**Dekoratory**

Tworzenie i wykorzystanie.

Zacznijmy od uświadomienia sobie, że dekoratory możemy zarówno tworzyć od zera, jak i wykorzystywać już stworzone i wbudowane w standard języka.

Służą one bowiem do rozszerzania funkcjonalności metod. Wszędzie tam, gdzie zauważysz zapis @keyword to właśnie przykład dekoratora. Co więcej, warto również wiedzieć, że wykorzystanie dekoratorów zalicza się do **technik metaprogramowania**.

W skrócie - metaprogramowanie polega na na modyfikowaniu działania programu **już w trakcie jego kompilacji**. Może to służyć do **przenoszenia obliczeń z czasu wykonywania na czas kompilacji**, do **generowania kodu przy użyciu obliczeń czasu kompilacji** oraz do włączania **samomodyfikującego się kodu**.

**Dekoratory w życiu codziennym**

Zanim przejdziemy do meritum, chciałem Ci zaprezentować najpopularniejsze przykłady wbudowanych w **standard dekoratorów**. Z racji, że do tej pory nie rozszerzyliśmy tego tematu, to odniesiemy się głównie do paradygmatu programowania OOP.

* @classmethod

Dekorator ten wykorzystujemy, gdy chcemy powiązać **daną metodę z samą klasą**, aniżeli **z obiektem danej klasy**.

Metody poprzedzone dekoratorem @classmethod nie potrzebują bowiem do wywołania żadnej instancji klasy. Jednak drugą stroną medalu jest, że przez to, iż nie odnoszą się do klasy - jako całościowego bytu - **nie możemy się z ich poziomu odwoływać ani do pól, ani do innych metod składowych przy użyciu keyword self!**

**Syntax:**

@classmethod

def func(cls, args...)

cls - klasa (nie obiekt), na rzecz którego func została wywołana

args - dowolne argumenty metody

**Zastosowanie:**

Metoda opatrzona dekoratorem @classmethod służy przede wszystkim do **tworzenia nowych obiektów** (istnieje tutaj dość duża korelacja między, tzw. **factory methods**).

Prostym przykładem jej zastosowania może być klasa **Worker**, która będzie posiadała właśnie metodę z ustawionym dekoratorem @classmethod i służyć będzie do wytwarzania nowych obiektów o określonych cechach.

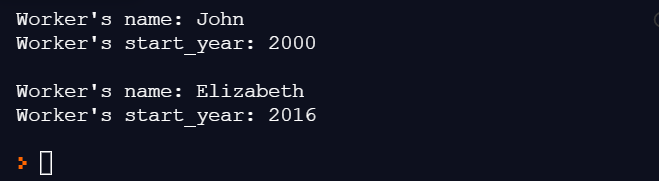
Załóżmy, że tworzona klasa posiadać będzie takie pola jak: **name** oraz **start\_year** (rok rozpoczęcia pracy).

Co do kluczowego zagadnienia – pod dekoratorem @classmethod, stworzymy metodę, która przyjmować będzie **argument klasowy (cls)**, imię pracownika oraz lata, ile już pracuje na danej posadzie (args).

Na podstawie tych informacji zadaniem metody będzie wytworzenie (fabrykowanie) i zwrócenie nowego pracownika.



**Listing 1**



**Listing 2**

**@staticmethod vs @classmethod**

Czepialscy, na podstawie powyższego podpunktu, mogą stwierdzić, że po co korzystać z @classmethod, skoro istnieje **@staticmethod** (odnosząc się do powyższego przykładu, zasada działania obu dekoratorów jest ta sama, jednak staticmethod **nie przyjmuje cls jako pierwszego argumentu**).

W porządku, ale idąc krok dalej, @classmethod jest o wiele bardziej **uniwersalnym rozwiązaniem**, gdy chcemy tworzyć "statyczne metody" (czyli choćby metody fabrykujące) w danej klasie.

Robi on bowiem zasadniczą różnicę w momencie, gdy **wykorzystujemy go w kontekście dziedziczenia**.

Okazuje się, że mając stworzoną metodę fabrykującą w klasie rodzicielskiej z dekoratorem @staticmethod, niemożliwe staje się wytwarzanie klas po niej dziedziczących. Czyli rodzic nie może prawidłowo fabrykować swoich dzieci. Konieczne jest bowiem przesyłanie argumentu **cls** (którego de facto w @staticmethod nie ma) – czyli klasy będącej potomkiem, a jedyną taką funkcjonalność zapewnia Nam @classmethod.

**Tworzenie własnych dekoratorów?**

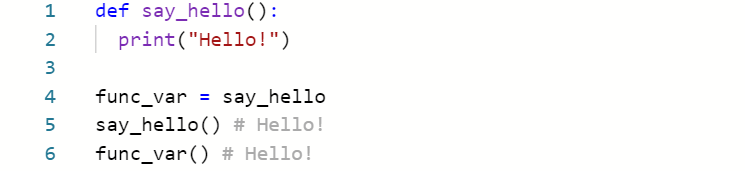
W porządku! Powyżej przedstawiłem Ci przykład dekoratora wbudowanego w standard, ale celem tego szkolenia jest przecież nauczyć Cię jak przyozdabiać metody (lub inne struktury) **własnymi** dekoratorami!

Jednak zanim do tego przejdziemy, aby w pełni zrozumieć poruszane zagadnienie, konieczne jest naświetlenie pewnych cech i charakterystycznych zachowań funkcji.

Wiedziałeś, że funkcja może **zwracać referencje do innej funkcji**? Co więcej, funkcje możemy **przekazywać jako argumenty**, a także **przypisywać je do dowolnej zmienne.**

Pełna dowolność! Dlatego mówimy, że funkcje w Pythonie są przykładami **obiektów pierwszej klasy**. Możemy nimi manipulować tak, jak obiektami dowolnej innej klasy!

**Funkcja przypisana do zmiennej:**



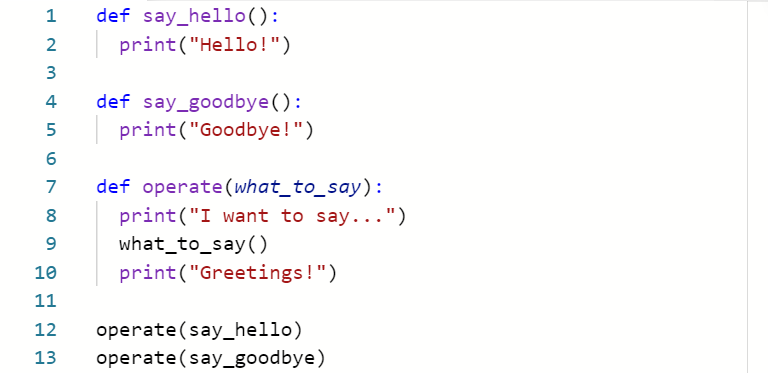
**Listing 3**

W wyniku powyższego kodu, na ekranie pojawiają się dwa te same napisy w wyniku dwukrotnego wywołania funkcji **say\_hello.**

W **linii 5.** wywołujemy ją **bezpośrednio**, a w **linii 6.** przez zdefiniowaną zmienną **func\_var**.

**Funkcja jako argument**

Możemy również przekazywać referencje do innych funkcji w postaci argumentów.



**Listing 4**

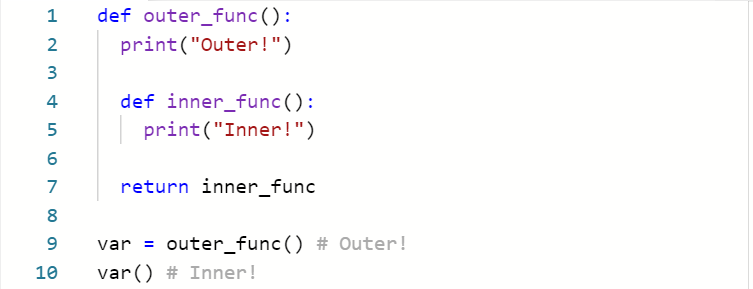


**Listing 5**

**Referencja do funkcji jako argument zwracany**

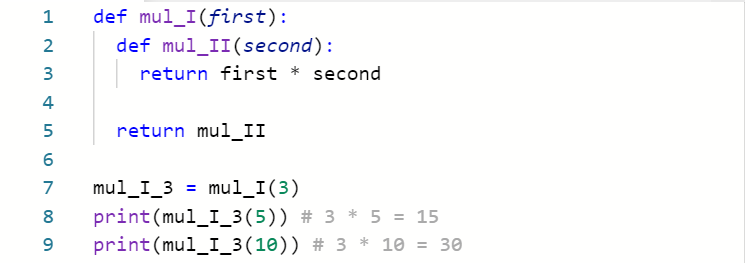
I przed Nami najważniejszy przykład! Funkcja może także zwracać **referencję do dowolnej funkcji**. I to jest bez wątpienia Nasz gwóźdź programu niezbędny do zrozumienia zasad budowania własnych dekoratorów!

**Przykład 1:**



**Listing 6**

**Przykład 2:**

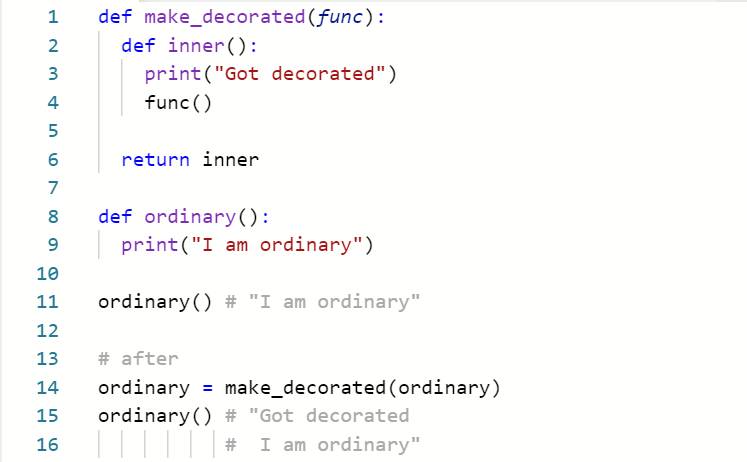


**Listing 7**

**Tworzenie pierwszego dekoratora**

Tworzenie dekoratorów jest ściśle powiązane z powyższymi przykładem, bo opiera się właśnie na zwracaniu przez funkcję (dekoratora) referencji do innej funkcji (która określa jego funkcjonalność).

Zapoznajmy się z przykładem, który implementuje dekorator "od podstaw".



**Listing 10.**

Spójrz na **linię 14.** To tutaj dzieje się cała magia. Właśnie w tym momencie dodajemy do funkcji ordinary stworzony przez Nas dekorator (co prawda, jeszcze w dość niekoniecznie zgrabny sposób, ale czytaj dalej…).

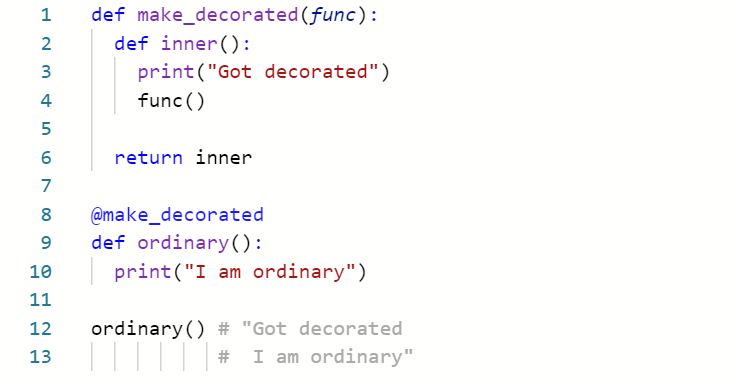
W wyniku drugiego output'u, widzimy, że dzięki 14. linii, rozszerzyliśmy funkcję ordinary o stworzony dekorator, który dodaje przed komunikat **"I am ordinary"**, tekst **"Got decorated".**

Nasuwa się więc ważny wniosek - dekoratory są po prostu dodatkami, opakowaniami, rozszerzającymi zbudowaną funkcjonalność jakiejś metody (funkcji).

Co więcej po powiązaniu dekoratora z daną funkcją, nie musimy już myśleć o tym, aby był on wywoływany w odpowiednich momentach. Cała procedura odbywa się automatycznie!

Idąc krok dalej, możemy jeszcze bardziej uprościć powyższy przykład. A to właśnie przez zastosowanie **@nazwa\_dekoratora** umieszczanego **nad dekorowaną funkcją**.

Po szybkiej zmianie, kod przyjmie postać:



**Listing 11**

Efekt jest taki sam, a zobacz jak o wiele czytelniejszy i krótszy stał się Nasz program!

**Dodanie dekoratora nad funkcję (linia 8.) zastępuje całą linię 14. z Listing'u 10! Kod staje się zgrabniejszy i bardziej czytelny!**

**Własny dekorator z argumentami**

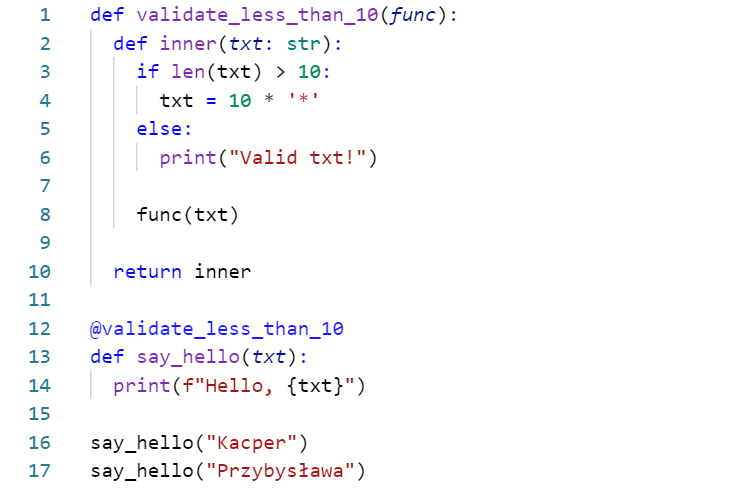
Jak już wiesz, dekoratory rozszerzają zachowanie dowolnej funkcji. Przyznaj, że jest to niezwykle przydatny Python-owy feature. Raz mądrze stworzony dekorator może zostać użyty niezliczoną ilość razy (spójrz choćby na powszechność i szerokie wykorzystanie przez programistów dekoratora @classmethod).

Pokażę Ci pierwszy praktyczniejszy przykład z wykorzystaniem własnoręcznie zaimplementowanego dekoratora.

Wyobraź sobie, że musisz napisać program, który będzie przyjmował od użytkownika imię. Program ma wyświetlać powitanie w stylu "Hello, {imię}".

Z racji jednak tego, że program chcemy napisać na wyświetlacze o rozdzielczości 17-znakowej (gdzie Hello,\_ wykorzystuje już 7 znaków), musimy wprowadzić walidację długości imienia i w przypadku, gdy będzie ono miało więcej niż 10 liter, chcemy zamiast niego wyświetlać 10 gwiazdek.

Rzecz jasna, możemy rozwiązać ten problem, tworząc metodę walidującą, ale nie ładniej byłoby wykorzystać poznane dekoratory?



**Listing 12**



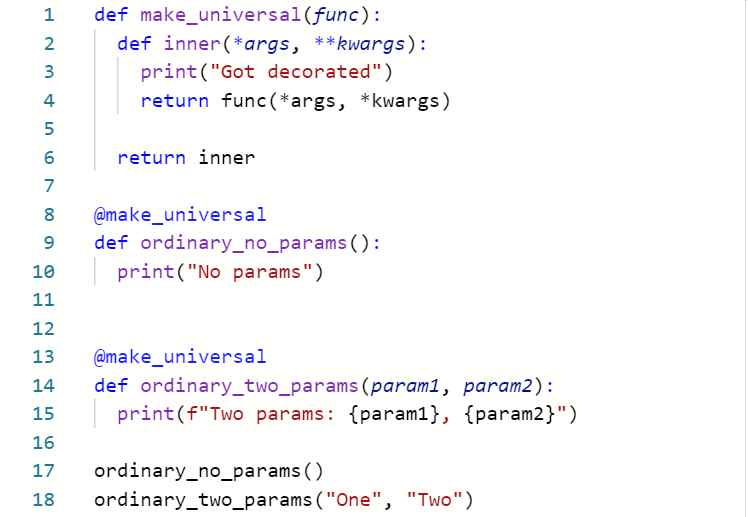
I w ten oto sposób, gdybyśmy chcieli, np. dodać funkcję wyświetlającą dowolnie inny komunikat i zabezpieczyć ją przed naruszeniem ilości możliwych do wyświetlenia znaków, wystarczyłoby przyozdobić ją stworzonym dekoratorem! Teraz już musisz przyznać mi rację, że dekoratory są bardzo praktyczne!

**Kolejne ulepszenia w kodzie**

W celu maksymalnego dopieszczenia kodu, pokażę Ci sztuczkę, które zapewni wprowadzenie jeszcze bardziej uniwersalnego postępowania.

Wystarczy parametry wewnętrznej funkcji dekoratora zamienić na **\*args** oraz **\*\*kwargs**. Wówczas będziemy mogli udekorować funkcje o **dowolnej ilości parametrów** i nie będziemy musieli **ograniczać się do konkretnej** sygnatury.

Powiem Ci więcej - w 90% przypadków będziesz spotykał się właśnie z wewnętrznymi metodami dekoratorów w takiej postaci. Dlatego nie ma żadnych przeciwwskazań, żebyś Ty również korzystał z takiego rozwiązania.

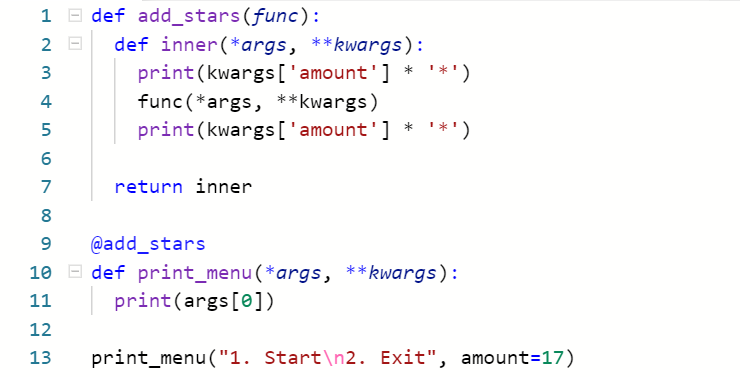


**Listing 13**



Zauważ, że zmian dokonaliśmy w **linii 2**. oraz **linii 4.**

**Przykład rysowania MENU z ramką:**



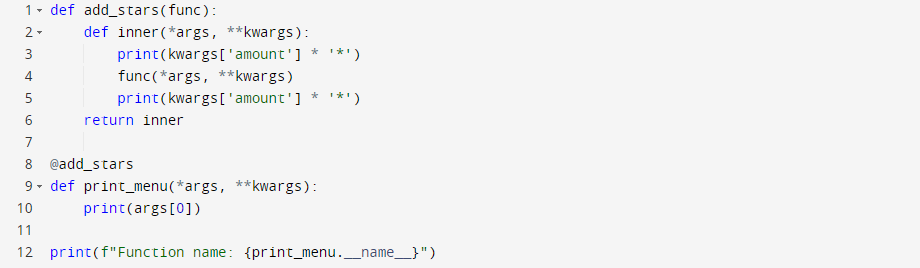
**Listing 14**



**@wraps w dekoratorze**

Sekcja ta zwieńczy dotąd zdobytą wiedzę i zaprezentuje Ci dość nieoczywisty problem pojawiający się przy pracy z dekoratorami. Jest nim mankament związany z przesłanianiem nazwy udekorowanej funkcji przez funkcję wewnętrzną w obrębie dekoratora.

Przeanalizujmy to na przykładzie (na ruszt weźmiemy wcześniej stworzony dekorator add\_stars):



Przykład ten nie różni się od poprzedniego, tak więc ograniczę tłumaczenie jego działania. Jedynie nową linią, która pojawiła się w kodzie jest print, który ma na celu wyświetlenie nazwy funkcji print\_menu. Taki zapis może choćby służyć do tworzenia dokumentacji kodu, czy wyświetlania użytkownikowi nazw wszystkich dostępnych w kodzie funkcji.

Pojawia się jednak problem… Po uruchomieniu kodu wyświetla się następujący komunikat:

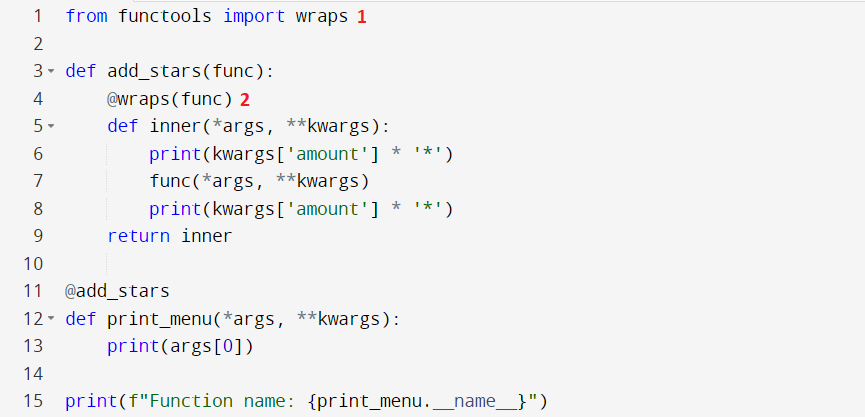


A oczekiwanym wynikiem jest przecież print\_menu (taką nazwę ma przecież funkcja, którą udekorowaliśmy i o której informacje chcemy wyświetlać!).

Spowodowane jest to faktem, że funkcja jest udekorowana przez dekorator, w którym znajduje się właśnie inner i to on jest wstawiany jako reprezentacja funkcji, do której się odnosimy. Możemy zatem powiedzieć, że w naszym kodzie funkcja oryginalna traci niektóre ze swoich właściwości.

Pytanie, jakie sobie powinniśmy zadać to: *“Jak pisać dekoratory, aby nie powodowały one nadpisania właściwej nazwy ozdobionej nimi funkcji?”*. Odpowiedź jest prosta - użyć dekorator @wraps i umieścić go nad wewnętrzną funkcją inner!

Kod po modyfikacji przyjmie wówczas następującą postać:



1. Aby użyć dekorator wraps - musimy zaimportować go z biblioteki functools
2. Nad funkcją inner umieściliśmy zaimportowany dekorator i przekazaliśmy do niego referencję do funkcji func.

W wyniku takiej zmiany, print da teraz następujący efekt:



Czyli efekt zgodny z oczekiwaniami.