Stan w zakresie zawierającym oraz zmiennej globalnej
# Zamiast atrybutów klas
# Zmienna calls jest globalna, a nie per funkcja

@tracer
def hack(a, b, c):
    # To samo co: hack = tracer(hack)

```

```

print(a + b + c)

@tracer
def code(x, y):
    print(x ** y)
    # To samo co: code = tracer(code)

if __name__ == '__main__':
    hack(1, 2, 3)
    hack(a=4, b=5, c=6)
    # Tak naprawdę wywołuje wrapper, dowiązany do hack
    # wrapper wywołuje hack

    code(4, 2)
    code(2, y=16)
    # Tak naprawdę wywołuje wrapper, dowiązany do code
    # Zmienna globalna calls nie jest tutaj liczona per funkcja!

```

Niestety, przeniesienie licznika do wspólnego zakresu globalnego, tak by mógł on być modyfikowany w ten sposób, oznacza również, że będzie on *współdzielony* przez każdą opakowaną funkcję. W przeciwieństwie do atrybutów instancji klas liczniki globalne są wspólne dla programu, a nie per funkcja — licznik inkrementowany jest z *każdym* wywołaniem śledzonej funkcji. Różnica staje się zauważalna, jeśli porównamy wynik tej wersji z wynikiem poprzedniej — jeden współdzielony, globalny licznik wywołań jest niepoprawnie uaktualniany przez wywołania każdej udekorowanej funkcji:

```

$ python3 decorator_state_globals.py
wywołanie 1 hack
6
wywołanie 2 hack
15
wywołanie 1 code
16
wywołanie 2 code
65536

```

## Stan ze zmiennymi nielokalnymi w zasięgu zawierającym

Współdzielony stan globalny może w pewnych przypadkach być tym, czego chcemy. Jeśli jednak potrzebny jest nam licznik *per funkcja*, możemy albo użyć klas (jak wcześniej), albo skorzystać z funkcji domykających i instrukcji `nonlocal`, opisanej w rozdziale 17. Ponieważ ta instrukcja pozwala na *modyfikowanie* zmiennych z zakresu funkcji zawierającej, może służyć jako zmienne dane per dekoracja. Przykład 39.4 pokazuje ten schemat w podstawowej wersji.

Przykład 39.4. *decorator\_state\_nonlocals.py*

```

def tracer(func):
    calls = 0
    def wrapper(*args, **kwargs):
        nonlocal calls
        calls += 1
        print(f'wywołanie {calls} to {func.__name__}')
        return func(*args, **kwargs)
    return wrapper

@tracer
def hack(a, b, c):
    print(a + b + c)
    # Stan w zakresie zawierającym i zmiennej nielokalnej
    # Zamiast atrybutów klasy lub zmiennej globalnej
    # Zmienna calls jest per funkcja, a nie globalna
    # To samo co: hack = tracer(hack)

```

```

@tracer
def code(x, y):
    print(x ** y)
    # To samo co: code = tracer(code)

if __name__ == '__main__':
    hack(1, 2, 3)
    hack(a=4, b=5, c=6)
    # Tak naprawdę wywołuje wrapper, dowiązany do hack
    # wrapper wywołuje hack

    code(4, 2)
    code(2, y=16)
    # Tak naprawdę wywołuje wrapper, dowiązany do code
    # Zmienna nielokalna calls nie jest tutaj per funkcja

```

Teraz, ponieważ zmienne z zakresu funkcji zawierającej nie są globalne dla całego programu, każda opakowana funkcja otrzymuje znowu własny licznik, tak jak to było w przypadku klas i atrybutów. Oto wynik wykonania powyższego kodu w Pythonie 3.x:

```

$ python3 decorator_state_nonlocals.py
wywołanie 1 hack
6
wywołanie 2 hack
15
wywołanie 1 code
16
wywołanie 2 code
65536

```

## Stan z atrybutami funkcji

Wreszcie, możemy ominąć zmienne globalne i klasy, na potrzeby zmiennego stanu wykorzystując *atrybuty funkcji*. Jak widzieliśmy w rozdziałach 17. i 19., możemy za pomocą zapisu *funkcja.atrybut=wartość* przypisywać dowolne atrybuty do funkcji w celu ich dołączania. Ponieważ funkcja fabryczna przy każdym wywołaniu tworzy nową funkcję, więc jej atrybuty przechowują stan. Co więcej, ta technika jest potrzebna tylko w przypadku stanów, które muszą się *zmieniać*. Odwołania otaczające zakres są zachowywane i działają automatycznie.

W przykładzie 39.5. możemy po prostu na potrzeby zapisania stanu wykorzystać kod `wrapper.calls`. Ten kod działa tak samo jak poprzednia wersja ze zmiennymi nielokalnymi, ponieważ licznik znów działa per udekorowana funkcja.

Przykład 39.5. *decorator\_state\_attributes.py*

```

def tracer(func):
    def wrapper(*args, **kwargs):
        wrapper.calls += 1
        print(f'wywołanie {wrapper.calls} to {func.__name__}')
        return func(*args, **kwargs)
        wrapper.calls = 0
    return wrapper
    # Stan w zakresie funkcji zawierającej i atrybucie funkcji
    # Zmienna calls jest per funkcja, a nie globalna

@tracer
def hack(a, b, c):
    print(a + b + c)
    # To samo co: hack = tracer(hack)

@tracer
def code(x, y):
    print(x ** y)
    # To samo co: code = tracer(code)

```

```

if __name__ == '__main__':
    hack(1, 2, 3)                # Tak naprawdę wywołuje wrapper, dowiązany do hack
    hack(a=4, b=5, c=6)         # wrapper wywołuje hack

    code(4, 2)                  # Tak naprawdę wywołuje wrapper, dowiązany do code
    code(2, y=16)               # Zmienna wrapper.calls działa tutaj per funkcja

```

Jak wiemy z rozdziału 17., rozwiązanie to działa tylko dzięki temu, że nazwa `wrapper` zachowywana jest w zakresie funkcji zawierającej `tracer`. Gdy później inkrementujemy zmienną `wrapper.calls`, nie zmieniamy samej zmiennej `wrapper`, dlatego deklaracja `nonlocal` nie jest wymagana:

```

$ python3 decorator_state_attributes.py
...wynik taki sam jak poprzednio, ale kod działa również w wersji 2.x...

```

Powyższe rozwiązanie niemalże trafiło do przypisu, ponieważ jest mniej oczywiste niż deklaracja `nonlocal` i lepiej jest zostawić je na potrzeby przypadków, w których inne rozwiązania zawodzą. Jednak atrybuty funkcji mają niezaprzeczalne zalety. Po pierwsze dają dostęp do zapisanego stanu *spoza* kodu dekoratora. Zmienne nielokalne są widoczne tylko wewnątrz zagnieżdżonej funkcji, natomiast atrybuty funkcji mają szerszy zakres widoczności.

Atrybuty funkcji wykorzystamy jednak w odpowiedzi do jednego z pytań kończących rozdział, w którym ich widzialność poza obiektem wywoływalnym jest zaletą. Jako zmienne stanu skojarzone z kontekstem, w którym są stosowane, są odpowiednikami zmiennych nielokalnych w otaczającym zakresie. Jak zwykle, wybór jednego z wielu dostępnych narzędzi jest nieodłączną częścią programowania.

Ponieważ dekoratory często wymuszają kilka poziomów obiektów wywoływalnych, możemy połączyć funkcje z zakresami zawierającymi i klasami z atrybutami w celu uzyskania różnych rodzajów struktur kodu. Jak jednak zobaczymy nieco później, czasami różnice mogą być bardziej subtelne, niż moglibyśmy się tego spodziewać — każda udekorowana funkcja powinna mieć własny stan, a każda udekorowana klasa może wymagać stanu zarówno dla siebie samej, jak i dla każdej wygenerowanej instancji.

Tak naprawdę, jak zobaczymy w kolejnym podrozdziale, jeśli chcemy zastosować dekoratory funkcji także do metod klas, musimy również uważać na rozróżnienie, które Python robi pomiędzy dekoratorami zapisanymi w postaci wywoływalnych obiektów instancji klas a dekoratorami zapisanymi w postaci funkcji.

## Pułapki związane z klasami — dekorowanie metod klas

Kiedy napisałem pierwszy dekorator funkcji `tracer` w przykładzie 39.2, naiwnie założyłem, że można go również zastosować do dowolnej *metody* — udekorowane metody powinny działać tak samo, jednak automatyczny argument instancji `self` miałby zostać dołączony na początku `*args`. Niestety *nie miałem racji*, choć powody mojego niepowodzenia nie są oczywiste.

W skrócie: po zastosowaniu do metody klasy pierwsza wersja dekoratora `tracer` nie działa, ponieważ `self` jest instancją klasy dekoratora, a instancja udekorowanej klasy podmiotowej nie

zostaje dołączona do `*args`. Oto przypomnienie klasy, o której mowa, abyś nie musiał przewracać stron:

```
class tracer:
    def __init__(self, func):
        self.calls = 0
        self.func = func
    def __call__(self, *args, **kwargs):
        self.calls += 1
        print(f'wywołanie {self.calls} to {self.func.__name__}')
        return self.func(*args, **kwargs)
```

*# Stan w atrybucie instancji*  
*# W momencie dekoracji @*  
*# Zapisanie func dla późniejszego wywołania*  
*# W momencie wywołania oryginalnej funkcji*

To zjawisko było omawiane wcześniej w tym rozdziale, ale teraz możemy je zobaczyć w kontekście działającego kodu. Dekorator oparty na klasie z przykładu 39.2 działa zgodnie z wcześniejszym opisem dla zwykłych funkcji (dla kopiujących i wklejających: nie kopiuj początkowych znaków ... z REPL zawartych w tym rozdziale, aby zachować wcięcia po wierszach dekoratora):

```
>>> from decorator_state_classes import tracer
>>> @tracer
... def hack(a, b, c):
    print(a + b + c)
```

*# hack = tracer(hack)*  
*# Uruchamia tracer.\_\_init\_\_*

```
>>> hack(1, 2, 3)
Wywołanie 1 hack
6
```

*# Uruchamia tracer.\_\_call\_\_*

```
>>> hack(a=4, b=5, c=6)
Wywołanie 2 hack
15
```

*# hack zapisany w atrybucie instancji*

Dekoracja metody klasy nadal jednak nie działa (bardziej czujni Czytelnicy rozpoznają w tym przykładzie klasę `Person` wziętą z omówienia programowania zorientowanego obiektowo w rozdziale 28.):

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

    @tracer
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
```

*# giveRaise = tracer(giveRaise)*

```
>>> bob = Person('Robert Zielony', 50000)
>>> bob.giveRaise(.10)
```

*# tracer pamięta funkcję metod*  
*# Wykonuje tracer.\_\_call\_\_(???, .10)*

wywołanie 1 giveRaise  
TypeError: giveRaise() missing 1 required positional argument: 'percent'

Sedno problemu leży w argumencie `self` metody `__call__` klasy `tracer` — czy jest to instancja klasy `tracer`, czy może instancja klasy `Person`? W obecnej postaci kodu potrzebujemy *obu* — instancji `tracer` dla stanu dekoratora, natomiast instancji `Person` dla przekierowania do oryginalnej metody. Tak naprawdę `self` musi jednak być obiektem `tracer`, tak byśmy mogli uzyskać dostęp do informacji o stanie klasy `tracer`. Będzie to prawdziwe bez względu na to, czy dekorujemy prostą funkcję, czy też metodę.

Niestety, gdy nazwa naszej udekorowanej metody zostaje ponownie dowiązana do obiektu instancji klasy za pomocą metody `__call__`, Python przekazuje do `self` jedynie instancję `tracer`. Nie przekazuje w liście argumentów elementu `Person`. Co więcej, ponieważ klasa `tracer` nie wie nic o instancji `Person`, którą próbujemy przetwarzać za pomocą wywołań metod, nie ma możliwości utworzenia metody dowiązanej do instancji i tym samym nie da się w sposób poprawny wykonać wywołań. Nie jest to błąd, tylko niezwykle subtelna zawiłość.

Tak naprawdę powyższa wersja kodu przekazuje zbyt małą liczbę argumentów do udekorowanej metody i kończy się zwróceniem błędu. Można to zweryfikować, dodając do metody `__call__` dekoratora wiersz wyświetlający wszystkie argumenty. Jak widać, `self` to instancja klasy `tracer`, natomiast instancji klasy `Person` w ogóle nie ma:

```
>>> bob.giveRaise(.10)
<__main__.tracer object at 0x02D6AD90> (0.10,) {}
```

Jak wspomniano wcześniej, ma to miejsce, gdyż Python przekazuje domniemaną instancję podmiotową do `self`, gdy nazwa metody dowiązywana jest jedynie do prostej funkcji. Kiedy jest to instancja klasy wywoływalnej, zamiast tego przekazywana jest instancja tej klasy. Python tworzy *obiekt dowiązanej metody* zawierający podmiotową instancję, *tylko* wtedy, gdy metoda jest prostą funkcją, a nie wywoływalną instancją innej klasy.

## Wykorzystywanie zagnieżdżonych funkcji do dekoracji metod

Jeśli chcemy, by dekoratory funkcji działały *zarówno* na prostych funkcjach, *jak i* na metodach klas, najprostszym rozwiązaniem jest wykorzystanie jednego z opisanych wcześniej rozwiązań dla zachowywania stanu. Dekorator funkcji należy zapisać w kodzie w postaci *zagnieżdżonych* instrukcji `def`, tak byśmy nie musieli polegać na pojedynczym argumencie instancji `self`, który ma być zarówno instancją klasy opakowującej, jak i instancją klasy podmiotowej.

Już to widzieliśmy — zarówno przykład 39.4, jak i 39.5 działają zarówno dla funkcji, jak i dla metod klas, wykorzystując zagnieżdżone funkcje wraz z nielokalnymi zmiennymi lub atrybutami funkcji:

```
>>> from decorator_state_nonlocals import tracer    # Zobacz przykład 39.4

>>> @tracer
... def hack(a, b, c):                             # Działa dla funkcji
    print(a + b + c)

>>> hack(1, 2, 3)
wywołanie 1 hack
6

>>> class Person:
    def __init__(self, name, pay):                 # I działa dla metod
        self.name = name
        self.pay = pay
    @tracer
    def giveRaise(self, percent):                  # self zawarte w argumentach
        self.pay *= (1.0 + percent)                # Licznik w zmiennej nielokalnej
```

```
>>> bob = Person('Robert Zielony', 50000)
>>> bob.giveRaise(.10)
wywołanie 1 giveRaise
>>> bob.giveRaise(.10)
wywołanie 2 giveRaise
>>> f'{bob.pay:,.2f}'
'60,500.00'

>>> from decorator_state_attributes import tracer           # Zobacz przykład 39.5
...te same prawidłowe wyniki...                             # Licznik w atrybutach
```

Ponieważ dekorowane metody są tutaj ponownie wiązane z prostymi funkcjami zamiast z obiektami instancji, Python poprawnie przekazuje obiekt `Person` jako pierwszy argument, a dekorator propaguje go jako pierwszy element `*args` do argumentu `self` rzeczywistych, dekorowanych metod. Przeanalizuj te wyniki i dekoratory, aby upewnić się, że rozumiesz ten model. Następny punkt przedstawia alternatywę, która obsługuje klasy, ale jest znacznie bardziej skomplikowana.

## Wykorzystywanie deskryptorów do dekorowania metod

Choć rozwiązanie z zagnieżdżonymi funkcjami przedstawione wyżej jest najprostszym sposobem obsługi dekoratorów mających zastosowanie zarówno do funkcji, jak i do metod klas, możliwe są również inne rozwiązania. Mogą nam tutaj pomóc na przykład *deskryptory* omówione w poprzednim rozdziale.

Przypomnijmy, że zgodnie z informacjami z poprzedniego rozdziału deskryptor może być atrybutem klasy przypisywanym do obiektów za pomocą metody `__get__` wykonywanej automatycznie wtedy, gdy następuje odwołanie się do atrybutu oraz jego pobranie:

```
class Descriptor:
    def __get__(self, instance, owner): ...

class Subject:
    attr = Descriptor()

X = Subject()
X.attr                                     # Wykonuje w przybliżeniu Descriptor.__get__(Subject.attr, X, Subject)
```

Deskryptory mogą również zawierać metody dostępu `__set__` oraz `__del__`, których tutaj jednak nie potrzebujemy. Ponieważ metoda `__get__` deskryptora otrzymuje po wywołaniu instancje klasy deskryptora, *jak i* klasy podmiotowej, dobrze nadaje się do dekorowania metod, kiedy przy wywołaniach potrzebne są nam zarówno stan dekoratora, jak i instancja oryginalnej klasy. Rozważmy następujący wariant dekoratora śledzącego w przykładzie 39.6, będącego również deskryptorem, jeżeli zastosuje się go na poziomie metod klasy.

Przykład 39.6. *calltracer\_desc\_class.py*

```
class tracer(object):
    def __init__(self, func):           # Dekorator i deskryptor
        self.calls = 0                 # W momencie dekoracji @
        self.func = func               # Zapisanie func na potrzeby późniejszego wywołania
    def __call__(self, *args, **kwargs): # W momencie wywołania oryginalnej funkcji
```



```

        self.calls += 1
        print(f'wywołanie {self.calls} to {self.func.__name__}')
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):          # W momencie pobrania atrybutu metody
        return wrapper(self, instance)

class wrapper:
    def __init__(self, desc, subj):              # Zapisanie obu instancji
        self.desc = desc                        # Przekierowanie wywołań z powrotem do dekoratora
        self.subj = subj
    def __call__(self, *args, **kwargs):
        return self.desc(self.subj, *args, **kwargs)    # Wykonuje tracer.__call__

@tracer
def hack(a, b, c):                              # hack = tracer(hack)
    ...                                           # Wykorzystuje jedynie __call__

class Person:
    ...
    @tracer
    def giveRaise(self, percent):                # giveRaise = tracer(giveRaise)
        ...                                     # Sprawia, że giveRaise staje się deskryptorem

```

Takie rozwiązanie działa tak samo jak poprzedni kod funkcji zagnieżdżonej. Wykonywane operacje są różne w zależności od kontekstu:

- Udekorowane *funkcje* wywołują jedynie metodę `__call__`, natomiast nowsze wywołują `__get__`.
- Udekorowane *metody* wywołują najpierw metodę `__get__` w celu przeanalizowania pobrania nazwy metody (dla *instancja.metoda*). Obiekt zwracany przez metodę `__get__` zachowuje instancję klasy podmiotowej i jest następnie wywoływany w celu zakończenia wyrażenia wywołania, uruchamiając metodę `__call__` (dla `()`).

Przykładowo poniższe wywołanie kodu testu:

```
anna.giveRaise(.10)          # Wykonuje __get__, a następnie __call__
```

wykonuje najpierw metodę `tracer.__get__`, ponieważ atrybut `giveRaise` klasy `Person` został ponownie dowiązany do deskryptora za pomocą dekoratora funkcji. Wyrażenie wywołania uruchamia następnie metodę `__call__` zwracanego obiektu opakowującego, która z kolei wywołuje metodę `tracer.__call__`. Innymi słowy, wywołanie udekorowanej metody uruchamia *pięciopięt* proces: wywołanie `tracer.__get__`, po nim wywołanie metody `wrapper.__init__`, następnie metod `wrapper.__call__` i `tracer.__call__`, a na koniec wywołanie oryginalnej, opakowanej metody.

Obiekt `wrapper` zachowuje zarówno instancję deskryptora, jak i instancję podmiotową, dlatego może przekazać sterowanie z powrotem do instancji oryginalnej klasy dekoratora (czy deskryptora). W rezultacie obiekt `wrapper` zapisuje instancję podmiotowej klasy dostępną w trakcie pobrania atrybutu metody i dodaje ją do listy argumentów późniejszego wywołania, przekazywanej do metody `__call__`. Przekierowanie wywołania w ten sposób z powrotem do instancji klasy deskryptora wymagane jest w tej aplikacji, by wszystkie wywołania opakowanej metody wykorzystywały te same informacje o stanie licznika `calls` w obiekcie instancji deskryptora.

Alternatywnie moglibyśmy wykorzystać funkcję zagnieżdżoną i referencje do zakresu funkcji zawierającej w celu uzyskania tego samego efektu. Przykład 39.7 działa tak samo jak poprzednia, zamieniając jednak klasę i atrybuty obiektów na funkcję zagnieżdżoną i referencje do zakresu. Wymaga też wyraźnie mniej kodu, ale to ten sam wieloetapowy proces dla każdego wywołania udekorowanej metody.

Przykład 39.7. *calltracer\_desc\_func.py*

```
class tracer(object):
    def __init__(self, func):
        self.calls = 0
        self.func = func
    def __call__(self, *args, **kwargs):
        self.calls += 1
        print(f'wywołanie {self.calls} to {self.func.__name__}')
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):
        def wrapper(*args, **kwargs):
            return self(instance, *args, **kwargs)
        return wrapper
```

...reszta tak samo jak w przykładzie 39.6...

Te dwa śledzenia oparte na deskryptorach działają tak samo jak wersja z zagnieżdżonymi funkcjami, więc pominiemy tutaj ich wyniki. Jeśli to pomoże, dodaj instrukcje `print` do ich metod, aby prześledzić ich wieloetapowe procesy pobierania i wywoływania. W obu przypadkach ten schemat oparty na deskryptorach jest również znacznie subtelniejszy niż opcja z zagnieżdżonymi funkcjami, więc prawdopodobnie jest to drugorzędna opcja do wyboru. Mówiąc wprost: jeśli jego złożoność nie przyprawia Cię o zawroty głowy, to koszty wydajności raczej powinny! Niemniej jednak może to być przydatny wzorec kodowania w innych kontekstach.

Zanim przejdziemy dalej, warto również wspomnieć, że taki dekorator oparty na deskryptorze można zakodować w prostszy sposób, jak w przykładzie 39.8, który jednak *dotyczy tylko metod*, a nie prostych funkcji. Wynika to z wewnętrznego ograniczenia deskryptorów atrybutów (i stanowi negację problemu, który chcieliśmy rozwiązać: znaleźć dekorator zarówno dla funkcji, jak i metod).

Przykład 39.8. *calltracer\_desc\_fail.py*

```
class tracer(object):
    def __init__(self, meth):
        self.calls = 0
        self.meth = meth
    def __get__(self, instance, owner):
        def wrapper(*args, **kwargs):
            self.calls += 1
            print(f'wywołanie {self.calls} to {self.meth.__name__}')
            return self.meth(instance, *args, **kwargs)
        return wrapper
```

```
@tracer
# Działa dla metod, ale NIE DZIAŁA
# w przypadku funkcji
```

```
def hack(a, b, c):
    ...
# hack = tracer(hack)
# Tutaj nie jest pobierany żaden atrybut

...reszta tak samo jak w przykładzie 39.6...
```

W dalszej części rozdziału będziemy raczej wykorzystywać klasy lub funkcje w celu zapisania w kodzie dekoratorów funkcji, o ile będą one miały zastosowanie jedynie do funkcji. Niektóre dekoratory mogą nie wymagać instancji oryginalnej klasy i nadal będą działały zarówno na funkcjach, jak i metodach, jeśli zapiszemy je w postaci klas. Coś takiego jak własny dekorator `staticmethod` Pythona nie wymaga na przykład instancji klasy podmiotowej (i tak naprawdę jedynym jego celem jest usunięcie instancji z wywołania).

Morał płynący z tej historii jest jednak taki, że jeśli chcemy, by nasze dekoratory działały zarówno na prostych funkcjach, jak i na metodach klas, lepiej będzie wykorzystać przedstawiony tutaj wzorzec kodu oparty na *funkcji zagnieżdżonej* zamiast klasy z przechwytywaniem wywołań.

## Mierzenie czasu wywołania

By w nieco szerszym zakresie wypróbować możliwości dekoratorów funkcji, przejdźmy do innego przypadku użycia. Nasz kolejny dekorator mierzy czas trwania *wywołań* udekorowanej funkcji — zarówno dla pojedynczego wywołania, jak i całkowity czas dla wszystkich wywołań. W przykładzie 39.9 dekorator zostanie zastosowany do dwóch funkcji w celu porównania czasu wymaganego do tworzenia listy składanej oraz wywołania funkcji wbudowanej `map`.

Przykład 39.9. *timerdeco1.py*

```
"Uwaga: tak zakodowana klasa timer nie działa w przypadku metod
➡(patrz rozwiązanie quizu)"
import time, sys

class timer:
    def __init__(self, func):
        self.func = func
        self.alltime = 0
    def __call__(self, *args, **kargs):
        start = time.perf_counter()
        result = self.func(*args, **kargs)
        elapsed = time.perf_counter() - start
        self.alltime += elapsed
        print(f'{self.func.__name__}: {elapsed:.5f}, {self.alltime:.5f}')
        return result

@timer
def listcomp(N):
    return [x * 2 for x in range(N)]

@timer
def mapcall(N):
    return force(map((lambda x: x * 2), range(N)))

if __name__ == '__main__':
    for func in (listcomp, mapcall):
```

```

result = func(5)                                # Czas dla tego wywołania
func(50000)
func(500000)
func(1000000)
print(result)
print(f'allTime = {func.alltime}\n')           # Całkowity czas dla wszystkich wywołań func

print('**map/comp =', round(mapcall.alltime / listcomp.alltime, 3))

```

Po uruchomieniu powyższego kodu na komputerze z systemem macOS i z CPythonem 3.12 otrzymamy przedstawiony niżej wynik zawierający nazwę wywoływanej funkcji, czas jej wykonania, całkowity czas dotychczasowych wywołań, wynik zwrócony po pierwszym wywołaniu, całkowity czas wszystkich wywołań i na końcu stosunek czasów wywołań obu funkcji:

```

$ python3 timerdecor1.py
listcomp: 0.00000, 0.00000
listcomp: 0.00366, 0.00366
listcomp: 0.03134, 0.03500
listcomp: 0.05213, 0.08713
[0, 2, 4, 6, 8]
allTime = 0.08712841104716063

mapcall: 0.00001, 0.00001
mapcall: 0.00396, 0.00397
mapcall: 0.04082, 0.04479
mapcall: 0.07789, 0.12268
[0, 2, 4, 6, 8]
allTime = 0.12268476499593817

**map/comp = 1.408

```

Uzyskane wyniki zależą od wersji Pythona, szybkości komputera i innych czynników. Sumaryczny czas jest tutaj atrybutem instancji klasy. Jak zwykle funkcja `map` jest niemal dwukrotnie wolniejsza od listy składanej, która nie wywołuje żadnych funkcji (oznacza to, że funkcja `map` jest wolniejsza, ponieważ musi wywoływać funkcję).

Dla porównania w rozdziale 21. został opisany sposób pomiaru czasu wykonania iteracji *bez użycia dekoratorów*. Dla przypomnienia: użyte były tam dwie techniki umożliwiające mierzenie czasu pojedynczych wywołań — jedna własnego pomysłu, a druga wykorzystująca specjalną bibliotekę. Poniżej techniki te zostały użyte do pomiaru czasu przetwarzania wyrażenia listowego iterującego milion razy kod testujący dekorator, choć występuje tutaj dodatkowy koszt spowodowany kodem, co nieco wypacza wyniki (dodaj folder z rozdziału 21. do swojej zmiennej `PYTHONPATH` lub `sys.path` albo przejdź do tego folderu, aby uruchomić kod):

```

>>> def listcomp(N): [x * 2 for x in range(N)]

>>> import timer                                # Techniki z rozdziału 21.
>>> timer.total(1, listcomp, 1000000)
(0.08150088600814342, None)
>>> timer.bestoftotal(5, 1, listcomp, 1_000_000)
(0.059792334999656305, None)

>>> import timeit
>>> timeit.timeit(number=1, stmt=lambda: listcomp(1000000))

```

```
0.08125517799635418
>>> min(timeit.repeat(repeat=5, number=1, stmt=lambda: listcomp(1_000_000)))
0.06156357398140244
```

W tym przypadku rozwiązanie bez dekoratora pozwalałoby na wykorzystanie funkcji podmiotowych z pomiarem czasu bądź bez niego, jednak skomplikowałoby także sygnaturę wywołania, gdyby pomiar czasu był pożądanym (musielibyśmy dodawać kod z każdym wywołaniem zamiast raz w instrukcji `def`). Co więcej, sposób bez użycia dekoratora nie dawałby bezpośrednio gwarancji, że wszystkie wywołania budujące listę w programie zostałyby przekierowane przez logikę pomiaru czasu — poza próbą odnalezienia wszystkich i potencjalnego zmodyfikowania ich. Z tego powodu trudno byłoby zmierzyć skumulowany czas wszystkich wywołań.

Zazwyczaj *dekoratory* warto stosować wtedy, gdy funkcje są już wdrożone, stanowią części większego systemu i trudno byłoby je analizować za pomocą innych funkcji w takcie działania. Z drugiej strony, ponieważ dekoratory modyfikują każdą funkcję, wprowadzając do niej kody pomiarowe, podejście *bez dekoratorów* może okazać się lepsze, gdy trzeba monitorować wybrane wywołania. Jak zawsze, różne narzędzia pełnią różne role.

## Dodawanie argumentów dekoratora

Dekorator mierzący czas z poprzedniego podrozdziału działa, ale byłoby miło, gdyby był bardziej konfigurowalny. Podanie etykiety danych wyjściowych czy na przykład włączanie i wyłączanie komunikatów śledzenia mogłyby się przydać w tego typu uniwersalnym narzędziu. W takiej sytuacji przydają się *argumenty* dekoratora — po poprawnym dodaniu ich do kodu możemy je wykorzystać w celu określenia opcji konfiguracyjnych, które mogą się różnić dla każdej udekorowanej funkcji. Etykietę można na przykład dodać w następujący sposób:

```
def timer(label=''):
    def decorator(func):
        def onCall(*args):
            ...
            func(*args)
            print(label, ...)
        return onCall
    return decorator

@timer('==>')
def listcomp(N): ...

listcomp(...)
# Wielopoziomowe zachowanie stanu
# Argumenty przekazane do funkcji
# Funkcja zachowana w zakresie zawierającym
# Etykieta zachowana w zakresie zawierającym
# Zwraca sam dekorator
# Jak listcomp = timer('==>')(listcomp)
# Nazwa listcomp ponownie dowiązana do dekoratora
# Tak naprawdę wywołuje dekorator
```

Powyższy kod dodaje zakres funkcji zawierającej w celu zachowania argumentu dekoratora do zastosowania w późniejszym wywołaniu. Kiedy definiowana jest funkcja `listcomp`, tak naprawdę wywołuje ona funkcję `decorator` (wynik funkcji `timer`, wykonanej zanim nastąpi sama dekoracja) z wartością etykiety `label` dostępną w zakresie funkcji zawierającej. Oznacza to, że funkcja `timer` zwraca dekorator pamiętający zarówno argument dekoratora, jak i oryginalną funkcję oraz zwracający obiekt wywoływalny uruchamiający oryginalną funkcję w przypadku późniejszych wywołań. Ponieważ w ten sposób tworzony jest nowy dekorator i funkcja `onCall`, stan jest zachowywany tylko w obejmowanym zakresie.

Taką strukturę możemy wykorzystać w naszej funkcji mierzącej czas, tak by możliwe było przekazywanie etykiety oraz opcji sterowania śledzeniem w czasie dekoracji. Przykład 39.10 wykonyuje to działanie, tak by plik modułu mógł być importowany jako uniwersalne narzędzie. Do zachowania stanu wykorzystywana jest klasa, a nie zagnieżdżona funkcja, jednak ostateczny efekt jest podobny.

Przykład 39.10. *timerdeco2.py*

```
import time

def timer(label='', trace=True):          # Dla argumentów dekoratora: zachowanie argumentów
    class Timer:
        def __init__(self, func):        # Dla @: zachowanie udekorowanej funkcji
            self.func = func
            self.alltime = 0
        def __call__(self, *args, **kargs): # Dla wywołań: wywołanie oryginalnej funkcji
            start = time.perf_counter()
            result = self.func(*args, **kargs)
            elapsed = time.perf_counter() - start
            self.alltime += elapsed
            if trace:
                if label: print(label, end=' ')
                print(f'{self.func.__name__}: {elapsed:.5f}, {self.alltime:.5f}')
            return result
    return Timer
```

Większość naszej pracy polegała tutaj na osadzeniu oryginalnej klasy `Timer` w funkcji zawierającej w celu utworzenia zakresu zachowującego argumenty dekoratora. Zewnętrzna funkcja `timer` wywoływana jest przed wystąpieniem dekoracji i po prostu zwraca klasę `Timer`, która służy jako faktyczny dekorator. W momencie dekoracji tworzona jest instancja klasy `Timer`, pamiętająca samą udekorowaną funkcję i mająca również dostęp do argumentów dekoratora z zakresu funkcji zawierającej.

Tym razem zamiast osadzać kod testu samosprawdzającego w pliku, wykonamy dekorator w innym pliku. Przykład 39.11 to klient naszego dekoratora mierzącego czas, który ponownie zastosuje dekorator do alternatywnych rozwiązań w zakresie iteracji po sekwencjach.

Przykład 39.11. *testseqs.py*

```
import sys
from timerdeco2 import timer

@timer(label='[CCC]==>')
def listcomp(N):
    return [x * 2 for x in range(N)]

# Jak listcomp = timer(...) (listcomp)
# listcomp(...) uruchamia Timer.__call__

@timer(trace=True, label='[MMM]==>')
def mapcall(N):
    return map((lambda x: x * 2), range(N))

for func in (listcomp, mapcall):
    result = func(5)
    func(50000)

# Czas dla tego wywołania, wszystkich wywołań,
# zwracana wartość
```

```
func(500000)
func(1000000)
print(result)
print(f'allTime = {func.alltime}\n')    # Całkowity czas dla wszystkich wywołań
```

```
print('**map/comp = ', round(mapcall.alltime / listcomp.alltime, 3))
```

Po wykonaniu powyższy plik wyświetla następujące wyniki. Każda udekorowana funkcja ma teraz własną etykietę, zdefiniowaną za pomocą argumentów dekoratora. Jest to przydatne rozwiązanie w przypadku, gdy w większych wynikach zwracanych przez program trzeba odszukać pomiary czasu:

```
$ python3 testseqs.py
[CCC]==> listcomp: 0.00000, 0.00000
[CCC]==> listcomp: 0.00379, 0.00379
[CCC]==> listcomp: 0.03142, 0.03521
[CCC]==> listcomp: 0.05188, 0.08709
[0, 2, 4, 6, 8]
allTime = 0.08709081003325991

[MMM]==> mapcall: 0.00001, 0.00001
[MMM]==> mapcall: 0.00401, 0.00402
[MMM]==> mapcall: 0.04025, 0.04427
[MMM]==> mapcall: 0.07776, 0.12203
[0, 2, 4, 6, 8]
allTime = 0.12203056103317067

**map/comp = 1.401
```

Przeprowadź dodatkowe testy samodzielnie, aby zobaczyć, jak dokładnie działają argumenty konfiguracyjne dekoratora. Powyższy dekorator funkcji mierzący czas można wykorzystać dla dowolnej funkcji, zarówno w modułach, jak i w sesji interaktywnej. Innymi słowy, automatycznie można go zakwalifikować jako *narzędzie ogólnego przeznaczenia* służące do pomiaru czasu wykonywania kodu w skryptach. Następne przykłady argumentów dekoratora wkrótce, gdy będziemy implementować prywatność atrybutów i sprawdzanie zakresu argumentu za pomocą dekoratorów.



*Metody pomiaru czasu:* Dekorator `timer` z niniejszego podrozdziału działa na wszystkich *funkcjach*, jednak w celu zastosowania go również do *metod klas* wymagana jest niewielka modyfikacja. Mówiąc w skrócie, jak wspomnieliśmy w podrozdziale „Pułapki związane z klasami — dekorowanie metod klas”, należy unikać wykorzystywania zagnieżdżonej klasy. Ponieważ jednak taka zmiana będzie tematem pytań kończących rozdział, całkowicie pominę tutaj podanie pełnego rozwiązania.

## Kod dekoratorów klas

Dotychczas pisaliśmy kod dekoratorów funkcji zarządzających *wywołaniami* funkcji, jednak, jak widzieliśmy, dekoratory zostały rozszerzone w taki sposób, by działać również na klasach. Zgodnie z wcześniejszym opisem, choć są one podobne do dekoratorów funkcji, dekoratory

klas stosowane są zamiast tego do klas — można je wykorzystać albo do zarządzania samymi *klasami*, albo do przechwytywania wywołań tworzących instancje w celu zarządzania *instancjami*. Podobnie do dekoratorów funkcji, dekoratory klas są tak naprawdę składnią opcjonalną, choć wiele osób uważa, że pozwalają one uczynić intencje programisty bardziej oczywistymi, a także minimalizują liczbę błędnych wywołań.

## Klasy singletona

Ponieważ dekoratory klas mogą przechwytywać wywołania tworzące instancje, można je wykorzystać albo do zarządzania wszystkimi instancjami klasy, albo do rozszerzenia interfejsów tych instancji. By to zademonstrować, poniżej został zamieszczony przykład 39.12 z dekoratorem klasy, który wykonuje to pierwsze zadanie — zarządzania instancjami klasy. Kod ten implementuje klasyczny wzorec projektowy *singletona*, w którym istnieje maksymalnie jedna instancja klasy. Funkcja `singleton` definiuje i zwraca funkcję zarządzającą instancjami, natomiast składnia ze znakiem `@` automatycznie opakowuje w tę funkcję klasę podmiotową.

Przykład 39.12. *singletons1.py*

```
instances = {}

def singleton(aClass):
    def onCall(*args, **kwargs):
        if aClass not in instances:
            instances[aClass] = aClass(*args, **kwargs)
        return instances[aClass]
    return onCall
```

# W momencie dekoracji @  
# W momencie tworzenia instancji  
# Jeden wpis słownika na klasę

By wykorzystać powyższy kod, należy udekorować klasy, dla których chcemy wymusić model z pojedynczą instancją, jak w przykładzie 39.13.

Przykład 39.13. *singletons-test.py*

```
from singletons1 import singleton

@singleton
class Person:
    def __init__(self, name, hours, rate):
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate

@singleton
class Hack:
    def __init__(self, val):
        self.attr = val

anna = Person('Anna', 50, 20)
print(anna.name, anna.pay())

bob = Person('Robert', 40, 10)
print(bob.name, bob.pay())
```

# Person = singleton(Person)  
# Ponownie dowiązuje Person do onCall  
# onCall pamięta Person  
  
# Hack = singleton(Hack)  
# Ponownie dowiązuje Hack do onCall  
# onCall pamięta Hack  
  
# Tak naprawdę wywołuje onCall  
  
# Ten sam pojedynczy obiekt



```
X = Hack(42)
Y = Hack(99)
print(X.attr, Y.attr)
```

```
# Jeden obiekt Person, jeden obiekt Hack
```

Gdy klasa `Person` bądź `Hack` zostaje później wykorzystana do utworzenia instancji, warstwa logiki opakowującej udostępniana przez dekorator przekierowuje wywołania tworzące instancję do funkcji `onCall`, która z kolei wywołuje funkcję `getInstance` zarządzającą pojedynczą instancją na klasę i ją współdzielącą, bez względu na to, ile wywołań tworzących zostanie wykonanych. Oto wynik powyższego kodu uruchomionego poprzez wiersz poleceń:

```
$ python3 singletons.py
Robert 1000
Robert 1000
42 42
```

## Alternatywne rozwiązania

Co ciekawe, możemy także wykorzystać rozwiązanie alternatywne, jeśli jesteśmy w stanie użyć instrukcji `nonlocal` do modyfikacji zmiennych z zakresu zawierającego, zgodnie z wcześniejszym opisem. Poniższa alternatywa daje identyczny wynik, wykorzystując tylko jeden zakres funkcji zawierającej na klasę zamiast jednego wpisu do globalnej tabeli na klasę. Poniższa wersja działa tak samo, jednak nie jest uzależniona od nazw z zakresu globalnego poza dekoratorem. (Należy zwrócić uwagę, że w instrukcji warunkowej można użyć operatora `is` zamiast `==`; niemniej jednak jest to trywialne porównanie):

```
def singleton(aClass):
    instance = None
    def onCall(*args):
        nonlocal instance
        if instance == None:
            instance = aClass(*args)
        return instance
    return onCall
```

```
# W momencie dekoracji @
```

```
# W momencie tworzenia instancji
```

```
# Jeden zakres na klasę
```

Możemy także napisać takie samodzielne rozwiązanie za pomocą atrybutów funkcji lub klasy. W pierwszym z poniższych kodów wykorzystana jest pierwsza opcja. Wykorzystany jest fakt, że dekorowana jest tylko jedna *funkcja* `onCall`. Przestrzeń nazw obiektów pełni tutaj tę samą rolę, co otaczający zakres. W drugim kodzie do dekorowania wykorzystywana jest jedna *instancja*, a nie obejmujący zakres czy globalna tabela. Tak naprawdę wykorzystywany jest tu ten sam wzorzec projektowym, który później zobaczymy w często spotykanym rozwiązaniu związanym z dekoratorem klasy. Tutaj *potrzebna* jest nam tylko jedna instancja, jednak nie zawsze tak jest:

```
def singleton(aClass):
    def onCall(*args, **kwargs):
        if onCall.instance == None:
            onCall.instance = aClass(*args, **kwargs)
        return onCall.instance
    onCall.instance = None
    return onCall

class singleton:
```

```
# W momencie dekoracji @
```

```
# W momencie tworzenia instancji
```

```
# Jedna funkcja na klasę
```

```

def __init__(self, aClass):
    self.aClass = aClass
    self.instance = None
def __call__(self, *args, **kwargs):
    if self.instance == None:
        self.instance = self.aClass(*args, **kwargs) # Jedna instancja na klasę
    return self.instance

```

By uczynić z tego dekoratora prawdziwe narzędzie ogólnego przeznaczenia, należy przechować go w pliku modułu, który można importować, a także dokonać indentacji kodu testu samosprawdzającego pod sprawdzeniem `__name__` (pozostawię to jako sugerowane ćwiczenie). Ostatnia wersja oparta na klasie oferuje jawną opcję z dodatkowymi strukturami, które mogą lepiej wspierać późniejszy rozwój, ale programowanie obiektowe może nie być uzasadnione we wszystkich kontekstach.

## Śledzenie interfejsów obiektów

Przykład z singletonem z poprzedniego podrozdziału ilustrował użycie dekoratorów klas do zarządzania *wszystkimi* instancjami klasy. Kolejny często stosowany przypadek użycia dekoratorów klas rozszerza interfejs *każdej* z wygenerowanych instancji. Dekoratory klas mogą właściwie instalować na instancjach warstwę logiki opakowującej zarządzającą w jakiś sposób dostępem do ich interfejsów.

Przykładowo w rozdziale 31. metoda przeciążania operatora `__getattr__` zaprezentowana jest jako sposób opakowania całych interfejsów obiektów osadzonych instancji w celu zaimplementowania wzorca projektowego *delegacji*. Podobne przykłady widzieliśmy również w omówieniu zarządzanych atrybutów w poprzednim rozdziale. Warto przypomnieć, że metoda `__getattr__` jest wykonywana przy pobraniu niezdefiniowanej nazwy atrybutu. Ten punkt zaczepienia możemy wykorzystać do przechwytywania wywołań metod w klasie kontrolera i przekazywania ich do osadzonego obiektu.

### Śledzenie bez dekoratora

Jako punkt odniesienia poniżej znajduje się oryginalny przykład delegacji *niezawierający dekoratora*, działający na dwóch obiektach typów wbudowanych:

```

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

>>> x = Wrapper([1,2,3])
>>> x.append(4)

```

W powyższym kodzie klasa `Wrapper` przechwytuje próby dostępu do każdego z atrybutów opakowanego obiektu, wyświetla komunikat śledzenia i wykorzystuje funkcję wbudowaną `getattr` do przekazania żądania do opakowanego obiektu. W szczególności śledzi ona próby dostępu do atrybutów wykonywane *poza* klasą opakowanego obiektu. Próby dostępu wewnątrz metod

opakowanego obiektu nie są przechwytywane i są normalnie wykonywane. Ten model *całego interfejsu* różni się od zachowania dekoratorów funkcji, które opakowują tylko jedną, określoną metodę.

## Śledzenie interfejsów za pomocą dekoratorów klas

Dekoratory klas udostępniają alternatywny i wygodny sposób zapisu w kodzie techniki `__get` ➔ `__getattr__` służącej do opakowania całego interfejsu. Poprzedni przykład z klasą można zapisać w postaci dekoratora klasy uruchamiającego tworzenie opakowanej instancji, zamiast przekazywać przygotowaną wcześniej instancję do konstruktora obiektu opakowującego. Przykład 39.14 to rozszerzenie w celu obsługi argumentów ze słowami kluczowymi w `**kwargs`, a także zliczania liczby wykonanych prób dostępu, by pokazać zmienny stan.

Przykład 39.14. *interfacetracer.py*

```
def Tracer(aClass):                                     # W momencie dekoracji @
    class Wrapper:
        def __init__(self, *args, **kwargs):           # W momencie tworzenia instancji
            self.fetches = 0
            self.wrapped = aClass(*args, **kwargs)    # Użycie nazwy z zakresu funkcji zawierającej
        def __getattr__(self, attrname):               # Przechwytuje wszystko oprócz własnych
                                                    # atrybutów
            self.fetches += 1
            return getattr(self.wrapped, attrname)    # Delegacja do opakowanego obiektu
    return Wrapper

if __name__ == '__main__':
    @Tracer
    class Hack:                                         # Hack= Tracer(Hack)
        def display(self):                             # Klasa Hack ponownie dowiązana do Wrapper
            print('Hakować!' * 3)

    @Tracer
    class Person:                                     # Person = Tracer(Person)
        def __init__(self, name, hours, rate):         # Wrapper pamięta Person
            self.name = name
            self.hours = hours
            self.rate = rate
        def pay(self):                                # Próby dostępu spoza klasy są śledzone
            return self.hours * self.rate              # Próby dostępu z wewnątrz metody
                                                    # nie są śledzone

    work = Hack()                                     # Wywołuje Wrapper()
    work.display()                                    # Wywołuje __getattr__
    print([work.fetches])

    print()
    bob = Person('Robert', 40, 50)                   # Obiekt bob jest tak naprawdę
                                                    # instancją Wrapper
    print(bob.name)                                   # Wrapper osadza instancję Person
    print(bob.pay())
```

```

print()
anna = Person('Anna', rate=100, hours=60)    # Obiekt anna jest inną instancją Wrapper
print(anna.name)                             # Z inną klasą Person
print(anna.pay())

print()
print(bob.name)                               # Obiekt bob ma inny stan niż anna
print(bob.pay())
print('wywołanie:', [bob.fetches, anna.fetches]) # Atrybuty klasy Wrapper nie są śledzone

```

Istotne jest, by zwrócić uwagę na to, jak bardzo kod ten różni się od dekoratora śledzącego, z którym spotkał się wcześniej. W podrozdziale „Kod dekoratorów funkcji” przyglądaliśmy się dekoratorom pozwalającym na śledzenie i pomiar czasu *wywołań* określonej funkcji lub metody. W przeciwieństwie do nich dzięki przechwytywaniu wywołań tworzących instancje dekorator klasy pozwala nam śledzić pełny *interfejs* obiektu — czyli próby dostępu do dowolnego z jego atrybutów.

Ponadto warto zwrócić uwagę, że metoda `__getattr__` tego dekoratora nie przechwyci domyślnych pobrań atrybutów wbudowanych operacji, zgodnie z tym, co powiedzieliśmy we wcześniejszym rozdziale. Więcej na ten temat później, gdy zajmiemy się kodowaniem prywatności atrybutów.

Poniżej znajdują się dane wyjściowe zwracane przez powyższy kod. Próby pobrania atrybutów instancji zarówno klasy `Hack`, jak i `Person` wywołują logikę metody `__getattr__` klasy `Wrapper`, ponieważ obiekty `work` oraz `bob` są tak naprawdę instancjami klasy `Wrapper` dzięki przekierowaniu wywołań tworzących instancje przez dekorator:

```

$ python3 interfacetracer.py
Śledzenie: display
Hakować!Hakować!Hakować!
[1]

```

```

Śledzenie: name
Robert
Śledzenie: pay
2000

```

```

Śledzenie: name
Anna
Śledzenie: pay
6000

```

```

Śledzenie: name
Robert
Śledzenie: pay
wywołanie: 2000
[4, 2]

```

Należy zwrócić uwagę, że w jednej dekoracji wykorzystywana jest jedna klasa `Wrapper` z zachowaniem stanu, generowana przez zagnieżdżoną instrukcję `class` w funkcji `Tracer`. Ponadto każda instancja uzyskuje własny licznik odczytów dzięki generowaniu nowej instancji klasy `Wrapper`. Jak się przekonamy, tego rodzaju kodowanie jest trudniejsze, niż się na pozór wydaje.

## Stosowanie dekoratorów klas z wbudowanymi typami

Warto zauważyć, że poprzedni kod dekoruje klasę zdefiniowaną przez użytkownika. Tak jak w oryginalnym przykładzie z rozdziału 31., możemy także wykorzystać dekorator do opakowania typu wbudowanego, takiego jak lista, o ile albo wykorzystamy klasę podrzędną w celu umożliwienia składni dekoracji, albo dekorację wykonamy ręcznie — składnia dekoratora wymaga instrukcji `class` dla wiersza ze znakiem `@`. W poniższym kodzie zmienna `x` jest tak naprawdę znowu instancją klasy `Wrapper` z powodu pośredniego działania dekoracji:

```
>>> from interfacetracer import Tracer

>>> @Tracer
... class MyList(list): pass                # MyList = Tracer(MyList)

>>> x = MyList([1, 2, 3])                  # Wywołuje Wrapper()
>>> x.append(4)                            # Wywołuje __getattr__, append
Śledzenie: append
>>> x.wrapped
[1, 2, 3, 4]

>>> MyList = Tracer(list)                  # Lub ręczne wykonanie dekoracji
>>> x = MyList([4, 5, 6])                  # Inaczej wymagana instrukcja klasy podrzędnej
>>> x.append(7)
Śledzenie: append
>>> x.wrapped
[4, 5, 6, 7]
```

Rozwiązanie z dekoratorem pozwala nam na przeniesienie tworzenia instancji do samego dekoratora zamiast wymagania przekazania utworzonego wcześniej obiektu. Choć różnica wydaje się nieznaczna, pozwala nam to na zachowanie normalnej składni tworzenia instancji i ogólnie na skorzystanie ze wszystkich zalet dekoratorów. Zamiast wymagać, by wszystkie wywołania tworzące instancję ręcznie przekierowywały obiekty do obiektu pośredniczącego, wystarczy, że rozszerzymy klasy za pomocą składni dekoratorów:

```
@Tracer                                     # Rozwiązanie z dekoratorem
class Person: ...
bob = Person('Robert', 40, 50)
anna = Person('Anna', rate=100, hours=60)

class Person: ...                           # Rozwiązanie bez dekoratora
bob = Wrapper(Person('Robert', 40, 50))
anna = Wrapper(Person('Anna', rate=100, hours=60))
```

Zakładając, że będziemy tworzyć więcej niż jedną instancję klasy, dekoratory zazwyczaj będą lepszym rozwiązaniem z punktu widzenia zarówno wielkości kodu, jak i jego późniejszego utrzymywania.

## Pułapki związane z klasami — zachowanie większej liczby instancji

Co ciekawe, funkcję dekoratora z tego przykładu można by *prawie* zapisać w kodzie jako klasę, a nie funkcję, z odpowiednim protokołem przeciążania operatora. Przykład 39.15 to nieco uproszczona alternatywa, która działa podobnie, ponieważ jej metoda `__init__` wywoływana

jest, kiedy dekorator @ zostanie zastosowany do klasy, natomiast jej metoda `__call__` uruchamiana jest, kiedy tworzona jest instancja klasy podmiotowej. Nasze obiekty są tym razem tak naprawdę instancjami klasy `Tracer` i w gruncie rzeczy wymieniamy tutaj referencję do zakresu funkcji zawierającej na atrybut instancji.

Przykład 39.15. *interfacetracer-fail.py (start)*

```
class Tracer:
    def __init__(self, aClass):
        self.aClass = aClass
    def __call__(self, *args):
        self.wrapped = self.aClass(*args)
        return self
    def __getattr__(self, attrname):
        print('Śledzenie: ' + attrname)
        return getattr(self.wrapped, attrname)

@Tracer
class Hack:
    def display(self):
        print('Hakować!' * 3)

...
work = Hack()
work.display()
```

# W momencie dekoracji @  
# Użycie atrybutu instancji  
# W momencie tworzenia instancji  
# JEDNA (OSTATNIA) INSTANCJA NA KLASĘ!  
  
# Wywołuje \_\_init\_\_  
# Jak: Hack = Tracer(Hack)  
  
# Wywołuje \_\_call\_\_  
# Wywołuje \_\_getattr\_\_

Jak jednak widzieliśmy wcześniej, alternatywa z klasą obsługuje większą liczbę klas, podobnie jak poprzednie rozwiązanie, jednak w zasadzie nie działa dla *większej liczby instancji* określonej klasy — każde wywołanie tworzące instancję wywołuje metodę `__call__`, która nadpisuje poprzednią instancję. W rezultacie klasa `Tracer` zapisuje tylko jedną instancję — ostatnią utworzoną. Warto samodzielnie poeksperymentować, by się o tym przekonać, jednak poniżej znajduje się przykład 39.16, który pokazuje ten problem.

Przykład 39.16. *interfacetracer-fail.py (kontynuacja)*

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

bob = Person('Robert')
print(bob.name)
Anna = Person('Anna')
print(anna.name)
print(bob.name)
```

# Person = Tracer(Person)  
# Klasa Wrapper dowiązana do Person  
  
# Obiekt bob jest tak naprawdę instancją Wrapper  
# Wrapper osadza Person  
  
# Obiekt anna nadpisuje obiekt bob  
# UPS: teraz obiekt bob nazywa się 'Anna'!

Wynik powyższego kodu będzie następujący. Ponieważ ta klasa śledząca ma tylko jedną współdzieloną instancję, druga z nich nadpisuje pierwszą:

```
$ python3 interfacetracer-fail.py
Trace: display
Hakować!Hakować!Hakować!
Śledzenie: name
```

```
Robert
Śledzenie: name
Anna
Śledzenie: name
Anna
```

Problemem jest tutaj złe zachowanie *stanu*. Tworzymy jedną instancję dekoratora na klasę, ale nie na instancję klasy, przez co jedynie ostatnia instancja zostanie zachowana. Rozwiązaniem, jak w poprzednich rozważaniach na temat klas w przypadku dekoracji metod, jest porzucenie dekoratorów opartych na klasach.

Wersja klasy Tracer oparta na funkcji z przykładu 39.14 *działa* dla większej liczby instancji, ponieważ każde wywołanie tworzące instancję tworzy nową instancję klasy Wrapper, zamiast nadpisywać stan pojedynczej, współdzielonej instancji Tracer. Oryginalna wersja bez dekoratora również poprawnie obsługuje większą liczbę instancji — z tych samych przyczyn. Dekoratory są nie tylko nieco magiczne, ale także dość subtelne w swoim działaniu!

## Przykład: atrybuty „prywatne” i „publiczne”

Ostatnie dwie części niniejszego rozdziału prezentują większe przykłady zastosowania dekoratorów. Oba zawierają skróconą część opisową — po części dlatego, że rozdział ten już przekroczył przewidziany dla niego limit długości, ale również z powodu tego, że Czytelnik powinien już na tyle dobrze rozumieć podstawy dekoratorów, by potrafić przestudiować przykłady samodzielnie.

## Implementacja atrybutów prywatnych

Najpierw spójrzmy na przykład 39.17 z *dekoratorem klasy*, który implementuje deklarację `Private` i sprawdza dostęp do atrybutów instancji klasy. Oznacza to, że atrybuty są przechowywane w instancji lub dziedziczone po jednej z jej klas.

Przykład 39.17. *access1.py*

```
"""
Prywatność dla atrybutów pobranych z instancji klas.
Przykłady użycia można znaleźć w kodzie testu samosprawdzającego na dole pliku.
Dekorator jest tym samym co: Doubler = Private('data', 'size')(Doubler).
Private zwraca onDecorator, onDecorator zwraca onInstance, a każda instancja onInstance osadza instancję
Doubler.
"""

traceMe = False
def trace(*args):
    if traceMe: print(f'[{ ' '.join(map(str, args))}]') # Literal f-string od Pythona 3.12

def Private(*privates):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.wrapped = aClass(*args, **kargs)
        # privates w zakresie funkcji zawierającej
        # aClass w zakresie funkcji zawierającej
        # Opakowane w atrybut instancji
```

```

def __getattr__(self, attr):
    trace('pobranie:', attr)
    if attr in privates:
        raise TypeError('pobranie atrybutu prywatnego: ' + attr)
    else:
        return getattr(self.wrapped, attr)

def __setattr__(self, attr, value):
    trace('ustawienie:', attr, value)
    if attr == 'wrapped':
        self.__dict__[attr] = value
    elif attr in privates:
        raise TypeError('modyfikacja atrybutu prywatnego: ' + attr)
    else:
        setattr(self.wrapped, attr, value)
return onInstance
return onDecorator

if __name__ == '__main__':
    traceMe = True

    @Private('data', 'size')
    class Doubler:
        def __init__(self, label, start):
            self.label = label
            self.data = start

        def size(self):
            return len(self.data)

        def double(self):
            for i in range(self.size()):
                self.data[i] = self.data[i] * 2

        def display(self):
            print(f'{self.label} => {self.data}')

    print('Tworzenie instancji...')
    X = Doubler('X to', [1, 2, 3])
    Y = Doubler('Y to', [-10, -20, -30])

    # Poniższe testy kończą się powodzeniem

    print('\nBadanie instancji X...')
    print(X.label)
    X.display(); X.double(); X.display()

    print('\nBadanie instancji Y...')
    print(Y.label)
    Y.display(); Y.double()
    Y.label = 'Hakować'
    Y.display()

    # Poniższe testy wszystkie kończą się niepowodzeniem (zgodnie z zamierzeniami)
    """
    print(X.size())

    print(X.data)

```



```

X.data = [1, 1, 1]
X.size = lambda S: 0
print(Y.data)
print(Y.size())
"""
# Wyświetla "TypeError: modyfikacja atrybutu
# prywatnego: data"

```

Ten dekorator nie pozwala na pobieranie i modyfikowanie takich atrybutów *spoza* udekorowanej klasy, jednak nadal umożliwia samej klasie swobodny dostęp do tych zmiennych wewnątrz jej metod. Kod ten nie do końca jest tym samym co mechanizmy języków C++ czy Java, ale umożliwia podobną kontrolę dostępu jako element opcjonalny Pythona dla rzadkich i nietypowych przypadków.

W rozdziale 30. widzieliśmy już niepełną, wstępną implementację prywatności atrybutów instancji w przypadku *modyfikacji*. Wersja z tego rozdziału rozszerza tę koncepcję w taki sposób, by sprawdzać również próby *pobierania* atrybutu. Do implementacji tego modelu wykorzystuje także delegację zamiast dziedziczenia. Tak naprawdę w pewnym sensie jest to tylko rozszerzenie dekoratora klasy śledzącej atrybuty, z którym spotkaliśmy się wcześniej.

Choć przykład ten wykorzystuje do implementacji prywatności atrybutów nowość składniową, jaką są dekoratory klas, przechwytywanie atrybutów jest w nim w dużej mierze nadal oparte na metodach przeciążania operatorów `__getattr__` oraz `__setattr__`, z którymi spotkaliśmy się w poprzednich rozdziałach. Kiedy wykryta zostaje próba dostępu do atrybutu prywatnego, ta wersja kodu wykorzystuje instrukcję `raise` do zgłoszenia wyjątku wraz z komunikatem o błędzie. Wyjątek ten można przechwycić w instrukcji `try` lub pozwolić mu na zakończenie skryptu.

Przykład 39.17 przedstawia wstępną wersję kodu dekoratora, wraz z kodem samotestującym na końcu pliku. W obecnej postaci przechwytuje wszystkie jawne pobrania atrybutów, ale nie przechwytuje pobrań niejawnych w operacjach wbudowanych (więcej na ten temat za chwilę).

Kiedy `traceMe` zwraca `True`, kod testu samosprawdzającego pliku modułu zwraca następujące dane wyjściowe. Warto zwrócić uwagę na to, w jaki sposób dekorator przechwytuje i sprawdza zarówno próby pobierania, jak i przypisania wykonywane *poza* opakowaną klasą — jednak nie przechwytuje prób dostępu wykonywanych *wewnątrz* samej klasy:

```

$ python3 access1.py
Tworzenie instancji...
[set: wrapped <__main__.Doubler object at 0x00000000029769B0>]
[set: wrapped <__main__.Doubler object at 0x00000000029769E8>]

Badanie instancji X...
[pobranie: label]
X to
[pobranie: display]
X to => [1, 2, 3]
[pobranie: double]
[pobranie: display]
X to => [2, 4, 6]

Badanie instancji Y...
[pobranie: label]
Y to

```

```
[pobranie: display]
Y to => [-10, -20, -30]
[pobranie: double]
[ustawienie: label Hakować]
[pobranie: display]
Mielonka => [-20, -40, -60]
```

## Szczegóły implementacji I

Powyższy kod jest nieco skomplikowany i najlepiej będzie prześledzić go samodzielnie w celu przekonania się, jak działa. By pomóc w tej analizie, poniżej znajduje się kilka elementów, na które warto zwrócić uwagę.

### Dziedziczenie a delegacja

Pierwszy, wstępny przykład implementacji prywatności z rozdziału 30. wykorzystywał *dziedziczenie* do wmieszania metod `__setattr__` do przechwytywania prób dostępu. Dziedziczenie mocno to jednak utrudnia, ponieważ rozróżnienie pomiędzy dostępem z wewnątrz i z zewnątrz klasy nie jest tak oczywiste (dostęp z wewnątrz powinien móc się odbywać normalnie, natomiast dostęp z zewnątrz powinien być ograniczony). By to obejść, przykład z rozdziału 30. wymagał, aby dziedziczące klasy wykorzystywały przypisanie `__dict__` w celu ustawienia atrybutów — rozwiązanie to można w najlepszym razie nazwać niepełnym.

Wersja zaprezentowana powyżej wykorzystuje *delegację* (osadzenie jednego obiektu wewnątrz innego) zamiast dziedziczenia. Taki wzorec kodu lepiej nadaje się do naszego zadania, gdyż bardzo ułatwia rozróżnienie pomiędzy próbami dostępu z wewnątrz oraz z zewnątrz podmiotowej klasy. Dostęp do atrybutów spoza klasy podmiotowej jest przechwytywany przez metody przeciążania operatorów warstwy opakowującej i delegowany do klasy, jeśli jest poprawny. Próby dostępu wewnątrz samej klasy (na przykład za pośrednictwem `self` wewnątrz kodu metody) nie są przechwytywane i mogą być wykonywane normalnie bez sprawdzania, ponieważ prywatność nie jest tutaj dziedziczona.

### Argumenty dekoratora

Wykorzystany tutaj dekorator klasy przyjmuje dowolną liczbę argumentów nazywających atrybuty prywatne. Tak naprawdę argumenty przekazywane są do funkcji `Private`, natomiast `Private` zwraca funkcję dekoratora, która ma być zastosowana do podmiotowej klasy. Argumenty wykorzystywane są zatem przed wystąpieniem dekoracji. Funkcja `Private` zwraca dekorator, który z kolei „pamięta” listę atrybutów prywatnych w postaci referencji do zakresu funkcji zawierającej.

### Zachowywanie stanu i zakresy funkcji zawierającej

A skoro już mowa o zakresach funkcji zawierającej, tak naprawdę w powyższym kodzie zachowywanie stanu działa na *trzech poziomach*:

- Argumenty funkcji `Private` wykorzystywane są, zanim nastąpi dekoracja, i są zachowywane w postaci referencji do zakresu funkcji zawierającej — do wykorzystania w `onDecorator` oraz `onInstance`.
- Argument klasy dla `onDecorator` wykorzystywany jest w czasie dekoracji i zachowywany w postaci referencji do zakresu funkcji zawierającej — do wykorzystania w czasie tworzenia instancji.
- Obiekt opakowanej instancji zachowywany jest w postaci atrybutu instancji w `onInstance` — do wykorzystania, gdy w późniejszym czasie ma miejsce próba dostępu do atrybutów spoza klasy.

Wszystko to działa w miarę naturalnie w oparciu o reguły Pythona dotyczące zakresów oraz przestrzeni nazw.

## Wykorzystanie `__dict__` i `__slots__` (i innych nazw wirtualnych)

Metoda `__setattr__` z powyższego kodu, próbując ustawić własny opakowany atrybut w `onInstance`, opiera się na słowniku `__dict__` przestrzeni nazw atrybutów obiektu instancji. Jak wiemy z poprzedniego rozdziału, nie może ona przypisać atrybutu w sposób bezpośredni bez wykonania pętli. Wykorzystuje ona jednak funkcję wbudowaną `setattr` w miejsce `__dict__` w celu ustawienia atrybutów w samym *opakowanym* obiekcie. Co więcej, do pobrania atrybutów w opakowanym obiekcie wykorzystana zostaje metoda `getattr`, ponieważ mogą one być przechowane w samym obiekcie bądź odziedziczone przez niego.

Z tej przyczyny kod ten będzie działał dla większości klas, w tym z „wirtualnymi” atrybutami opartymi na *slotach*, *właściwościach*, *deskryptorach*, a nawet metodzie `__getattr__` i jej podobnych. Klasa opakowująca, dzięki temu, że rezerwuje słownik przestrzeni nazw tylko dla siebie i wykorzystuje niezależne narzędzia, jest pozbawiona ograniczeń typowych dla innych narzędzi.

Niektóre osoby mogą pamiętać z rozdziału 32., że klasy ze `__slots__` mogą nie przechowywać atrybutów w słowniku `__dict__` (a nawet mogą tych atrybutów nie mieć). Ponieważ jednak polegamy na `__dict__` jedynie na poziomie `onInstance`, a nie w opakowanej instancji, a także dlatego, że metody `setattr` oraz `getattr` mają zastosowanie do atrybutów opartych zarówno na `__dict__`, jak i na `__slots__`, nasz dekorator działa na klasach wykorzystujących dowolny z tych mechanizmów przechowywania.

Z tego samego powodu dekoratory można stosować z właściwościami oraz podobnymi narzędziami. Delegowane nazwy będą ponownie wyszukiwane w opakowanej instancji niezależnie od atrybutów samego pośrednika dekoratora.

## Uogólnienie kodu pod kątem deklaracji atrybutów jako publicznych

Skoro już mamy implementację prywatności, dość łatwo jest uogólnić ten kod w celu dopuszczenia także deklaracji publicznych — są one właściwie odwrotnością deklaracji prywatnych, dlatego wystarczy, że będą negować wewnętrzny test. Przykład 39.18. pozwala klasie wykorzystywać

dekoratory do zdefiniowania zbioru publicznych lub prywatnych atrybutów instancji (atrybutów przechowywanych w instancji lub odziedziczonych po jej klasach) za pomocą następującej semantyki:

- `Private` deklaruje atrybuty instancji klas, których *nie można* pobierać lub przypisywać, z wyjątkiem pochodzących z wewnątrz kodu metod klasy. Oznacza to, że nie jest możliwy dostęp z zewnątrz do żadnej zmiennej zadeklarowanej jako prywatna, natomiast wszystkie zmienne niezadeklarowane w ten sposób można swobodnie pobierać i przypisywać z zewnątrz.
- `Public` deklaruje atrybuty instancji klas, które *można* pobierać lub przypisywać zarówno z zewnątrz klasy, jak i z wewnątrz jej metod. Oznacza to, że dostęp do każdej zmiennej zadeklarowanej jako publiczna jest możliwy z dowolnego miejsca, natomiast dostęp z zewnątrz do zmiennych niezadeklarowanych w ten sposób nie jest możliwy.

Deklaracje `Private` oraz `Public` mają się wzajemnie wykluczać. Kiedy używamy `Private`, wszystkie niezadeklarowane zmienne uznawane są za publiczne, natomiast jeśli użyjemy `Public`, wszystkie niezadeklarowane zmienne uznawane są za prywatne. Są one swoimi przeciwieństwami, choć niezadeklarowane zmienne nieutworzone przez metody klas zachowują się nieco inaczej — mogą zostać przypisane i tym samym utworzone poza klasą pod `Private` (wszystkie niezadeklarowane zmienne są dostępne), jednak pod `Public` już nie (wszystkie niezadeklarowane zmienne są niedostępne).

Ponownie zachęcam do samodzielnego przestudiowania kodu w celu przekonania się, jak on działa. Warto zwrócić uwagę na to, że rozwiązanie to na górze dodaje dodatkowy, *czwarty poziom zachowywania stanu*, poza poziomami opisanymi w poprzednim podrozdziale. Funkcje sprawdzające wykorzystywane przez `lambda` są zapisywane w dodatkowym zakresie zawierającym. Przykład ten ma pewne ograniczenie, jak poprzedni przykład z atrybutami dla operacji wbudowanych (wyjaśnione pokrótce w łańcuchu znaków dokumentacji pliku i nieco szerzej w opisie następującym po kodzie).

Przykład 39.18. `access2.py`

"""

*Dekorator klasy z deklaracjami atrybutów jako prywatne i publiczne.*

*Kontroluje dostęp do atrybutów przechowywanych w instancji lub dziedziczonych przez nią po klasach. Private deklaruje nazwy atrybutów, których nie można pobrać lub przypisać z zewnątrz udekorowanej klasy. Public deklaruje nazwy atrybutów, które można pobrać lub przypisać z zewnątrz.*

*Uwaga: działa jedynie dla jawnie wymienionych atrybutów. Metody przeciążania operatorów `__X__` wykonywane w sposób niejawni dla operacji wbudowanych nie wywołują metod `__getattr__` ani `__getattribute__` w klasach i z tego powodu nie będą delegowane do żadnego opakowanego obiektu, który je definiuje. Jeśli to konieczne, należy tutaj dodać metody `__X__` w celu przechwycenia i wydelegowania nazw wbudowanych.*

"""

```
traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + '']')
```

```

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.__wrapped = aClass(*args, **kargs)

            def __getattr__(self, attr):
                trace('pobranie:', attr)
                if failIf(attr):
                    raise TypeError('pobranie atrybutu prywatnego: ' + attr)
                else:
                    return getattr(self.__wrapped, attr)
            def __setattr__(self, attr, value):
                trace('ustawienie:', attr, value)
                if attr == '_onInstance_wrapped':
                    self.__dict__[attr] = value
                elif failIf(attr):
                    raise TypeError('modyfikacja atrybutu prywatnego: ' + attr)
                else:
                    setattr(self.__wrapped, attr, value)
        return onInstance
    return onDecorator

def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))

def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))

```

Przykłady użycia można znaleźć w kodzie testu samosprawdzającego poprzedniego przykładu. Oto szybkie spojrzenie na działanie tych dekoratorów klas w sesji interaktywnej. Zgodnie z tym, co zapowiadaliśmy, nazwy nieprywatne lub publiczne można pobierać i modyfikować spoza podmiotowej klasy, natomiast nie można tego robić w przypadku nazw prywatnych lub niepublicznych:

```

>>> from access2 import Private, Public

>>> @Private('age')
... class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
# Person = Private('age')(Person)
# Person = onInstance ze stanem
# Próby dostępu z wewnątrz odbywają się normalnie

>>> X = Person('Robert', 40)
>>> X.name
'Robert'
>>> X.name = 'Anna'
>>> X.name
'Anna'
>>> X.age
TypeError: pobranie atrybutu prywatnego: age
>>> X.age = 'Tomasz'
TypeError: modyfikacja atrybutu prywatnego: age

>>> @Public('name')
... class Person:
    def __init__(self, name, age):

```

```

        self.name = name
        self.age = age

>>> X = Person('robert', 40)                                # X jest instancją onInstance
>>> X.name                                                    # onInstance osadza Person
'robert'
>>> X.name = 'Anna'
>>> X.name
'Anna'
>>> X.age
TypeError: pobranie atrybutu prywatnego: age
>>> X.age = 'Tomasz'
TypeError: modyfikacja atrybutu prywatnego: age

```

## Szczegóły implementacji II

Pomocne w analizie kodu z przykładu 39.18 będzie kilka zamieszczonych poniżej finalnych uwag dotyczących tej wersji. Ponieważ jest ona jedynie uogólnieniem przykładu z poprzedniego podrozdziału, większość uwag, które go dotyczą, ma zastosowanie również tutaj.

### Użycie nazw pseudoprywatnych `__X`

Poza uogólnieniem wersja ta wykorzystuje również opcję nazw pseudoprywatnych Pythona `__X` (z którą spotkaliśmy się w rozdziale 31.) do zlokalizowania atrybutu `wrapped` w klasie kontrolnej dzięki automatycznemu poprzedzeniu go nazwą klasy. Pozwala nam to uniknąć ryzyka konfliktów z atrybutem `wrapped`, który może być wykorzystywany przez prawdziwą, opakowaną klasę (co miało miejsce w poprzedniej wersji kodu) — może się to przydać w tak uniwersalnym narzędziu. Opcja ta nie zapewnia jednak prywatności, ponieważ taka nazwa może być swobodnie wykorzystywana poza klasą. Warto również zauważyć, że w metodzie `__setattr__` będziemy musieli użyć łańcucha znaków pełnej, rozszerzonej nazwy ('`__onInstance__wrapped`'), ponieważ na to zmienia ją Python.

### Złamanie prywatności

Choć powyższy przykład implementuje kontrolę dostępu dla atrybutów instancji i jej klas, można tę kontrolę na różne sposoby odwrócić — na przykład w sposób jawny przechodząc rozszerzoną wersję atrybutu `wrapped` (`bob.pay` może nie zadziałać, ale pełna nazwa `bob.__onInstance__wrapped.pay` już może!). Jeśli jednak musimy to robić w sposób jawny, poziom kontroli powinien być wystarczający na potrzeby normalnego, zamierzonego użycia. Oczywiście kontrolę prywatności można odwrócić w każdym języku programowania, o ile będziemy się wystarczająco mocno starać (w niektórych implementacjach C++ może także zadziałać `#define private public`). Choć kontrola dostępu może ograniczyć przypadkowe zmiany, w dowolnym języku programowania wiele zależy od programisty. W każdej sytuacji, w której kod źródłowy można zmienić, kontrola dostępu będzie mrzonką. Python ma na celu *dawanie możliwości*, a nie kontrolowanie. Prywatność to narzędzie, którego najlepiej używać oszczędnie (jeśli w ogóle).

## Kompromisy związane z dekoratorem

I znów, moglibyśmy uzyskać te same wyniki bez dekoratorów, wykorzystując funkcje zarządzające lub ręcznie pisząc kod ponownie dowiązujący nazwy dekoratorów. Składnia dekoratorów sprawia jednak, że całość jest nieco bardziej czywista i spójna. Największą potencjalną wadą tego rozwiązania (i wszystkich innych opartych na obiektach opakowujących) jest to, że dostęp do atrybutów wymaga dodatkowego wywołania, a instancje udekorowanych klas nie są tak naprawdę instancjami oryginalnej dekorowanej klasy. Jeśli na przykład sprawdzimy ich typ za pomocą `X.__class__` czy `isinstance(X, C)`, okaże się, że są one instancjami klasy *opakowującej*. O ile jednak nie planujemy wykonywać introspekcji na typach obiektów, kwestia ta jest najprawdopodobniej bez znaczenia, a dodatkowe wywołania będą miały miejsce głównie na etapie kodowania. Jak się przekonamy później, są sposoby automatycznego usuwania dekoratorów (poprzez `-0`), gdy zajdzie taka potrzeba.

## Delegowanie operacji wbudowanych

W obecnej postaci przykład ten działa zgodnie z zamierzeniami w przypadku metod *jawnie* wywoływanych za pomocą ich nazw. Jak to jednak ma miejsce w przypadku większości oprogramowania, zawsze coś można poprawić. Przede wszystkim to narzędzie obniża wydajność metod przeciążających operatory, jeżeli są one stosowane w klasach klienckich.

Dokładniej mówiąc, klasa pośrednicząca nie weryfikuje ani nie deleguje metod przeciążania operatorów pobieranych *niejawnie* przez operacje wbudowane, chyba że takie metody zostaną zdefiniowane ponownie w pośredniku. Klienci, które nie używają przeciążania operatorów, są w pełni obsługiwane, ale inne mogą wymagać dodatkowego kodu. Nie jest jasne, czy metody przeciążania operatorów *powinny* być sprawdzane pod kątem tego, czy są prywatne, czy publiczne, ale są one częścią interfejsu obiektu i powinny przynajmniej być kierowane do obiektów opakowanych, które je definiują.

Na ten problem natknęliśmy się już kilka razy w tej książce, przeanalizujemy ogólnie jego wpływ na bardzo praktyczny kod, który tutaj napisaliśmy, oraz zbadajmy możliwości jego rozwiązania. Podstawowy problem jest łatwy do zademonstrowania. Oto jak zwykle działa klasa przeciążająca wywołania `print` i wyrażenia `+`:

```
>>> class Tally:
    def __init__(self):
        self.sum = 0
    def __str__(self):
        return f'Tally: {self.sum}'
    def __add__(self, add):
        self.sum += add

>>> X = Tally()
>>> X.sum                                # Wszystkie atrybuty są dostępne
0
>>> print(X)                             # To samo co X.__str__()
Tally: 0
>>> X + 5                                # To samo co X.__add__(5)
>>> print(X)
Tally: 5
```

Niestety obiekty implementujące wbudowane operacje w ten sposób zawiodą w klasach pośredniczących, ponieważ operacje wbudowane omijają protokoły wyszukiwania na poziomie instancji, takie jak `__getattr__`, i zamiast tego przeszukują przestrzenie nazw klas:

```
>>> from access2 import Private
>>> @ Private('sum', '__add__')
... class Tally:
...     ...tak samo jak wcześniej...
>>> X = Tally()
>>> X.sum
TypeError: pobranie atrybutu prywatnego: sum

>>> X.__add__(5)
TypeError: pobranie atrybutu prywatnego: __add__

>>> print(X)
<access2.accessControl.<locals>.onDecorator.<locals>.onInstance object at 0x...etc...>

>>> X + 5
TypeError: unsupported operand type(s) for +: 'onInstance' and 'int'
```

W tym kodzie pierwsze dwa *jawne* pobrania `sum` i `__add__` są odrzucane jako prywatne, co jest zgodne z oczekiwaniami. Jednak ostatnie dwa *niejawne* pobrania, `print` i `+`, nie są przechwytywane przez kod pośredniczący, więc nigdy nie są delegowane do opakowanego obiektu `Tally`. Funkcja `print` działa tutaj tylko dlatego, że używa domyślnej metody obiektu do wydrukowania samego pośrednika. Według poprzedniego rozdziału jest to niekonsekwencja w Pythonie, a zgodnie z następnymi punktami można jej również w pełni uniknąć.

## Sposoby redefiniowania metod przeciążających operatory

Najprostszym rozwiązaniem obsługi operacji wbudowanych w pośredniczeniu delegacji jest powtórne zdefiniowanie nazw przeciążających operatory, które mogą się pojawić w obiektach osadzonych. Powoduje to pewną powtarzalność kodu, ale nie jest to również zbyt duży wysiłek programistyczny, można go do pewnego stopnia zautomatyzować za pomocą narzędzi lub klas nadrzędnych. Co więcej, można zdecydować się na przeprowadzenie lub pominięcie walidacji dla nazw przeciążających operatorów zadeklarowanych jako prywatne lub publiczne, w zależności od trasowania redefinicji.

Część listingu z przykładu 39.19 wykorzystuje definicję *śródwierszową*. Przechwytuje i deleguje wbudowane operacje, dodając definicje metod do samego kodu pośredniczącego dla każdej metody przeciążającej operator, którą może zdefiniować opakowany obiekt. Dla celów ilustracyjnych zostały dodane cztery metody przechwytyjące operatory. W przypadku innych operatorów należy postąpić w taki sam sposób (nowy kod jest wyróżniony pogrubioną czcionką, a wszystkie przykłady opierają się na dekoratorze z pliku *access2.py* w przykładzie 39.18).

Przykład 39.19. *access\_builtins\_inline\_direct.py*

```
"Metody śródwierszowe, pomijanie walidacji"

def accessControl(failIf):
    def onDecorator(aClass):
```



```

class onInstance:
    def __init__(self, *args, **kargs):
        self.__wrapped = aClass(*args, **kargs)

    # Przechwycenie i wydelegowanie metod przeciążających operatory

    def __add__(self, other):
        return self.__wrapped + other          # Lub getattr(), __getattr__()
    def __str__(self):
        return str(self.__wrapped)             # Lub self.__wrapped.__str__()
    def __getitem__(self, index):
        return self.__wrapped[index]
    def __call__(self, *args, **kargs):
        return self.__wrapped(*arg, *kargs)
    #...i ewentualne inne potrzebne metody...

    # Przechwycenie i wydelegowanie nazwanych atrybutów

    def __getattr__(self, attr): ...to samo...
    def __setattr__(self, attr, value): ...to samo...
    return onInstance
return onDecorator

```

To działa, ponieważ wbudowane operacje znajdują wymagane metody w *klasie* pośredniczącej po pominięciu instancji pośredniczącej. W obecnej postaci nowe metody przechwytyjące *bezpośrednio* wywołują metody przeciążające operatory opakowanego obiektu, omijając tym samym kontrolę dostępu metody `__getattr__`, co może być pożądane lub nie. Przejdźmy teraz do alternatywnych rozwiązań kodowania.

## Kodowanie metod przeciążających operatory w klasach nadrzędnych

Alternatywnym rozwiązaniem jest wstawienie powyższych metod za pomocą wspólnej *klasy nadrzędnej*. Jeżeli takich metod jest kilkanaście, wtedy lepiej do tego zadania nadaje się zewnętrzna klasa, szczególnie jeżeli jest na tyle uniwersalna, że można ją wykorzystywać w dowolnym interfejsie klasy pośredniczącej.

Aby to zademonstrować, klasa nadrzędna z przykładu 39.20 przechwytyuje wbudowane operacje i ponownie kieruje je *bezpośrednio* do opakowanego obiektu. To w dużej mierze tylko przeorganizowanie poprzedniego schematu wewnętrznego, ale jako osobna klasa wymaga pośredniczącego atrybutu o nazwie `__wrapped`, który daje dostęp do osadzonego obiektu. Sam dekorator musi używać tej nazwy zamiast `__wrapped` w odwołaniach do `self` i bez zamiany nazwy w `__setattr__`. Może to być mniej optymalne, ponieważ wyklucza tę samą nazwę w opakowanych obiektach i tworzy zależność klasy podrzędnej, ale jest lepsze niż użycie zamienionej i specyficznej dla podklasy nazwy `_onInstance__wrapped` i nie jest gorsze niż podobnie nazwana metoda.

Przykład 39.20. *access\_builtins\_mixin\_direct.py*

"Dziedziczenie metod, pomijanie sprawdzania"

```

class BuiltinsMixin:
    def __add__(self, other):

```

```

        return self._wrapped + other
def __str__(self):
    return str(self._wrapped)
def __getitem__(self, index):
    return self._wrapped[index]
def __call__(self, *args, **kwargs):
    return self._wrapped(*args, **kwargs)
    # ...i wiele innych potrzebnych operacji

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance(BuiltinsMixin):
            ...Należy użyć self._wrapped zamiast self.__wrapped...

```

Alternatywnie klasa nadrzędna w przykładzie 39.21 przechwytyuje wbudowane operacje i przekierowuje je przez `__getattr__` w podklasie, aby zastosować kontrole dostępu do nazwy metody operacji. Wymaga, aby nazwy przeciążające operatory były oznaczone jako nieprywatne lub publiczne zgodnie z argumentami dekoratora, jeśli mają być uruchamiane. Traktuje niejawne pobrania wbudowanych operacji tak samo jak jawne pobrania nazw i nie wymaga `_wrapped` w klasach podrzędnych.

Przykład 39.21. *access\_builtins\_mixin\_getattr.py*

```

"Dziedziczenie metod, uruchomienie walidacji"

class BuiltinsMixin:
    def __add__(self, other):
        return self._getattr('__add__')(other)
    def __str__(self):
        return self._getattr('__str__')()
    def __getitem__(self, index):
        return self._getattr('__getitem__')(index)
    def __call__(self, *args, **kwargs):
        return self._getattr('__call__')(*args, **kwargs)
    # ...i wiele innych potrzebnych operacji

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance(BuiltinsMixin):
            ...reszta bez zmian...

```

Podobnie jak podejścia śródwierszowego, obie te klasy mieszane również wymagają jednej metody na każdą wbudowaną operację w ogólnych narzędziach, które pośredniczą dla dowolnych interfejsów obiektów. Następny pomysł działa nieco lepiej.

## Generowanie deskryptorów przeciążających operatory

Wszystkie podejścia dla wbudowanych operacji, które widzieliśmy do tej pory, kodują każdą metodę przeciążającą operator jawnie i przechwytyują rzeczywiste wywołanie wydane dla operacji, łącznie z jej argumentami. To sprawia, że są odpowiedzialne za dokończenie operacji, czy to przez składnię operacji, czy równoważne wywołania.

W alternatywnym podejściu moglibyśmy jedynie przechwytywać *pobranie* atrybutu poprzez dzające wywołanie, korzystając z *deskryptorów* na poziomie klasy z poprzedniego rozdziału. Co więcej, ponieważ wszystkie takie deskryptory będą działać tak samo, można je generować automatycznie z listy nazw metod. Przykład 39.22 pokazuje jeden ze sposobów zakodowania tego schematu. Podobnie jak w przykładzie 39.21, wbudowane operacje są kierowane do logiki walidacji dekoratora, aby *sprawdzać*, czy deskryptor jest prywatny, czy publiczny.

Przykład 39.22. *access\_builtins\_mixin\_desc.py*

"Dziedziczenie deskryptorów, uruchamianie walidacji"

```
class BuiltinsMixin:
    class ProxyDesc:                                # Definicja deskryptora
        def __init__(self, attrname):
            self.attrname = attrname
        def __get__(self, instance, owner):
            return instance.__getattr__(self.attrname)    # Uruchomienie walidacji

    builtins = ['add', 'str', 'getitem', 'call']        # I inne metody
    for attr in builtins:
        exec(f'__{attr}__ = ProxyDesc("__{attr}__")')    # Utworzenie deskryptorów

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance(BuiltinsMixin):                # Dziedziczenie deskryptorów
            ...reszta bez zmian...
```

Powyższy kod jest prawdopodobnie najbardziej zwięzły, ale też najbardziej niejawny i skomplikowany. Przypomnij sobie, że wbudowana funkcja `exec` domyślnie wykonuje ciąg kodu, jak gdyby był on wklejony w miejscu, gdzie pojawia się `exec`. Zatem pętla na końcu klasy mieszanej jest odpowiednikiem poniższego kodu. Działa w lokalnym zakresie klasy mieszanej:

```
__add__ = ProxyDesc("__add__")
__str__ = ProxyDesc("__str__")
...itd...
```

W rezultacie tworzone są deskryptory odpowiadające na pierwsze wyszukiwania nazw poprzez pobieranie z nowego obiektu opakowanego w `__get__`, a nie samodzielne przechwytywanie wykonywanych później operacji. Jeśli nadal nie rozumiesz tego kodu (i prawdopodobnie tak jest), to wiedz, że jest on odpowiednikiem tej uproszczonej wersji, chociaż pobranie atrybutu `name` następuje niejawnie w operacji wbudowanej, która omija protokoły instancji:

```
>>> class B:
    class D:
        def __get__(s, i, o): return i.meth()
        name = D()

>>> class A(B):
    def meth(self): return 'hakować'

>>> I = A()
>>> I.name
'hakować'
```

W tym schemacie moglibyśmy również *pominąć* walidację dekoratora dla operacji wbudowanych, kierując pobranie atrybutów bezpośrednio do opakowanego obiektu — chociaż wymaga to dostępnego atrybutu `_wrapped` w dekoratorze, podobnie jak w przykładzie 39.20:

```
class ProxyDesc:
    ...
    def __get__(self, instance, owner):
        return getattr(instance._wrapped, self.attrname)    # Zakładając, że jest _wrapped
```

Ostatecznie wszystkie te podejścia sprawiają, że klasy, które przeciążają wbudowane operacje, działają poprawnie z dekoratorami prywatnymi i publicznymi — oraz innymi dekoratorami opartymi na delegowaniu, podobnymi do nich:

```
>>> from access_builtins_mixin_desc import Private

>>> @Private('sum')
... class Tally:
    def __init__(self):
        self.sum = 0
    def __str__(self):
        return f'Tally: {self.sum}'
    def __add__(self, add):
        self.sum += add

>>> X = Tally()
>>> X.sum                                     # Sprawdzane jawnie
TypeError: pobranie atrybutu prywatnego: sum
>>> X + 10                                   # Delegowana operacja wbudowana
>>> print(X)                                # Delegowana operacja wbudowana
Tally: 10
```

Publiczne (nieprywatne) operacje wbudowane są teraz delegowane i działają, natomiast prywatne operacje wbudowane są weryfikowane i anulowane:

```
>>> @Private('sum', '__add__')
... class Tally:
    tak samo jak wcześniej...

>>> X = Tally()
>>> X.sum                                     # Sprawdzane jawnie
TypeError: pobranie atrybutu prywatnego: sum
>>> X + 10                                   # Anulowana operacja wbudowana — prywatna
TypeError: pobranie atrybutu prywatnego: __add__
>>> print(X)                                # Dopuszczona operacja wbudowana — publiczna
Tally: 0

>>> @Private('__str__')
... class Tally:
    tak samo jak wcześniej...

>>> print(Tally())                           # Anulowana operacja wbudowana — prywatna
TypeError: pobranie atrybutu prywatnego: __str__
```

Jeśli chcesz dalej eksperymentować z przykładami z tego punktu, zobacz przykłady w materiałach do pobrania dla tej książki, gdzie znajdziesz pełny kod, a także jego kompleksowy skrypt testowy `access_builtins_TEST.py` i wyniki. Teraz czas przejść do następnego i ostatniego studium przypadku dekoratorów w tym rozdziale.

## Przykład: sprawdzanie poprawności argumentów funkcji

W ostatnim przykładzie użyteczności dekoratorów w niniejszym podrozdziale napiszemy *dekorator funkcji*, który automatycznie sprawdza, czy argumenty przekazane do funkcji bądź metody mieszczą się w określonym przedziale liczbowym. Został on zaprojektowany w celu użycia na etapie programowania lub produkcji i można go wykorzystać jako szablon dla innych, podobnych zadań (na przykład sprawdzania typów argumentów, jeśli musimy je wykonywać). Kod z tego przykładu będzie w dużej mierze materiałem do samodzielnego przestudiowania, z ograniczoną częścią opisową. Jak zwykle więcej szczegółów można znaleźć, przeglądając sam kod.

### Cel

W omówieniu programowania zorientowanego obiektowo w rozdziale 28. napisaliśmy klasę, która przyznawała podwyżkę obiektom reprezentującym ludzi w oparciu o przekazaną wartość procentową:

```
class Person:
    ...
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

Zauważyliśmy tam, że gdybyśmy chcieli, by kod był bardziej rozbudowany, niezłym pomysłem byłoby sprawdzenie, czy wartość procentowa nie jest zbyt duża lub zbyt mała. Moglibyśmy wprowadzić tego typu sprawdzanie za pomocą instrukcji `if` lub `assert` *wstawionych* do samej metody:

```
class Person:
    def giveRaise(self, percent):
        if percent < 0.0 or percent > 1.0:
            raise TypeError, 'niepoprawna wartość procentowa'
        self.pay = int(self.pay * (1 + percent))

class Person:
    def giveRaise(self, percent):
        assert percent >= 0.0 and percent <= 1.0, 'niepoprawna wartość procentowa'
        self.pay = int(self.pay * (1 + percent))
```

Takie rozwiązanie zanieczyszcza jednak metodę testami wewnętrznymi, które najprawdopodobniej przydadzą się jedynie w trakcie pisania programu. W bardziej skomplikowanych przypadkach może się to stać dość żmudne (wyobraźmy sobie próby wstawiania kodu potrzebnego do zaimplementowania prywatności atrybutów z dekoratora z poprzedniego podrozdziału). Co jednak gorsze, jeśli logika sprawdzająca będzie się musiała kiedykolwiek zmienić, potencjalnie będziemy musieli odnaleźć i uaktualnić dowolnie dużą liczbę wierszy.

Ciekawszą i bardziej przydatną alternatywą byłoby napisanie narzędzia ogólnego przeznaczenia, które jest w stanie automatycznie wykonywać dla nas testy przedziałów dla argumentów dowolnej funkcji bądź metody, jaką utworzymy teraz lub w przyszłości. Rozwiązanie z *dekoratorem*

sprawia, że będzie się to odbywało w sposób spójny i jawny i będzie można to łatwo wyłączyć po zakończeniu tworzenia rozwiązania:

```
class Person:
    @rangetest(percent=(0.0, 1.0))           # Użycie dekoratora do sprawdzania
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

Wyizolowanie logiki sprawdzającej poprawność w dekoratorze upraszcza zarówno kod klienta, jak i późniejsze utrzymywanie.

Warto zauważyć, że nasz cel jest tutaj inny od sytuacji ze sprawdzaniem poprawności atrybutów z ostatniego przykładu z poprzedniego rozdziału. Tutaj chcemy sprawdzić wartości *argumentów funkcji* przy przekazywaniu, a nie *wartości atrybutów* przy ustawianiu. Dekorator Pythona wraz z narzędziami do introspekcji pozwolą nam wdrożyć to nowe zadanie równie łatwo.

## Prosty dekorator sprawdzający przedziały dla argumentów pozycyjnych

Zacznijmy od prostej implementacji testu przedziału. Dla uproszczenia zaczniemy od napisania kodu dekoratora, który działa jedynie dla argumentów *pozycyjnych* i zakłada, że zawsze, w każdym wywołaniu pojawiają się one na tej samej pozycji. Nie mogą być przekazane po nazwie słowa kluczowego, ponieważ może to zmienić pozycje zadeklarowane w dekoratorze. Przykład 39.23 to nasz wstępny kod.

Przykład 39.23. *rangetest0.py*

```
def rangetest(*argchecks):
    def onDecorator(func):
        if not __debug__:
            return func
        else:
            def onCall(*args):
                for (ix, low, high) in argchecks:
                    if args[ix] < low or args[ix] > high:
                        errmsg = f'Argument {ix} nie mieści się w przedziale {low}..{high}'
                        raise TypeError(errmsg)
                return func(*args)
            return onCall
    return onDecorator
```

W obecnej postaci powyższy kod jest właściwie powtórzeniem omówionych wcześniej wzorców kodu. Wykorzystujemy między innymi *argumenty* dekoratora i zagnieżdżone *zakresy* na potrzeby zachowywania stanu.

Używamy również zagnieżdżonych instrukcji `def`, by przykład działał zarówno dla prostych funkcji, jak i dla *metod*, zgodnie z tym, czego nauczyliśmy się wcześniej. Po wykorzystaniu dla metody klasy `onCall` otrzymuje instancję klasy podmiotowej w pierwszym elemencie `*args` i przekazuje ją dalej do `self` w oryginalnej funkcji metody. Jawnie przekazywane numery argumentów zakodowane w linii z `@` dekoratora zaczynają się od 1 w tym przypadku, a nie od 0, aby uwzględnić niejawnego `self`.

Warto również zauważyć, że kod ten wykorzystuje wbudowaną zmienną `__debug__` wprowadzoną w rozdziale 34. Python ustawia ją na `True`, o ile nie wykonujemy kodu z ustawioną opcją wiersza poleceń `-O` oznaczającą optymalizację (na przykład `python -O main.py`). Jak omówiono wcześniej, użycie opcji we wbudowanej funkcji `compile` oraz w module standardowej biblioteki `compileall` przed uruchomieniem kodu może mieć podobny efekt.

Kiedy zmienna `__debug__` zwraca `False`, dekorator zwraca oryginalną funkcję bez zmian w celu uniknięcia dodatkowych wywołań i związanego z tym negatywnego wpływu na wydajność. Innymi słowy, w przypadku użycia argumentu `-O` dekorator automatycznie *usuwa* modyfikujący kod, dzięki czemu nie trzeba tego robić ręcznie.

Przykład 39.24 pokazuje, jak będzie wykorzystywana pierwsza wersja naszego rozwiązania.

Przykład 39.24. *rangetest0\_test.py*

```
from rangetest0 import rangetest
print(f'__debug__={}')                                # False, jeśli "python -O main.py"

@rangetest((1, 0, 120))                                # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):                                # age musi się mieścić w przedziale 0..120
    print(f'{name} ma {age} lat')

@rangetest([0, 1, 31], [1, 1, 12], [2, 0, 2009])
def birthday(D, M, Y):
    print(f'Data urodzenia = {M}/{D}/{Y}')

class Person:
    def __init__(self, name, job, pay):
        self.job = job
        self.pay = pay

    @rangetest([1, 0.0, 1.0])                            # giveRaise = rangetest(...)(giveRaise)
    def giveRaise(self, percent):                        # Argument 0 to tutaj instancja self
        self.pay = int(self.pay * (1 + percent))

# Wiersze z komentarzem zgłaszają błąd TypeError, o ile w wierszu poleceń powłoki nie wykorzystano "python -O"

persinfo('Robert Zielony', 45)                        # Tak naprawdę wykonuje onCall(...) ze stanem
#persinfo('Robert Zielony', 200)                      # Lub person, jeśli ustawiono argument wiersza poleceń -O

birthday(31, 8, 2024)
#birthday(32, 8, 2024)

anna = Person('Anna Czerwona', 'programista', 100000)
anna.giveRaise(.10)                                    # Tak naprawdę wykonuje onCall(self, .10)
print(anna.pay)                                        # Lub giveRaise(self, .10), jeśli z -O
#anna.giveRaise(1.10)
```

Po wykonaniu poprawne wywołania w powyższym kodzie zwracają następujące wyniki:

```
$ python3 rangetest0_test.py
__debug__=True
Robert Zielony ma 45 lat
Data urodzenia = 31/8/2024
110000
```

Usunięcie komentarzy z któregoś z niepoprawnych wywołań powoduje zgłoszenie błędu `TypeError` przez dekorator. Oto wynik po pozwoleniu na wykonanie dwóch ostatnich wierszy (jak zwykle pominąłem część tekstu komunikatu o błędzie w celu zaoszczędzenia miejsca):

```
$ python3 rangetest0_test.py
__debug__ = True
Robert Zielony ma 45 lat
Data urodzenia = 31/8/2024
110000
TypeError: Argument 1 nie mieści się w przedziale 0.0..1.0
```

Uruchomienie Pythona z opcją wiersza poleceń `-0` wyłączy sprawdzanie przedziałów, jednak pozwoli także uniknąć pogorszenia wydajności związanego z warstwą opakującą. W rezultacie wywołujemy oryginalną, nieudekorowaną funkcję w sposób bezpośredni. Zakładając, że narzędzie to służy nam tylko do debugowania, możemy wykorzystać tę opcję do zoptymalizowania programu przed użyciem go w środowisku produkcyjnym. Oto efekt z ostatnim wierszem i dodanym `print`, aby pokazać fantastyczną podwyżkę pensji Anny:

```
$ python3 -0 rangetest0_test.py
__debug__ = False
Robert Zielony ma 45 lat
Data urodzenia = 31/8/2024
110000
231000
```

## Uogólnienie kodu pod kątem słów kluczowych i wartości domyślnych

Poprzednia wersja kodu ilustruje podstawy, z których musimy skorzystać, jednak jest stosunkowo ograniczona. Obsługuje jedynie sprawdzanie poprawności argumentów przekazanych po pozycji i nie sprawdza argumentów ze słowami kluczowymi. Tak naprawdę zakłada, że żadne słowa kluczowe nie będą przekazywane w sposób zakłócający pozycję argumentów. Dodatkowo nie robi nic z argumentami z wartościami domyślnymi, które można pominąć w wywołaniu. Takie rozwiązanie będzie w porządku, jeśli wszystkie argumenty przekazywane są po pozycji i nigdy nie mają wartości domyślnych, ale jest to dalekie od ideału w przypadku narzędzia ogólnego przeznaczenia. Jak wiesz z rozdziału 18., Python obsługuje o wiele bardziej elastyczne tryby przekazywania argumentów, które nasz kod na razie pomija.

Odmiana naszego kodu zaprezentowana w przykładzie 39.25 radzi sobie nieco lepiej. Dopasowując oczekiwane argumenty opakowanej funkcji do prawdziwych argumentów przekazanych w wywołaniu, pozwala obsługiwać sprawdzanie poprawności przedziałów dla argumentów przekazywanych albo po pozycji, albo za pomocą słowa kluczowego. Pomijane jest testowanie argumentów domyślnych pominiętych w wywołaniu. Mówiąc w skrócie, argumenty do sprawdzenia są określane przez słowa kluczowe dla dekoratora, który później przechodzi zarówno krotkę pozycyjną `*pargs`, jak i słownik słów kluczowych `**kargs` w celu sprawdzenia poprawności.

Przykład 39.25. *rangetest.py*

```
"""
Dekorator funkcji wykonujący sprawdzanie poprawności przedziałów dla przekazanych argumentów
do dowolnej funkcji lub metody. Podsumowanie użycia:
```



```

@rangetest(percent=(0.0, 1.0), month=(1, 12))
def func-or-method(..., percent, ..., month=5, ...):
    ...
func-or-method(..., value, month=8, ...)

```

Argumenty dla dekoratora określone są po słowach kluczowych. W samym wywołaniu argumenty mogą być przekazywane po pozycji lub za pomocą słowa kluczowego. Wartości domyślne mogą być pomijane. Przykładowe przypadki użycia znajdują się w pliku `rangetest_test.py`.

```

"""
trace = True

def rangetest(**argchecks):
    # Sprawdzenie poprawności przedziałów dla obu
    # oraz wartości domyślnych
    def onDecorator(func):
        # onCall pamięta func oraz argchecks
        if not __debug__:
            # True, jeśli ustawiono "python -O main.py args..."
            return func
        # Opakowanie przy debugowaniu; inaczej użycie oryginału
        else:
            funcname = func.__name__
            funccode = func.__code__
            funcargs = funccode.co_varnames[:funccode.co_argcount]

            def onCall(*pargs, **kargs):
                # Wszystkie argumenty pozycyjne pargs dopasowują pierwsze N oczekiwanych argumentów po pozycji
                # Reszta musi być w kargs lub jest pomijanymi wartościami domyślnymi

                positionals = funcargs[:len(pargs)]
                errmsg = lambda *args: '%s Argument "%s" nie mieści się w przedziale'
                %s...' % args

                for (argname, (low, high)) in argchecks.items():
                    # Dla wszystkich argumentów, które mają być sprawdzone
                    if argname in kargs:
                        # Przekazane po nazwie
                        if kargs[argname] < low or kargs[argname] > high:
                            raise TypeError(errmsg(funcname, argname, low, high))

                    elif argname in positionals:
                        # Przekazane po pozycji
                        position = positionals.index(argname)
                        if pargs[position] < low or pargs[position] > high:
                            raise TypeError(errmsg(funcname, argname, low, high))

                    else:
                        # Założenie, że argument nie został przekazany (domyślny)
                        if trace:
                            print(f'-Argument "{argname}" ma wartość domyślną')

                return func(*pargs, **kargs)
            # OK: wykonanie oryginalnego wywołania
            return onCall
        return onDecorator

```

Skrypt testowy z przykładu 39.26 pokazuje, w jaki sposób wykorzystywany jest dekorator. Argumenty do sprawdzenia przekazywane są jako argumenty dekoratora ze słowami kluczowymi, natomiast w samym wywołaniu możemy przekazać je albo po nazwie, albo po pozycji i pominąć argumenty z wartościami domyślnymi, nawet jeśli mają one być sprawdzane w inny sposób.

### Przykład 39.26. *rangetest\_test.py*

```

"""
Test dekoratora rangetest (użycie różni się od rangetest0)
Wiersze z komentarzami zgłaszają błąd TypeError, o ile nie ustawiono "python -O" w wierszu poleceń powłoki
"""

from rangetest import rangetest
def announce(what): print(what.center(24, '-')) # Metoda str

# Testowanie funkcji, pozycyjne i po słowach kluczowych
announce('Funkcje')

@rangetest(age=(0, 120))                                     # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):
    print(f'{name} ma {age} lat')

@rangetest(D=(1, 31), M=(1, 12), Y=(0, 2004))
def birthday(D, M, Y):
    print(f'urodziny = {D}/{M}/{Y}')

persinfo('Robert', 40)
persinfo(age=40, name='Robert')
birthday(1, M=5, Y=2025)
#persinfo('Robert', 150)
#persinfo(age=150, name='Robert')
#birthday(40, M=5, Y=2025)

# Testowanie metod, pozycyjne i po słowach kluczowych
announce('Metody')

class Person:
    def __init__(self, name, job, pay):
        self.job = job
        self.pay = pay

    @rangetest(percent=(0.0, 1.0))                             # giveRaise = rangetest(...)(giveRaise)
    def giveRaise(self, percent):                             # Wartość percent przekazana po nazwie lub pozycji
        self.pay = int(self.pay * (1 + percent))

bob = Person('Robert Zielony', 'programista', 100000)
anna = Person('Anna Czerwona', 'programista', 100000)
bob.giveRaise(.10)
anna.giveRaise(percent=.20)
print(f'anna=>{anna.pay}, bob=>{bob.pay}')
#anna.giveRaise(1.20)
#bob.giveRaise(percent=1.20)

# Testowanie pominiętych wartości domyślnych: pominięte
announce('Domyślne')

@rangetest(a=(1, 10), b=(1, 10), c=(1, 10), d=(1, 10))
def omitargs(a, b=7, c=8, d=9):
    print(a, b, c, d)

omitargs(1, 2, 3, 4)                                         # Pozycyjne
omitargs(1, 2, 3)                                           # Domyślne d

```

```

omitargs(1, 2, 3, d=4)           # Argument ze słowem kluczowym — d
omitargs(1, d=4)                 # Domyślne b i c
omitargs(d=4, a=1)               # Jak wyżej
omitargs(1, b=2, d=4)            # Domyślne c
omitargs(d=8, c=7, a=1)          # Domyślne d

#omitargs(1, 2, 3, 11)           # Niepoprawne d
#omitargs(1, 2, 11)              # Niepoprawne c
#omitargs(1, 2, 3, d=11)         # Niepoprawne d
#omitargs(11, d=4)               # Niepoprawne a
#omitargs(d=4, a=11)             # Niepoprawne a
#omitargs(1, b=11, d=4)          # Niepoprawne b
#omitargs(d=8, c=7, a=11)        # Niepoprawne a

```

Po wykonaniu powyższego skryptu argumenty niemieszczące się w przedziałach tak jak wcześniej zgłaszają wyjątki, jednak możemy je przekazywać po nazwie bądź pozycji, a pominięte wartości domyślne nie są sprawdzane. Warto prześledzić dane wyjściowe i kontynuować testy na własną rękę w celach eksperymentalnych. Kod działa jak wcześniej, jednak jego zakres został rozszerzony:

```

$ python3 rangetest_test.py
Robert ma 40 lat
Robert ma 40 lat
urodziny = 31/8/2024
-----Metody-----
anna=>120000, bob=>110000
-----Domyślne-----
1 2 3 4
-Argument "d" ma wartość domyślną
1 2 3 9
1 2 3 4
-Argument "b" ma wartość domyślną
-Argument "c" ma wartość domyślną
1 7 8 4
-Argument "b" ma wartość domyślną
-Argument "c" ma wartość domyślną
1 7 8 4
-Argument "c" ma wartość domyślną
1 2 8 4
Argument "b" ma wartość domyślną
1 7 7 8

```

Zauważ, że sprawdzanie argumentów odbywa się w *kolejności*, w jakiej zostały wymienione w dekoratorze, ponieważ Python zachowuje kolejność wstawiania w słownikach. W przypadku błędów walidacji otrzymujemy wyjątek, jak wcześniej, chyba że do Pythona zostanie przekazany argument `-0` wiersza poleceń, aby wyłączyć logikę dekoratora. Oto sytuacja, gdy znacznik komentarza zostanie usunięty sprzed jednego z wierszy testującego kodu:

```

$ python3 rangetest_test.py
-----Funkcje-----
Robert ma 40 lat
Robert ma 40 lat
urodziny = 31/8/2024
-----Metody-----
anna=>120000, bob=>110000

```

```
TypeError: giveRaise argument "percent" not in 0.0..1.0
```

```
$ python3 -0 rangetest_test.py
```

...brak komunikatów o błędach lub domyślnych śladów...

## Szczegóły implementacji

Kod tego dekoratora opiera się zarówno na API introspekcji, jak i subtelnych ograniczeniach przekazywania argumentów. By go w pełni uogólnić, powinniśmy spróbować naśladować pełną logikę dopasowywania argumentów Pythona w celu sprawdzenia, które nazwy zostaną przekazane w których trybach, jednak jest to zbyt wysoki stopień skomplikowania jak na nasze narzędzie. Lepiej byłoby, gdybyśmy mogli jakoś argumenty przekazane po nazwie dopasować do zbioru wszystkich oczekiwanych nazw argumentów w celu ustalenia, na jakiej pozycji pojawiają się one w określonym wywołaniu.

## Dalsza introspekcja

Okazuje się, że API do introspekcji dostępne dla obiektów funkcji oraz powiązanych z nimi obiektów kodu ma do dyspozycji narzędzie, którego potrzebujemy. API to zostało krótko wprowadzone w rozdziale 19., jednak teraz możemy je wykorzystać. Zbiór oczekiwanych *nazw argumentów* to po prostu pierwszych *N* nazw zmiennych dołączonych do obiektu kodu funkcji:

```
>>> def func(a, b, c, e=True, f=None):      # Trzy wymagane i dwa domyślne argumenty
    x = 1                                   # I jeszcze dwie zmienne lokalne
    y = 2

>>> code = func.__code__                   # Obiekt kodu obiektu funkcji
>>> code.co_nlocals
7
>>> code.co_varnames                       # Nazwy wszystkich zmiennych lokalnych
('a', 'b', 'c', 'e', 'f', 'x', 'y')
>>> code.co_varnames[:code.co_argcount]    # Pierwsze N zmiennych lokalnych
                                           # to oczekiwane argumenty
('a', 'b', 'c', 'e', 'f')
```

Jak zwykle: za pomocą argumentów, które w obiekcie pośredniczącym są opatrzone *gwiazdkami*, można odczytać dowolną liczbę argumentów i dopasować je do oczekiwanych argumentów:

```
>>> def catcher(*pargs, **kargs): print('%s, %s' % (pargs, kargs))

>>> catcher(1, 2, 3, 4, 5)
(1, 2, 3, 4, 5), {}
>>> catcher(1, 2, c=3, d=4, e=5)          # Argumenty wywoływanej funkcji
(1, 2), {'d': 4, 'e': 5, 'c': 3}
```

Wywołaj `dir` z nazwą funkcji i obiektami kodu, by uzyskać więcej informacji.

## Założenia dotyczące argumentów

Gdy mamy zbiór oczekiwanych nazw argumentów, rozwiązanie opiera się na dwóch ograniczeniach w zakresie *kolejności* ich przekazywania, narzuconych przez Pythona i opisanej w rozdziale 18.:

- W wywołaniu wszystkie argumenty pozycyjne pojawiają się przed wszystkimi argumentami ze słowami kluczowymi.
- W instrukcji `def` wszystkie argumenty bez wartości domyślnej pojawiają się przed wszystkimi argumentami z wartościami domyślnymi.

Oznacza to, że argument niebędący słowem kluczowym nie może w wywołaniu pojawić się po argumencie ze słowem kluczowym, natomiast argument bez wartości domyślnej nie może się pojawić po argumencie z wartością domyślną w *definicji* funkcji. Cała składnia *nazwa=wartość* musi się w obu miejscach pojawiać po wszystkich zwykłych argumentach typu *nazwa*. Ponadto, jak już wiemy, Python dopasowuje wartości argumentów podanych za pomocą pozycji do ich nazw w kolejności od lewej do prawej, przez co wartości te są zawsze przypisywane do nazw znajdujących się w nagłówkach *najbliżej lewej strony*. Natomiast słowa kluczowe dopasowywane są według nazw, a dany argument może mieć tylko jedną wartość.

W celu uproszczenia sobie pracy możemy także założyć, że wywołanie jest ogólnie *poprawne*, to znaczy wszystkie argumenty albo otrzymają wartości (po nazwie bądź pozycji), albo zostaną celowo pominięte w celu pobrania wartości domyślnych. Takie założenie niekoniecznie będzie prawdziwe, ponieważ funkcja nie zostaje jeszcze wywołana, zanim logika opakowująca sprawdzi poprawność. Wywołanie może nadal się nie powieść później, po uruchomieniu za pośrednictwem warstwy opakowującej, z uwagi na niepoprawne przekazanie argumentów. Dopóki jednak nie będzie to powodowało jeszcze większego błędu w warstwie opakowującej, będziemy mogli pracować nad szczegółami poprawności wywołania. Jest to dla nas pomocne, ponieważ sprawdzanie poprawności wywołań przed ich wykonaniem wymagałoby pełnej emulacji algorytmu dopasowywania argumentów Pythona.

## Algorytm dopasowywania

Po przyjęciu powyższych założeń i poznaniu ograniczeń możemy teraz w naszym algorytmie zarówno pozwolić na użycie argumentów ze słowami kluczowymi, jak i pomijanie argumentów z wartościami domyślnymi w wywołaniu. Kiedy wywołanie zostaje przechwycone, możemy poczynić następujące założenia:

1. Niech  $N$  oznacza liczbę argumentów pozycyjnych równą długości krotki `*pargs`.
2. Wszystkie  $N$  przekazanych argumentów pozycyjnych z `*pargs` muszą odpowiadać pierwszym  $N$  oczekiwanych argumentów pozyskanych z obiektu kodu funkcji. Jest to prawdziwe zgodnie z przedstawionymi wcześniej regułami kolejności wywołań Pythona, ponieważ wszystkie argumenty pozycyjne pojawiają się przed wszystkimi argumentami ze słowami kluczowymi.
3. W celu uzyskania nazw argumentów przekazanych po pozycji musimy wykonać wycinek z listy wszystkich oczekiwanych argumentów aż do długości  $N$  krotki pozycyjnej `*pargs`.

4. Wszystkie argumenty po pierwszych *N* oczekiwanych argumentach zostały albo przekazane po słowie kluczowym, albo pominięte w czasie wywołania i przyjmują wartości domyślne.
5. Każda nazwa argumentu sprawdzanego przez dekorator jest przetwarzana w następujący sposób:
  - a. Jeśli znajduje się w `**kargs`, to znaczy, że argument został przekazany za pomocą nazwy i jego wartość jest odczytywana poprzez indeksowanie `**kargs`.
  - b. Jeśli znajduje się ona w pierwszych *N* oczekiwanych argumentach to znaczy, że została przekazana po pozycji (w którym to przypadku jego względna pozycja na liście oczekiwanych argumentów określa jego względną pozycję w krotce `*pargs`).
  - c. W przeciwnym razie możemy założyć, że argument został pominięty w wywołaniu, ma wartość domyślną i nie musi być sprawdzany.

Innymi słowy, możemy pominąć testy argumentów, które zostały pominięte w wywołaniu, zakładając, że pierwszych *N* faktycznie przekazanych argumentów pozycyjnych z `*pargs` musi odpowiadać pierwszym *N* nazw argumentów z listy wszystkich oczekiwanych argumentów, a wszystkie pozostałe albo musiały być przekazane po słowach kluczowych (i tym samym znajdować się w `**kargs`), albo przybierają wartości domyślne. W tym rozwiązaniu dekorator po prostu pominie wszelkie argumenty do sprawdzenia pominięte pomiędzy znajdującym się najbardziej na prawo argumentem pozycyjnym a znajdującym się najbardziej na lewo argumentem ze słowem kluczowym, pomiędzy argumentami ze słowami kluczowymi lub ogólnie po argumentcie pozycyjnym znajdującym się najbardziej na prawo. By przekonać się, w jaki sposób odbywa się to w kodzie, warto prześledzić kod dekoratora oraz jego skryptu testowego.

## Znane problemy

Choć nasze narzędzie sprawdzające przedziały działa zgodnie z planem, ma trzy mankamenty: nie wykrywa niepoprawnych wywołań, nie obsługuje niektórych sygnatur dowolnych argumentów i nie obsługuje zagnieżdżonych dekoratorów. Rozwiązaniem może być napisanie rozszerzenia lub zastosowanie zupełnie innego podejścia. Poniżej przedstawiony jest krótki opis poszczególnych problemów.

### Niepoprawne wywołania

Po pierwsze, jak wspomniano wcześniej, wywołania oryginalnej funkcji, które *nie są poprawne*, nadal nie powiodą się w ostatecznej wersji dekoratora. Przykładowo oba poniższe wywołania zwracają wyjątki `TypeError` z powodu brakującego argumentu pozycyjnego:

```
omitargs()
omitargs(d=8, c=7, b=6)
```

Nie powiodą się one jednak tylko tam, gdzie próbujemy *wywołać* oryginalną funkcję — na końcu obiektu opakowującego. Choć moglibyśmy próbować imitować mechanizm dopasowywania argumentów Pythona w celu uniknięcia tej sytuacji, nie za bardzo jest powód, by to

robić — ponieważ wywołanie na tym etapie i tak by się nie powiodło, możemy równie dobrze pozwolić własnej logice dopasowywania argumentów Pythona na wykrycie tego problemu za nas.

## Dowolne argumenty

Po drugie, choć nasza ostateczna wersja kodu obsługuje argumenty pozycyjne, argumenty ze słowami kluczowymi oraz pominięte wartości domyślne, nadal nie robi nic z `*args` oraz `**args`, które mogą zostać użyte w udekorowanej funkcji przyjmującej *dowolną liczbę* argumentów. Najprawdopodobniej nie będziemy się tym jednak musieli przejmować w naszej sytuacji:

- Jeśli dodatkowy argument ze *słowem kluczowym* zostanie przekazany, jego nazwa pokaże się w słowniku `**kwargs` i zostanie on normalnie przetestowany, o ile zostanie wspomniany w dekoratorze.
- Jeśli dodatkowy argument ze słowem kluczowym nie zostanie przekazany, jego nazwy nie znajdziemy ani w słowniku `**kwargs`, ani w wycinku oczekiwanych argumentów pozycyjnych, dlatego nie zostanie on sprawdzony. Zostanie potraktowany tak, jakby miał mieć wartość domyślną, nawet jeśli tak naprawdę jest to dodatkowy argument opcjonalny.
- Jeśli przekazany zostanie dodatkowy argument *pozycyjny*, i tak nie można się do niego w żaden sposób odwołać w dekoratorze — jego nazwy nie będzie ani w słowniku `**kwargs`, ani w wycinku listy oczekiwanych argumentów, dlatego zostanie po prostu pominięty. Ponieważ takie argumenty nie są wymienione w definicji funkcji, nie można odwzorować nazwy podanej do dekoratora z powrotem na oczekiwaną względną pozycję.

Innymi słowy, w obecnej postaci kod obsługuje sprawdzanie dowolnych argumentów ze słowami kluczowymi po nazwie, ale już nie dowolnych argumentów pozycyjnych, które pozostały bez nazwy i tym samym nie mają ustalonej pozycji w sygnaturze argumentu funkcji. W przypadku interfejsu API obiektu funkcyjnego efekt użycia tych narzędzi w udekorowanej funkcji jest następujący:

```
>>> def func(*kwargs, **pargs): pass
>>> code = func.__code__
>>> code.co_nlocals, code.co_varnames
(2, ('kwargs', 'pargs'))
>>> code.co_argcount, code.co_varnames[:code.co_argcount]
(0, ())

>>> def func(a, b, *kwargs, **pargs): pass
>>> code = func.__code__
>>> code.co_argcount, code.co_varnames[:code.co_argcount]
(2, ('a', 'b'))
```

Ponieważ argumenty z gwiazdkami są widoczne jako zmienne lokalne, a *nie* jako oczekiwane argumenty, nie mają one wpływu na nasz algorytm dopasowujący — poprzedzające je nazwy w nagłówku funkcji można weryfikować w zwykły sposób, ale nie można przekazywać żadnych dodatkowych argumentów pozycyjnych. Co do zasady moglibyśmy rozszerzyć interfejs dekoratora, tak by obsługiwał on `*pargs` w udekorowanej funkcji na potrzeby rzadkich przypadków, w których może się to przydać (na przykład specjalna nazwa argumentu z testem do zastosowania do wszystkich argumentów w krotce `*pargs` obiektu opakowującego wykraczających

poza długość listy oczekiwanych argumentów), ale nie będziemy się tutaj zajmować takim rozszerzeniem.

Pamiętaj również, że to dotyczy wartości w *obiektach zbierających* oznaczonych gwiazdką tylko w nagłówkach funkcji. Ponieważ *rozpakowania* oznaczone gwiazdką w wywołaniach są spłaszczane, zanim dotrą do naszego dekoratora, są nieistotne dla jego kodu. Przypomnijmy nie najlepszy przykład z rozdziału 18.:

```
>>> def f(a, b, c, d, e, f, g, h, i): pass
>>> f._code_.co_varnames[:f._code_.co_argcount]
('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i')

>>> def f(*p, **k): print(p, k)
>>> f(*[1], 2, *[3], 4, f=6, *[5], **dict(g=7), h=8, **{'i': 9})
(1, 2, 3, 4, 5) {'f': 6, 'g': 7, 'h': 8, 'i': 9}
```

Argumenty opatrzone gwiazdką w wywołaniu są rozwiązywane *przed* rozpoczęciem działania funkcji. Ponieważ dekorator znajduje wartości przekazywane do nazw argumentów przez indeksowanie słów kluczowych oraz mapowanie oczekiwanych argumentów na rzeczywiste pozycje, może pozostać zupełnie nieświadomy gwiazdek w wywołaniu i będzie działał normalnie w tym przykładzie (choć, szczerze mówiąc, określenie „normalnie” może być tu przesadą).

## Zagnieżdżone dekoratory

Trzeci i najbardziej subtelny problem polega na tym, że opisany sposób kodowania nie pozwala na pełne **zagnieżdżanie dekoratorów** i łączenie operacji. Ponieważ w definicjach funkcji argumenty są analizowane za pomocą nazw, a nazwy użyte w funkcji pośredniczącej zwracane przez zagnieżdżony dekorator nie odpowiadają nazwom argumentów oryginalnej funkcji lub dekoratora, więc tryb zagnieżdżania nie jest w pełni obsługiwany.

Z technicznego punktu widzenia podczas zagnieżdżania w pełni wykonywane są tylko najgłębiej zagnieżdżone weryfikacje. Na wszystkich pozostałych poziomach zagnieżdżenia testy są wykonywane tylko na argumentach przekazywanych za pomocą słów kluczowych. Prześledźmy kod, aby dowiedzieć się, dlaczego tak jest. Ponieważ sygnatura funkcji pośredniczącej `onCall` nie zawiera nazwanych argumentów pozycyjnych, wszystkie argumenty przeznaczone do sprawdzenia, przekazane za pomocą pozycji, są traktowane, jakby nie były użyte i otrzymały domyślne wartości, dlatego ostatecznie są pomijane.

To może być cecha właściwa zastosowanemu rozwiązaniu. Funkcje pośredniczące zmieniają sygnatury nazw argumentów na ich poziomach, przez co nie ma możliwości bezpośredniego wiązania nazw w argumentach dekoratora z pozycjami przekazywanymi w sekwencjach argumentów. Jeżeli stosowane są funkcje pośredniczące, wtedy *nazwy* argumentów ostatecznie mają zastosowanie tylko w przypadku słów kluczowych. Natomiast wstępne rozwiązanie oparte na *pozycjach* argumentów może lepiej obsługiwać funkcje pośredniczące, ale nie obsługuje w pełni słów kluczowych.

Zamiast implementowania zagnieżdżania w rozwiązaniu quizu na końcu rozdziału uogólnimy dekorator tak, aby pojedynczo realizował *różne rodzaje* weryfikacji. Dodatkowo pokazane tam będą przykłady ograniczeń zagnieżdżania. Ponieważ jednak wyczerpaliliśmy już miejsce



przeznaczone na ten przykład, osoby, które mają ochotę zająć się takimi ulepszeniami, oficjalnie kierujemy do krainy ćwiczeń sugerowanych.

## Argumenty dekoratora a adnotacje funkcji

Co ciekawe, opcja adnotacji funkcji mogłaby stanowić alternatywę dla argumentów dekoratora wykorzystywanych w naszym przykładzie do określania testów przedziałów. Jak wiemy z rozdziału 19., adnotacje pozwalają nam wiązać wyrażenia z argumentami i zwracanymi wartościami poprzez zapisanie ich w kodzie w samym wierszu nagłówka instrukcji `def`. Python zbiera adnotacje w słownik i dołącza je do opisanego w ten sposób funkcji.

Moglibyśmy wykorzystać to w naszym przykładzie do zapisania granic przedziałów w wierszu nagłówka zamiast w argumentach dekoratora. Nadal potrzebowalibyśmy dekoratora funkcji do opakowania funkcji w celu przechwycenia późniejszych wywołań, jednak zamienilibyśmy następującą składnię argumentów dekoratora:

```
@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):
    print(a + b + c)                # func = rangetest(...)(func)
```

na poniższą składnię adnotacji:

```
@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):
    print(a + b + c)
```

Oznacza to, że granice przedziałów zostałyby przesunięte do samej funkcji, a nie pozostawałyby w kodzie poza nią. Skrypt z przykładu 39.27 ilustruje strukturę wynikowych dekoratorów w obu rozwiązaniach, w niekompletnym szkielecie kodu. Wzorzec z argumentami dekoratorów pochodzi z zaprezentowanego wcześniej pełnego rozwiązania. Alternatywa z adnotacjami wymaga o jeden poziom zagnieżdżenia mniej, ponieważ nie musi zachowywać argumentów dekoratorów.

Przykład 39.27. *decoargs-vs-annotation.py*

```
# Wykorzystanie argumentów dekoratora

def rangetest(**argchecks):
    def onDecorator(func):
        def onCall(*pargs, **kargs):
            print(argchecks)
            for check in argchecks:
                pass
            return func(*pargs, **kargs)
        return onCall
    return onDecorator

@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):
    print(a + b + c)

func(1, 2, c=3)
```

*# Tutaj dodanie kodu sprawdzającego*

*# func = rangetest(...)(func)*

*# Wykonuje onCall, argchecks w zakresie*

```
# Wykorzystanie adnotacji funkcji

def rangetest(func):
    def onCall(*pargs, **kargs):
        argchecks = func.__annotations__
        print(argchecks)
        for check in argchecks:
            pass
        return func(*pargs, **kargs)
    return onCall

@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):
    print(a + b + c)

func(1, 2, c=3)
```

*# Tutaj dodanie kodu sprawdzającego*

*# func = rangetest(func)*

*# Wykonuje onCall, adnotacje w funkcji*

Po wykonaniu oba rozwiązania mają dostęp do tych samych informacji testu sprawdzającego, jednak w innych formach. Informacje z wersji z argumentami dekoratora zachowywane są w argumencie w zakresie zawierającym, natomiast informacje z wersji z adnotacją zachowywane są w atrybucie samej funkcji. Natomiast w wersji 3.x ze względu na zastosowanie adnotacji funkcji uzyskujemy następujący wynik:

```
$ python3 decoargs-vs-annotation.py
{'a': (1, 5), 'c': (0.0, 1.0)}
6
{'a': (1, 5), 'c': (0.0, 1.0)}
6
```

Utworzenie reszty wersji opartej na adnotacji pozostawiam jako sugerowane ćwiczenie; jej kod byłby identyczny z kodem zaprezentowanego wcześniej pełnego rozwiązania, ponieważ informacje z testów przedziałów znajdują się po prostu w funkcji zamiast w zakresie zawierającym. Tak naprawdę rozwiązanie to daje nam inny interfejs użytkownika dla naszego narzędzia — nadal, jak wcześniej, będzie ono musiało dopasowywać nazwy argumentów do nazw oczekiwanych argumentów w celu uzyskania ich względnych pozycji.

Właściwie użycie adnotacji w miejsce argumentów dekoratora w tym przykładzie tak naprawdę *ogranicza jego przydatność*. Przenosząc specyfikację sprawdzania poprawności do nagłówka, tak naprawdę ograniczamy funkcję do *jednej roli*. Ponieważ adnotacja pozwala nam na zapisanie w kodzie tylko jednego wyrażenia na argument, może pełnić tylko jeden cel. Przykładowo nie możemy wykorzystać adnotacji testów przedziałów do żadnej innej roli (w tym opcjonalne i nieużywane podpowiadanie typów omówione w rozdziale 6.).

Z kolei, ponieważ argumenty dekoratora zapisywane są w kodzie poza samą funkcją, nie tylko łatwiej jest je usunąć, ale są one również *bardziej ogólne*. Kod samej funkcji nie wymusza jednego celu dekoracji. Tak naprawdę dzięki *zagnieżdżeniu* dekoratorów z argumentami możemy zastosować większą liczbę kroków rozszerzających do tej samej funkcji. Adnotacja w sposób bezpośredni obsługuje tylko jeden krok. Po zastosowaniu argumentów dekoratora sama funkcja zachowuje także prostszy, normalny wygląd.

Mimo to, jeśli mamy na myśli jeden konkretny cel, wybór pomiędzy adnotacjami a argumentami dekoratora jest w dużej mierze subiektywny i sprowadza się do stylistyki. Jak to się często zdarza w życiu, adnotacje jednej osoby dla drugiej mogą być składniowym bałaganem.

## Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy dekoratory w odmianach związanych z funkcjami oraz klasami. Zgodnie z tym, czego się nauczyliśmy, dekoratory są sposobem wstawiania kodu, który będzie wykonywany automatycznie przy definiowaniu funkcji bądź klasy. Kiedy dekorator jest wykorzystywany, Python ponownie dowiązuje nazwę funkcji bądź klasy do zwracanego przez niego obiektu wywołalnego. Ten punkt zaczepienia pozwala nam dodać warstwę logiki opakowującej do wywołań funkcji oraz wywołań tworzących instancje klas w celu zarządzania funkcjami i instancjami. Dekoratory udostępniają rozwiązanie bardziej jednolite i jawne.

Dowiedzieliśmy się również, że dekoratory klas można wykorzystywać do zarządzania samymi klasami, a nie tylko ich instancjami. Ponieważ ta funkcjonalność pokrywa się z *metaklasami*, tematem kolejnego i ostatniego technicznego rozdziału — musisz czytać dalej, aby poznać zakończenie tej historii i całej tej książki.

Najpierw jednak należy wykonać quiz kończący rozdział. Ponieważ rozdział ten skupiał się w przeważającej mierze na większych przykładach, w quizie Czytelnik zostanie poproszony o zmodyfikowanie części kodu w celu sprawdzenia go. Zarówno oryginalne wersje kodów, jak i rozwiązania znajdują się w załączonych do książki plikach (patrz wskazówki w przedmowie). Rozwiązania znajdziesz w folderze dla tego rozdziału, w podfolderze `_QuizAnswers`. Jeżeli czas nagli, można przestudiować modyfikacje opisane w odpowiedziach — programowanie polega nie tylko na pisaniu kodu, ale też na czytaniu go.

## Sprawdź swoją wiedzę — quiz

1. **Dekoratory metod:** jak wspomniano w jednej ze wskazówek rozdziału, napisany w podrozdziale „Dodawanie argumentów dekoratora” i zawarty w pliku `timerdeco2.py` dekorator z argumentami mierzący czas wykonania funkcji można zastosować jedynie do prostych funkcji, ponieważ wykorzystuje on zagnieżdżoną klasę z metodą przeciążania operatora `__call__` przechwytyującą wywołania. Struktura ta nie działa jednak dla *metod* klas, ponieważ do `self` przekazywana jest instancja dekoratora, a nie instancja klasy podmiotowej. Należy przepisać ten dekorator w taki sposób, by można go było zastosować zarówno do prostych funkcji, jak i metod klas, a następnie przetestować na funkcjach i metodach. (Uwaga: wskazówek należy szukać w podrozdziale zatytułowanym „Uwagi na temat klas I — dekorowanie metod klas”). Warto zauważyć, że można skorzystać z przypisywania *atrybutów* obiektów funkcji w celu śledzenia całkowitego czasu, ponieważ nie będziemy mieli zagnieżdżonej klasy dla celów zachowywania stanu i nie możemy uzyskać dostępu do zmiennych nielokalnych spoza kodu dekoratora.
2. **Dekoratory klas:** dekoratory klas `Public` i `Private`, napisane w niniejszym rozdziale (zawarte w pliku `access2.py`), dodają pewne *obciążenie* związane z każdym pobraniem atrybutu w udekorowanej klasie. Choć moglibyśmy po prostu usunąć wiersz z dekoracją `@` w celu uzyskania szybkości kodu, możemy również rozszerzyć sam dekorator w taki sposób, by sprawdzał przełącznik `__debug__` i nie wykonywał opakowywania wtedy, gdy opcja `-O`

Pythona została przekazana w wierszu poleceń (tak samo, jak robiliśmy to w przypadku dekoratorów sprawdzających przedziały argumentów). W ten sposób możemy zwiększyć szybkość działania programu bez zmiany jego źródła za pomocą samych argumentów wiersza poleceń (`python -O main.py`). Możemy również wykorzystać jedną z opisanych technik wykorzystujących nadrzędne klasy mieszane i przechwytywać kilka *wbudowanych operacji*. Należy napisać kod takiego rozszerzenia oraz odpowiedniego testu.

3. **Uniwersalne weryfikowanie argumentów:** dekoratory funkcji i metod, zawarte w pliku *rangetest.py* z przykładu 39.25, sprawdzają, czy wartości argumentów mieszczą się w dopuszczalnych zakresach. Jak się jednak przekonaliśmy, ten sam wzorec można stosować do innych celów, na przykład do sprawdzania typów argumentów. Zadanie polega na uogólnieniu dekoratora sprawdzającego zakresy, tak aby można było jeden kod wykorzystywać do weryfikowania poprawności argumentów według *różnych* kryteriów. Biorąc pod uwagę przedstawiony tu sposób kodowania, najprostszym rozwiązaniem może być przekazywanie funkcji, jednak w kontekście bardziej obiektowego programowania podobne uogólnienie można osiągnąć również za pomocą podklas zawierających wymagane metody. To nie lada wyzwanie, więc koniecznie zapoznaj się z rozwiązaniem, aby uzyskać wskazówki.

## Sprawdź swoją wiedzę — odpowiedzi

Jak już wspomniano wcześniej, rozwiązania zadań z tego quizu znajdują się w podfolderze *\_QuizAnswers* dla tego rozdziału w pakiecie przykładów z książki. Każde pytanie ma tam swój własny podfolder z plikami, a plik tekstowy *\_Notes.txt* zawiera informacje wprowadzające. W tej edycji zdecydowano się przenieść te rozwiązania do sieci zamiast umieszczać je tutaj, ponieważ pozwala to zaoszczędzić około 10 stron.