Python

Poziom średniozaawansowany

Autor: Michał Nowotka
Prawa do korzystania z materiałów posiada Software Development Academy

DEKORATORY

Dekoratory

- Do tej pory nauczyliśmy się używać gotowych dekoratorów, takich jak **@staticmethod**, **@classmethod** czy **@abstractmethod**.
- Teraz spróbujemy nauczyć się definiować własne dekoratory.
- To ćwiczenie może pozwolić nam zrozumieć jak potężnym narzędziem w Pythonie są funkcje.

Funkcje - obiekty pierwszej kategorii

- W Pythonie funkcje są obiektami pierwszej kategorii. To znaczy, że funkcję (nie tylko jej wynik) można przypisać do zmiennej.
- Można ją przekazać jako argument do innej funkcji.
- Na przykładzie obok, funkcja **greet_kryśka**, przyjmuje jako argument inną funkcję, po czym wywołuje ją jako argument podając imię “Kryśka”.

```
In [1]: def say_hello(name):
   ...:     print(f'Siemka {name}!')
   ...:

In [2]: def say_goodbye(name):
   ...:     print(f'Trzymaj się, {name}!')
   ...:

In [3]: def greet_kryśka(greeting):
   ...:     return greeting('Kryśka')
   ...:

In [4]: greet_kryśka(say_hello)
Siemka Kryśka!

In [5]: greet_kryśka(say_goodbye)
Trzymaj się, Kryśka!
```

Funkcje wewnętrzne

- W Pythonie można również tworzyć funkcje wewnątrz innych funkcji.
- Taka wewnętrzna funkcja działa na wyłączny użytek funkcji, we wnętrzu której jest zdefiniowana. Nie można się do niej dostać z zewnątrz w żaden sposób.
- Na przykładzie obok mamy funkcję, która w swoim ciele definiuje inną funkcję, po czym ją wywołuje.

```
In [1]: def outer_function(text):
   ...:     print('outer function')
   ...:     def inner_function(txt):
   ...:         print('inner function')
   ...:         return txt.upper()
   ...:     return inner_function(text)
   ...:

In [2]: outer_function('Ala ma kota')
outer function
inner function
Out[2]: 'ALA MA KOTA'
```

Zwracanie funkcji

- Jedna funkcja może zwracać inną funkcję jako wynik swojego działania.
- Taką funkcję nazywamy fabryką.

```
In [9]: def greeting(time):
   ...:
   ...:     def morning_greeting(name):
   ...:         return f'Good morning, {name}!'
   ...:
   ...:     def afternoon_greeting(name):
   ...:         return f'Good afternoon, {name}!'
   ...:
   ...:     def late_night_greeting(name):
   ...:         return f'Good night, {name}!'
   ...:
   ...:     if time is 'morning':
   ...:         return morning_greeting
   ...:     if time is 'afternoon':
   ...:         return afternoon_greeting
   ...:     return late_night_greeting
   ...:

In [10]: greeting_fun = greeting('morning')

In [11]: greeting_fun('John')
Out[11]: 'Good morning, John!'
```

Prosty dekorator

- Nawet najprostszy dekorator używa wszystkich trzech poznanych technik:
  - przyjmuje funkcję jako swój argument
  - definiuje wewnętrzną funkcję
  - zwraca funkcję jako wynik swojego działania

```
In [1]: def my_decorator(func):
   ...:     def wrapper():
   ...:         return 'DECORATED --> ' + func() + ' <-- DECORATED'
   ...:     return wrapper
   ...:

In [2]: def hello_world():
   ...:     return 'Hello, world!'
   ...:

In [3]: print(hello_world())
Hello, world!

In [4]: print(my_decorator(hello_world)())
DECORATED --> Hello, world! <-- DECORATED
```

Prosty dekorator

- Dekorator jako jedyny argument, przyjmuje funkcję, którą ma udekorować.
- W środku definiuje funkcję tradycyjnie nazywaną **wrapper**, która wykonuje funkcję, którą dekorujemy ale oprócz tego wykonuje jakąś dodatkową czynność (w naszym przykładzie dopisuje dodatkowe napisy przed i po wyniku funkcji)
- Zwraca **wrapper** jako swój wynik.

```
In [1]: def my_decorator(func):
   ...:     def wrapper():
   ...:         return 'DECORATED --> ' + func() + ' <-- DECORATED'
   ...:     return wrapper
   ...:

In [2]: def hello_world():
   ...:     return 'Hello, world!'
   ...:

In [3]: print(hello_world())
Hello, world!

In [4]: print(my_decorator(hello_world)())
DECORATED --> Hello, world! <-- DECORATED
```

Prosty dekorator

- Zauważmy, że po zastosowaniu dekoratora, zwraca on oryginalną funkcję owiniętą we wrapper. Zatem końcowy użytkownik nie dotyka teraz oryginalnej funkcji tylko wrappera.
- To znaczy, że wrapper musi przyjmować dokładnie takie argumenty co opakowana funkcja żeby mógł je do tej opakowanej funkcji przekazać.
- Problem polega na tym, że ciężko z góry przewidzieć jaką funkcję ktoś opakuje w dekorator, który piszemy i jakie ona będzie mieć argumenty.
- Dlatego też sygnatura wrappera powinna zawsze wyglądać tak:
  def wrapper(*args, **kwargs), dzięki czemu jesteśmy w stanie opakować dowolną funkcję.

```
In [1]: def my_decorator(func):
   ...:     def wrapper(*args, **kwargs):
   ...:         return 'DECORATED --> ' + func(*args, **kwargs) + ' <-- DECORATED'
   ...:     return wrapper
   ...:

In [2]: def hello_name(name):
   ...:     return f'Hello, {name}!'

In [3]: def hello_two_names(first_name, second_name):
   ...:     return f'Hello, {first_name} and {second_name}!'

In [4]: hello_name('Krzysiek')
Out[4]: 'Hello, Krzysiek!'

In [5]: my_decorator(hello_name)('Krzysiek')
Out[5]: 'DECORATED --> Hello, Krzysiek! <-- DECORATED'

In [6]: hello_two_names('Krzysiek', 'Olga')
Out[6]: 'Hello, Krzysiek and Olga!'

In [7]: my_decorator(hello_two_names)('Krzysiek', 'Olga')
Out[7]: 'DECORATED --> Hello, Krzysiek and Olga! <-- DECORATED'
```

Prosty dekorator

- W dwóch poprzednich slajdach dekoratora używaliśmy w ten sposób: **dekorator(dekorowana_funkcja)(argumenty)**
- Takie użycie dekoratora nie jest zbyt czytelne, ale przecież Python daje nam zupełnie inny sposób.
- Kiedy mamy już zadeklarowany dekorator możemy go użyć dosłownie dekorując funkcję pisząc nad nią **@nazwa_dekoratora**.

```
In [1]: @my_decorator
   ...: def my_function(arg1, arg2):
   ...:     pass
   ...:
```

Prosty dekorator

Napisz dekorator o nazwie thrice, który powoduje wykonanie się opakowanej funkcji trzykrotnie.

Prosty dekorator

Napisz dekorator, który przed wykonaniem opakowywanej funkcji wypisze jej argumenty, a po wykonaniu wyprintuje komunikat „Wykonano z x argumentami”, gdzie x to liczba podanych argumentów do opakowywanej funkcji.

Prosty dekorator

Stwórz dekorator, który wykona funkcję opakowywaną lub nie, w zależności od wartości zmiennej globalnej SHOULD_BE_RUN (wartość True uruchamia funkcję, wartosc False oznacza, że powinien się pojawić tylko napis „Pomijam…”.

Prosty dekorator - debugowanie

- Każda funkcja ma kilka specjalnych atrybutów:
  - **__name__** - przechowuje nazwę funkcji
  - **__doc__** - przechowuje dokumentację (docstringi) funkcji drukowane wtedy, kiedy ktoś wywoła wbudowaną w Pythona funkcję **help**.
  - **__module__** - ścieżkę do modułu, w którym zdefiniowana jest funkcja.
- Niestety kiedy udekorujemy funkcję, wszystkie wymienione wyżej atrybuty będą pobrana nie z tej funkcji a z wrappera.
- To znacznie utrudnia debugowanie.
- Dlatego Python w swoim standardowym module o nazwie **functools** dostarcza dekoratora **@wraps**.
- Tym dekoratorem należy udekorować wrapper a wtedy zaciągnie on atrybuty z funkcji, którą opakowuje.

```
In [6]: hello_name.__name__
Out[6]: 'wrapper'

In [7]: hello_name.__module__
Out[7]: '__main__'

In [8]: print.__name__
Out[8]: 'print'

In [9]: print.__module__
Out[9]: 'builtins'
```

Prosty dekorator - ostateczna wersja

```
In [1]: from functools import wraps

In [2]: def my_decorator(func):
   ...:     @wraps(func)
   ...:     def wrapper(*args, **kwargs):
   ...:         return 'DECORATED --> ' + func(*args, **kwargs) + ' <-- DECORATED'
   ...:     return wrapper
   ...:

In [3]: @my_decorator
   ...: def hello_name(name):
   ...:     return f'Hello, {name}!'
   ...:

In [4]: hello_name('Krzysiek')
Out[4]: 'DECORATED --> Hello, Krzysiek! <-- DECORATED'

In [5]: hello_name.__name__
Out[5]: 'hello_name'
```

Autor: Michał Nowotka
Prawa do korzystania z materiałów posiada Software Development Academy

Prosty dekorator

Dodaj do poprzednich funkcji docstringi oraz sprawdź działanie atrybutów __name__, __doc__ oraz __module__. Opakuj funkcje z wykorzystaniem dekoratora wraps tak, by wartość __name__ była taka sama jak opakowywanej funkcji.

Prosty dekorator

Stwórz dekorator obliczający i wypisujący czas działania funkcji.

LAMBDA – FUNKCJA ANONIMOWA

Wyrażenia lambda

```
In [60]: def identity(x):
    ...:     return x
    ...:
```

vs

```
identity = lambda x: x
```

Wyrażenia lambda

- Wyrażenie lambda to inaczej anonimowa funkcja.
- Jest to funkcja która, nie posiada nazwy i jest definiowana w miejscu jej użycia.
- Ponieważ nie jest zwykłą funkcją nie definiuje się jej za pomocą słowa kluczowego **def**. Służy do tego oddzielne słowo kluczowe **lambda**.
- Po słowie kluczowym następuje lista argumentów rozdzielonych przecinkami.
- Po liście argumentów pisze się dwukropek a po nim pojedyncze wyrażenie, które zwraca wartość funkcji.
- Lambda może mieć w swoim ciele tylko jedno wyrażenie.

```
In [1]: my_lambda = lambda x: x.lower()

In [2]: my_lambda('HA HA HA')
Out[2]: 'ha ha ha'

In [3]: square_lambda = lambda x: x**2

In [4]: square_lambda(4)
Out[4]: 16

In [5]: equals_lambda = lambda x, y: x == y

In [6]: equals_lambda(1, 1)
Out[6]: True

In [7]: equals_lambda(1, 2)
Out[7]: False
```

map, reduce, filter

- Można zapytać do czego służą wyrażenia lambda?
- Najczęstszym miejscem ich użycia są funkcje **map**, **reduce** oraz **filter**.
- Wbudowana funkcja **map** przyjmuje dwa parametry. Pierwszym z nich jest jednoargumentowa funkcja, która zostanie użyta do przekształcenia (przemapowania) wszystkich elementów. Drugim parametrem jest kolekcja tych elementów. Funkcja ta przekształca elementy kolekcji używając podanej funkcji, którą najczęściej jest wyrażenie lambda.
- Funkcja **reduce** ze standardowego modułu **functools** redukuje kolekcję elementów podanych jako drugi parametr przy użyciu dwuargumentowej funkcji podanej jako pierwszy parametr. Funkcje **map** i **reduce** są często używane łącznie - najpierw przekształcamy elementy a później je redukujemy.

```
In [1]: items = [1, 2, 3, 4, 5]

In [2]: squared = list(map(lambda x: x**2, items))

In [3]: squared
Out[3]: [1, 4, 9, 16, 25]

In [4]: from functools import reduce

In [5]: square_sum = reduce((lambda x, y: x + y), squared)

In [6]: square_sum
Out[6]: 55
```

map

- dla każdego elementu z listy LISTA wykonaj funkcję FUNC,
- przetworzone elementy można przypisać do nowej listy,
- bez rzutowania list(map(…)) funkcja map zwraca generator.

```
map(FUNC, LISTA)
```

```
new_list = list(map(FUNC,LISTA)
```

map, reduce, filter

Z listy pierwszych dziesięciu liczb naturalnych, utwórz listę kwadratów pierwszych liczb naturalnych wykorzystując:

a) pętlę for,
b) listę składaną (list comprehension),
c) funkcję map oraz lambdę.

reduce

- redukuje listę LISTA do jednego elementu na podstawie funkcji FUNC,
- Funkcja FUNC przyjmuje dwa argumenty i zwraca wynik operacji między tymi argumentami.

```
reduce(FUNC, LISTA)
```

```
new_value = reduce(FUNC,LISTA)
```

map, reduce, filter

Na podstawie listy liczb od 10 do 20, wylicz iloczyn liczb się w niej znajdujących wykorzystując:

a) pętlę for,

b) funkcję reduce oraz lambdę.

map, reduce, filter

- Wbudowana funkcja **filter**, podobnie jak dwie poprzednie również przyjmuje jako swoje parametry funkcję oraz kolekcję. Funkcja musi być jednoargumentowa.
- Jako wynik zwraca te elementy kolekcji, dla których funkcja zwróciła **True**. Te dla których funkcja zwróciła **False** zostały odfiltrowane.
- Na przykładzie obok funkcja odfiltrowuje parzyste elementy kolekcji liczb całkowitych z przedziału od jednego do stu.
- W efekcie zostały nam tylko liczby nieparzyste.

```
In [3]: odds = list(filter(lambda x: x%2, range(1, 101)))

In [4]: odds
Out[4]:
[1,
 3,
 5,
 7,
 9,
 11,
 13,
 15,
 17,
 19
```

filter

- dla każdego elementu z listy LISTA, sprawdź czy spełnia on warunek z funkcji WARUNEK.
- WARUNEK to funkcja zwracająca True lub False w poszczególnych, zdefiniowanych przypadkach.
- Jeżeli dla danego elementu wartość zwrócona z funkcji WARUNEK będzie True, element nie zostanie odfiltrowany.
- przetworzone elementy można przypisać do nowej listy,
- bez rzutowania list(filter(…)) funkcja filter zwraca generator.

```
filter(WARUNEK, LISTA)
```

```
new_list = list(filter(WARUNEK,LISTA)
```

map, reduce, filter

Z listy pierwszych dziesięciu liczb naturalnych odfiltruj liczby niepodzielne przez 2 wykorzystując:

a) pętlę for,
b) listę składaną (list comprehension),
c) funkcję filter oraz lambdę.

map, reduce, filter - czy warto stosować?

- Obecnie używanie funkcji **map**, **reduce** oraz **filter** wychodzi z mody.
- Wynika to z potęgi wyrażeń typu comprehension (list, set, dict comprehensions).
- Większość wyrażeń napisanych przy użyciu **map**, **reduce** i **filter** da się zapisać za pomocą comprehensions, które są bardziej idiomatyczne w Pythonie.
- Dlatego też warto się zastanowić zanim użyjemy tych funkcji.

Funkcje sorted, min, max

- Wbudowane funkcje **sorted**, **min** oraz **max** poznaliśmy znacznie wcześniej.
- O czym nie wspomnieliśmy to fakt, że wszystkie trzy przyjmują opcjonalny argument o nazwie **key**.
- **key** jest funkcją, za pomocą której zostaną przekształcone elementy kolekcji, na których liczymy **min**, **max** czy **sorted**. Funkcja **min** zwróci ten element, którego wartość **key(element)** jest najmniejsza. Funkcja sorted również uporządkuje elementy według ich klucza.
- Domyślnie kluczem jest funkcja tożsamościowa, przekształcająca element na samego siebie.
- Czasami jednak mamy listę par i chcielibyśmy wybrać element, który ma najmniejszy drugi element pary albo posortować obiektu klasy **Employee** po imieniu.
- Do tego przyda nam się parameter **key**, do którego zazwyczaj przekazujemy wyrażenie lambda.

```
In [1]: pairs = [(1, 10), (2, 9), (3, 8)]

In [2]: min(pairs)
Out[2]: (1, 10)

In [3]: max(pairs)
Out[3]: (3, 8)

In [4]: min(pairs, key=lambda x:x[1])
Out[4]: (3, 8)

In [5]: max(pairs, key=lambda x:x[1])
Out[5]: (1, 10)

In [6]: min(pairs, key=lambda x: x[0] * x[1])
Out[6]: (1, 10)

In [7]: max(pairs, key=lambda x: x[0] * x[1])
Out[7]: (3, 8)
```

Funkcje sorted, min, max

Zaimplementuj własną funkcję max (nazwij ją maximum), która przyjmuje dwa argumenty: listę oraz klucz, zgodnie z którym ma wyszukiwać największą wartość w liście. Obydwa parametry są obowiązkowe.