


Wprowadzenie do OOP

atrybuty dynamiczne, wybrane metody magiczne

Dodatkowe techniki związane z programowaniem obiektowym, takie jak atrybuty dynamiczne, settery, przeciążanie operatorów, sprawiają, że kod Pythona może być jeszcze bardziej zwięzły i ekspresyjny.

12+

DOWIESZ SIĘ

 Jak tworzyć i wykorzystywać atrybuty dynamiczne w klasach oraz w jaki sposób zdefiniować działanie operatorów takich jak „+” czy „-” pomiędzy instancjami klas, a także instancją klasy i innymi obiektami.

POTRZEBNA WIEDZA

 Umiejętność tworzenia funkcji i definiowania prostych klas w Pythonie.

ATRYBUT DYNAMICZNY — PROPERTY

W artykule „Wprowadzenie do OOP” z poprzedniego wydania PJr zdefiniowaliśmy prostą klasę reprezentującą kwadrat. Dobrym zwyczajem w świecie programowania jest pisanie kodu z użyciem angielskich nazw. Język ten jest bardzo popularny i zrozumiały na całym świecie – szczególnie w takiej dziedzinie jak programowanie. Pisząc kod, staramy się, by był zrozumiały i czytelny dla innych programistów – także dla tych, którzy nie znają naszego ojczystego języka. Odtworzmy więc definicję kwadratu – tym razem z użyciem nazw angielskich: kwadrat to *square*, bok to *side*, a obwód to *circuit*.

Listing 1. Definicja klasy „Square”

```
class Square:
    def __init__(self, side):
        self.side = side
    def __str__(self):
        return f"<Square, side: {self.side}>"
    def circuit(self):
        return self.side * 4
```

Mając klasę, możemy utworzyć kwadrat, podając długość jego boku, i sprawdzić, jaki jest jego obwód, wywołując metodę `circuit`. Możemy też ustawić nową długość boku, co wpłynie na obwód kwadratu – tak jak w przykładzie z Listingu 2. W komentarzu podano wartości, jakie będą drukowane po uruchomieniu programu.

Listing 2. Zmiana boku kwadratu powoduje zmianę jego obwodu

```
square = Square(4)
print(square.side)           # 4
print(square.circuit())      # 16
square.side = 5
print(square.circuit())      # 20
```

Zauważmy, że do boku odwołujemy się tylko poprzez samą nazwę: `square.side`. W przypadku obwodu wywołujemy metodę, a to oznacza, że musimy dodać nawias: `square.circuit()`.

Obwód, bok, pole powierzchni, przekątna to cechy kwadratu. Obecnie odwołując się do nich, czasem musimy użyć nawiasów, a czasem nie (Listing 2). Dobrze

by było to ujednolicić tak, by korzystając z cech (atributów), nie trzeba było dodawać nawiasów wywołujących metodę, na przykład: `square.circuit`. Możemy to osiągnąć, specjalnie oznaczając metodę `circuit`, co zostało przedstawione w Listingu 3. Oznaczenie to nazywane jest też dekorowaniem.

Listing 3. Zastosowanie dekoratora „@property”

```
class Square:

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

    def __str__(self):
        return f"<Square, side: {self.side}>"

    @property
    def circuit(self):
        return self.side * 4

square = Square(4)
print(square.side)
print(square.circuit) # 16
square.side = 5
print(square.circuit) # 20
```

@property to tak zwany dekorator – więcej o dekoratorach możesz przeczytać na końcu tego artykułu.

Po zastosowaniu dekoratora @property nie musimy już wywoływać metody. Wystarczy, że odwołamy się do nazwy atrybutu – w taki sam sposób jak dla boku. Takie udekorowane metody nazywamy atrybutami dynamicznymi, wyliczalnymi.

Czy te atrybuty działają jednak na pewno tak samo? Jak widzimy w listingach, atrybut `side` możemy ustawić. Czy tak samo jest z obwodem? Spróbujmy:

```
square.circuit = 40

Traceback (most recent call last):
  File "PRJ12/main.py", line 39, in <module>
    square.circuit = 40
AttributeError: can't set attribute
```

Ustawienie atrybutu kończy się błędem. By to poprawić, musimy dodać specjalną i do tego odpowiednio udekorowaną metodę do naszej klasy. Nazywamy ją setterem. Przykład znajdziemy w Listingu 4.

Listing 4. Dodanie definicji settera

```
class Square:

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

    def __str__(self):
        return f"<Square, side: {self.side}>"

    @property
    def circuit(self):
        return self.side * 4

    @circuit.setter
    def circuit(self, value):
        self.side = value / 4
```

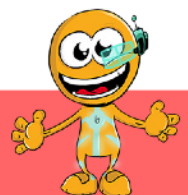
Jak widać, poprzednio udekorowana metoda `circuit` stała się obiektem, który ma atrybut `setter`. Ten atrybut pojawił się tu w wyniku zastosowania dekoratora `property`. Posłużył on nam do udekorowania nowej metody, która ma ustawić obwód: @circuit.setter. Co charakterystyczne, nowa metoda ma tę samą nazwę, ale tym razem przyjmuje argument, który nazwalimy `value`. Od teraz jeśli interpreter natrafi na linię ustawiającą ten atrybut, zostanie wykonana właśnie ta metoda. I właśnie dlatego nazywamy ją setterem – od słowa `set` oznaczającego „ustaw”

```
square.circuit = 40
print(square.side) # 10
```

Tym razem ustawienie obwodu wpływa na długość boku.

Utrwalmy to, dodając atrybut dynamiczny oraz setter do wyliczenia pola powierzchni. Ustawienie pola

CIEKAWOSTKA



„NotImplemented” to jedna z pięciu stałych wbudowanych w Pythona. Oprócz niej są to jeszcze „True”, „False”, „None” oraz „Ellipsis”, który ma też inny zapis w postaci „...”.

Stałe to takie obiekty, których wartości nie możemy zmienić.

powinno wpłynąć na długość boku kwadratu. By to zrobić, musimy umieć wyciągnąć pierwiastek kwadratowy z pola powierzchni. Można do tego użyć specjalnej metody `sqrt` z modułu `maths`, albo po prostu podnieść wartość do potęgi 0.5 – Listing 5.

Listing 5. Dodanie do klasy „Square” dynamicznego atrybutu „area” wraz z setterem

```
class Square:

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

    def __str__(self):
        return f"<Square, side: {self.side}>"

    @property
    def circuit(self):
        return self.side * 4

    @circuit.setter
    def circuit(self, value):
        self.side = value / 4

    @property
    def area(self):
        return self.side ** 2

    @area.setter
    def area(self, value):
        self.side = value ** 0.5

square = Square(4)
print(square.side)      # 4
print(square.circuit)   # 16
print(square.area)      # 16

square.area = 25
print(square.side)      # 5.0
print(square.circuit)   # 20.0
```

PRZECIĄŻANIE OPERATORÓW

Warto zastanowić się nad tym, w jaki sposób Python rozpoznaje to, jak mają działać na obiekty różne operatory. Przypomnijmy tu, że operatory to takie znaki jak `+`, `-`, `==`. Niektóre operatory wstawiamy między dwa obiekty: `a + b`. Inne mogą działać tylko na jeden: `-a`. Na przykład dodanie do siebie dwóch list da nam w wyniku połączoną listę. Podobnie jeśli dodamy

do siebie dwa napisy. Dodanie dwóch liczb zwróci wynik dodawania.

```
[1, 2] + [3, 4]      # [1, 2, 3, 4]
"ab" + "ab"          # "abab"
3 + 1                 # 4
```

Spróbujmy dodać do siebie dwie instancje naszego kwadratu:

Listing 6. Próba dodania do siebie dwóch kwadratów i jej wynik

```
a = Square(10)
b = Square(5)

a + b

Traceback (most recent call last):
  File "/PRJ12/main.py", line 38, in <module>
    a + b
TypeError: unsupported operand type(s) for +:
'Square' and 'Square'
```

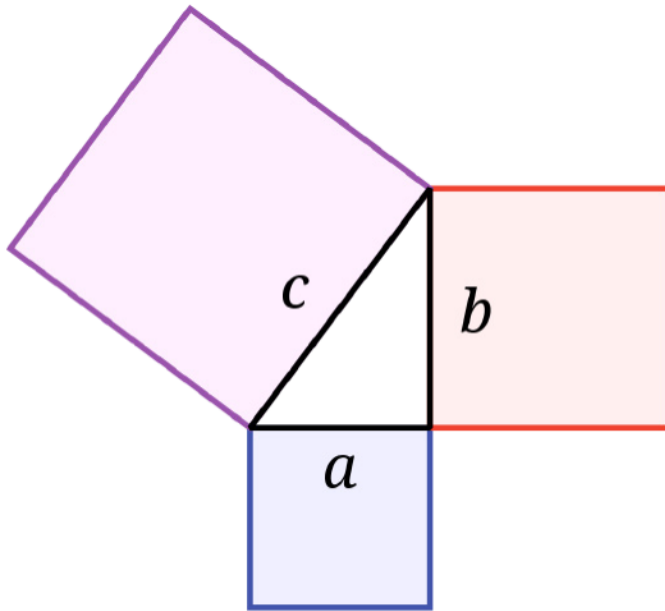
Python poinformował nas, że nie wie, jak to zrobić. Musimy go więc poinstruować. Po pierwsze, sami musimy ustalić, co w zasadzie ma dla nas oznaczać dodanie do siebie dwóch kwadratów. Myślę, że dobrym pomysłem jest utworzenie z nich nowego kwadratu o polu powierzchni równym sumie pól pierwotnych kwadratów. Ot, takie Twierdzenie Pitagorasa w praktyce.

WARTO WIEDZIEĆ



W Pythonie brak implementacji określonych przypadków w metodach czy funkcjach możemy oznaczyć, rzucając wyjątek (najczęściej „`NotImplementedError`”) lub zwracając obiekt „`NotImplemented`”.

Rzucenie wyjątku przerywa wywołanie funkcji i czeka na ewentualne przechwycenie i obsługę tego wyjątku. Zwrócenie obiektu `NotImplemented` spowoduje, że interpreter spróbuje wywołać tak zwaną funkcję odbicia (ang. *reflection function*). Na przykład dla metody „`__add__`” jest to „`__radd__`”, opisująca sytuację, w której obiekt jest po prawej stronie operatora. Zwrócenie obiektu „`NotImplemented`” stosujemy więc w tych metodach specjalnych, które obsługują dwuarumentowe działania operatorów.



Ilustracja 1. Pole kwadratu *c* jest sumą pól kwadratów *a* i *b*

Z punktu widzenia Pythona obsługa znaku `+` (nazywamy go w tym kontekście operatorem) polega na wywołaniu specjalnej metody w obiekcie. Metoda ta musi mieć nazwę: `__add__`, włączając w to znaki podkreślenia. To kolejna po `__str__` oraz `__init__` specjalna metoda. Nazywamy je metodami magicznymi albo też *dunder methods* (od *double under* – podwójne podkreślenie). Może ona mieć następującą definicję:

Listing 7. Dodanie metody `__add__` i wykorzystanie operatora `+` do utworzenia nowej instancji „Square”

```
class Square:
    ...
    def __add__(self, other):
        s = Square(1)
        s.area = self.area + other.area
        return s

a = Square(3)
b = Square(4)

print(a + b) # <Square, side: 5.0>
```

Wynikiem działania tej metody, zgodnie z tym jak ustaliliśmy wcześniej, powinna być instancja klasy `Square`. Tworzymy ją więc na początku, a następnie ustawiamy

jej pole, korzystając z settera `area`, i następnie tę nową instancję zwracamy. Zwróćmy uwagę na to, że metoda `__add__` przyjmuje dwa argumenty. W tym przypadku `self` to instancja z lewej strony znaku `+`, czyli `a`. Z kolei `other` to instancja `b`.

Moglibyśmy też pokusić się o opisanie sytuacji, w której do kwadratu dodamy jakąś liczbę. Przy obecnej definicji zakończy się to błędem – Listing 8.

Listing 8. Dodanie liczby do instancji klasy „Square”

```
a + 2
Traceback (most recent call last):
  File "/PRJ12/main.py", line 45, in <module>
    print(a + 2)
  File "/PRJ12/main.py", line 27, in __add__
    s.area = self.area + other.area
AttributeError: 'int' object has
no attribute 'area'
```

Tym razem z prawej strony znaku `+` stoi `2`. Czyli instancja klasy `int`. Nie ma ona atrybutu `area`, stąd błąd. Co ciekawe – już teraz widać, że operacja zakończyłaby się sukcesem, gdyby obiekt z prawej strony znaku `+` miał atrybut `area` zwracający liczbę.

Jeszcze inny komunikat błędu pojawi się, jeśli elementy zamienimy miejscami.

Listing 9. Próba dodania obiektu „Square” do liczby

```
2 + a
Traceback (most recent call last):
  File "/PRJ12/main.py", line 45, in <module>
    print(2 + a)
TypeError: unsupported operand type(s) for +:
'int' and 'Square'
```

Najpierw musimy ustalić, co ma się stać, gdy do naszego kwadratu dodamy liczbę. Powiedzmy, że w tym przypadku będzie to oznaczało zwrócenie nowego kwadratu o wydłużonym o tę liczbę boku. Musimy tak przerobić metodę `__add__`, by uwzględniała typ drugiego obiektu. Wygodnie jest zastosować do tego wbudowaną funkcję `isinstance`. Sprawdza ona, czy jej

pierwszy argument jest instancją typu (lub krotki typów) podanego jako drugi argument. Wykorzystajmy do tego wyrażenie warunkowe `if elif else` tak jak w Listingu 10.

Listing 10. Uwzględnienie typów w metodzie `__add__`

```
class Square:
    ...
    def __add__(self, other):
        if isinstance(other, (int, float)):
            return Square(self.side + other)
        elif isinstance(other, Square):
            s = Square(1)
            s.area = self.area + other.area
            return s
        else:
            return NotImplemented
```

Wartość `NotImplemented`, zwrócona gdy `other` nie jest ani liczbą, ani typu `Square`, pozwoli Pythonowi poinformować użytkownika o tym, że dla innych typów operacja ta nie jest zaimplementowana.

Dodanie liczby do naszego kwadratu zadziała teraz poprawnie:

```
print(a + 2)
<Square, side: 5>
```

Aby Python rozumiał odwróconą kolejność, musimy zdefiniować drugą metodę. `__radd__`.

W sytuacji dodawania: `a + b` – jeśli w `a` nie ma metody `__add__` – Python poszuka metody `__radd__` w `b`. Słowo *add* oznacza dodawanie. Przedrostek *r* wziął się od słowa *right* – określającego prawą stronę. Tym razem `self` oznacza obiekt `b`, czyli ten, z którego brana jest metoda, `other` zaś to `a`.

Listing 11. Określenie dodawania dla obiektu stojącego po prawej stronie operatora `+`

```
class Square:
    ...
    def __add__(self, other):
        if isinstance(other, (int, float)):
```

```
            return Square(self.side + other)
        elif isinstance(other, Square):
            s = Square(1)
            s.area = self.area + other.area
            return s
        else:
            return NotImplemented

    def __radd__(self, other):
        return self.__add__(other)
```

Definicja `__radd__` jest dość prosta. Wystarczy, że wywołamy i zwrócimy wcześniej zdefiniowaną metodę `__add__`, przekazując do niej odpowiednie elementy. Zapis:

```
return self.__add__(other)
```

równoważny jest zapisowi:

```
return Square.__add__(self, other)
```

Jak widzimy, jeśli odwołujemy się do metody poprzez instancję, to instancja ta automatycznie staje się pierwszym argumentem wywołania.

Oprócz `__add__` i `__radd__` mamy jeszcze wiele innych metod. Przykłady kilku z nich znajdziesz w poniższej tabeli. Pozostałe można odnaleźć w dokumentacji: <https://docs.python.org/3/library/operator.html?highlight=operators>.

OPERATOR	METODY
-	<code>__sub__</code> , <code>__rsub__</code>
*	<code>__mul__</code> , <code>__rmul__</code>
==	<code>__eq__</code>
>=	<code>__gte__</code>

CZYM SĄ DEKORATORY?

Dekorator to rodzaj funkcji, która ma za zadanie dodać jakieś dodatkowe efekty do działania innej funkcji. Chcemy jakby udekorować oryginalne efekty działania czymś dodatkowym.

Dekoratory mają specyficzną budowę. Jako argument dekorator przyjmuje funkcję (`func`).

Wewnątrz dekoratora definiowana jest inna funkcja – tak zwane opakowanie (wrapper). W naszym przykładzie przyjmuje ona dowolną ilość argumentów pozycyjnych i nazwanych.

W tej funkcji właśnie określone są te dekorujące dodatkowe efekty. Wywoływana jest tu oryginalna funkcja (`result = func(*args, **kwargs)`), a jej rezultat jest zwracany.

Sam dekorator zwraca funkcję opakowania.

```
def decorator(func):
    def wrapper(*args, **kwargs):
        print("Do something before")
        result = func(*args, **kwargs)
        print("Do something after")
        return result
    return wrapper
```

Jeśli mamy jakąś funkcję, na przykład taką jak `add`:

```
def add(a, b):
    return a + b
```

to zastosowanie dekoratora możemy wykonać w następujący sposób:

```
add = decorator(add)
```

Widzimy tu wyraźnie, że `add` staje się nową funkcją. Najczęściej dekorator stosuje się jednak przy definicji funkcji, wykorzystując specjalny zapis:

```
@decorator
def add(a, b):
    return a + b
```

Wywołanie udekorowanej funkcji:

```
print(add("Hello", "World"))
```

da nam wynik:

```
Do something before
Do something after
HelloWorld
```

Rafał Korzeniewski

Z wykształcenia muzyk-puzonista i fizyk. Z zamiłowania i zawodu Pythonista. Trener Pythona, współorganizator PyWaw (<http://pywaw.org>) – warszawskiego meetupu poświęconego Pythonowi. W wolnych chwilach uczy się gry na nowych instrumentach, udziela się społecznie i dużo czyta.

KORZENIEWSKI@GMAIL.COM



ZAPAMIĘTAJ

- 💡 Atrybut dynamiczny tworzymy przy pomocy metody udekorowanej przez „@property”.
- 💡 Do ustawienia wartości atrybutu dynamicznego konieczne jest zdefiniowanie metody settera.
- 💡 Zachowanie operatorów w odniesieniu do obiektów naszego typu opisujemy w specjalnych metodach z dwoma podkreśleniami.

ĆWICZ W DOMU

- 💡 Stwórz w domu klasę opisującą okrąg – „Circle” – z takimi atrybutami jak promień, obwód, średnica, pole powierzchni. Niech wynikiem dodania dwóch okręgów będzie trzeci – taki, którego pole powierzchni jest równe sumie pól dwóch pierwszych.
- 💡 Spróbuj opisywać inne obiekty i symulować w Pythonie ich wygląd i zachowanie.