**Programowanie funkcyjne**

Python nie jest zaliczany do grona funkcyjnych języków programowania, ale łączy ich niektóre rozwiązania z **paradygmatem programowania obiektowego**.

Dzięki temu nie piszemy w pełni funkcyjnego kodu, lecz przeplatamy ze sobą kilka paradygmatów i dzięki temu możemy jeszcze efektywniej tworzyć kod i implementować nowe funkcjonalności.

**Programowanie funkcyjne vs obiektowe**

Z punktu widzenia języków programowania, to **Python/C#/Java** itp. są językami **imperatywnymi (a dokładniej językami zorientowanymi obiektowo)**, klasyfikowanymi również jako **języki III poziomu**.

Gdybyś był ciekawy, to do grona języków funkcyjnych (deklaratywnych) możemy choćby zaliczyć **SQL/Haskell (języki IV poziomu)**.

**Czym jest kod imperatywny?**

Imperatywny kod to ten, z którym spotykamy się najczęściej, gdy uczymy się programować. Tworzymy w nim bowiem ciąg instrukcji, który ma zostać wykonany przez program i **spodziewamy się określonych efektów**.

Wchodząc głębiej i odnosząc się do programowania obiektowego w imperatywnym podejściu, mamy do czynienia głównie z **klasami i ich obiektami.** Pisząc program, zmieniamy często stany różnych obiektów, wywołując na nich określone metody, czy też modyfikując ich pola.

Podchodząc do programowania funkcyjnego, **nie myślimy** już o obiektach.   
W programowaniu tym, główną rolę pełnią bowiem funkcje.

Zasadniczą różnicą między podejściem imperatywnym a funkcyjnym, jest to, że w tym pierwszym przypisujemy wartości zmiennym i mutujemy je, a w funkcyjnym zwracamy wartość w **bezstanowy sposób** (nie ingerujemy w oryginalne wartości).

Co więcej, output w programowaniu funkcyjnym jest **niezmienny** dla tych samych danych na wejściu.

Aby lepiej zrozumieć różnice i wyciągnąć najważniejsze wnioski, zapoznaj się z poniższą tabelą:

| **Programowanie obiektowe** | **Programowanie funkcyjne** |
| --- | --- |
| Opisujesz, co ma zostać zrobione w danym momencie | Opisujesz, co chcesz osiągnąć |
| Za pomocą metody możemy zmieniać dowolne stany obiektów w programie | Wykonywane funkcje nie zmieniają stanu danych w programie; dążymy do jak najmniejszej ilości skutków ubocznych |
| Typy mutowalne | Typy niemutowalne - dane nie zmieniają się po wykonaniu operacji, zwracana jest ich kopia |
| Wykorzystuje paradygmat programowania imperatywnego | Wykorzystuje paradygmat programowania deklaratywnego |
| Najważniejszą rolę pełnią dane | Najważniejszą rolę nie pełnią dane, ale funkcje |
| Wyrażenia wykonują się w odpowiedniej kolejności, np. z góry do dołu | Wyrażenia nie wykonują się w określonej kolejności |
| Możliwa enkapsulacja danych; bezpieczeństwo przetwarzania | Brak enkapsulacji danych; brak bezpieczeństwa przetwarzania |

**Lambdy**

Aby gładko wejść w świat programowania funkcyjnego, możemy zacząć od poznania, tzw. **funkcji lambda** w Pythonie. Nazywane są one również **funkcjami anonimowym**i.

Gdybyś zapytał, jaka jest różnica między normalnymi funkcjami a funkcjami lambda, to oprócz wcześniej wymienionych różnic, różnią się również tym, że lambdy **nie mają nazw** oraz ich utworzenie **zajmuje dużo mniej linii kodu**.

**Kilka cech charakterystycznych:**

* funkcje lambda mogą przyjmować dowolną ilość argumentów, ale wykonują tylko **jedną konkretną operację**. Lambdy nie modyfikują nigdy przesłanych wartości, tylko zwracają ich zmodyfikowane kopie.
* zapisanie lambdy jest o wiele krótsze niż budowa jej odpowiednika w postaci funkcji

**Tworzenie lambdy**

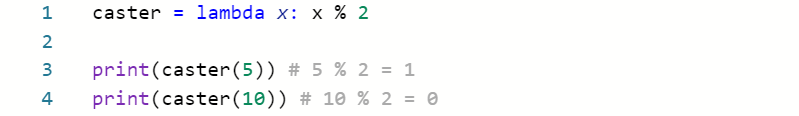
Aby móc utworzyć funkcję lambda, należy skorzystać z zapisu:

lambda argument(y): wyrażenie

Zanim przystąpimy do implementowania pierwszej funkcjonalności, zwróć uwagę, że zgodnie z tym, co wspomnieliśmy - lambda może zawierać **tylko jedno** wyrażenie.

Nie jest możliwe więc umieszczenie w niej żadnych instrukcji warunkowych, które zmieniałyby przepływ danych w zależności od inputu.

**Przykład 1:** *Lambda 1-argumentowa*



**Listing 1**

W listingu 1., lambda x: x % 2 jest omawianą funkcją programowania funkcyjnego.

Zmienna **x jest jej argumentem**, a **x % 2 wyrażeniem**. Jak widzisz, w wyniku wykonania **linii 3. i 4.**, otrzymujemy odpowiednie wyniki operacji obliczania reszty.

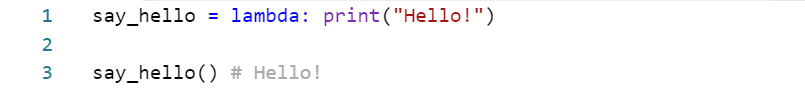
Co istotne, lambda nie potrzebuje **żadnego słowa kluczowego w postaci** **return** (tak jak robimy to w programowaniu imperatywnym), aby zwrócić wynik.

Po prostu jej naturą jest, że automatycznie otrzymujemy dany rezultat.

Zauważ również, że stworzona przez Nas lambda nie posiada żadnej nazwy (cecha funkcji anonimowej).

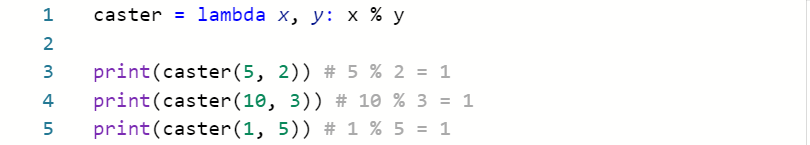
Dlatego, by móc się do niej odwoływać, konieczne było przypisanie jej do zmiennej **caster** w linii 1.

**Przykład 2:** *Lambda bez argumentów*

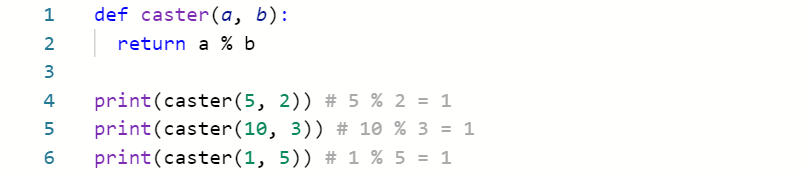


**Listing 2**

**Przykład 3:** *Lambda 2-argumentowa*

**Listing 3**

**Przykład 3.1:** *To samo, co w przykład 3, ale wykorzystując standardową funkcję*



**Listing 3.1**

Oczywiste jest, że powyższe proste operacje możemy równie dobrze wykonać przy użyciu zwykłych funkcji, ale wtedy kod stanie się dłuższy.

Ponadto, przyznaj, że lambdy zapewniają dużą czytelność i łatwe rozeznanie w kodzie. Bądź jednak świadomy tego, że nie zawsze jest możliwa migracja z rozwiązania przy wykorzystaniu standardowych funkcji na lambdy i na odwrót.

Nie będąc też świadomym, kiedy najlepiej wykorzystywać nowo poznany paradygmat programowania, łatwo stworzyć mętlik w kodzie.

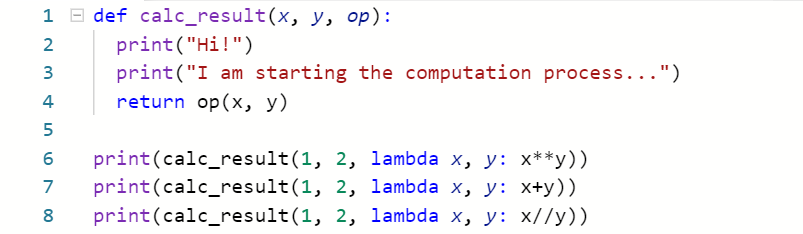
Dlatego czytaj dalej! W następnej sekcji odpowiadam, kiedy zrezygnowanie z zwykłych funkcji na rzecz lambd, sprawdzi się najbardziej.

**Kiedy stosować lambdy?**

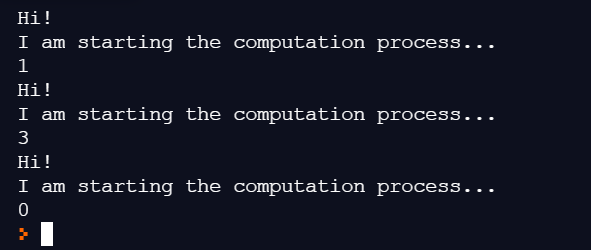
O tym, czy w danym rozwiązaniu lepiej jest użyć podejście imperatywne czy funkcjonalne, może decydować fakt, czy daną funkcję potrzebujemy tylko na **krótki okres czasu**.

Lambdy bowiem są szczególnie często wykorzystywane, gdy chcemy łączyć ze sobą jakieś funkcjonalności i dawać użytkownikowi możliwość łatwego sterowania programem. Na przykład w momencie, gdy chcemy kontrolować przebieg funkcji **higher order** (funkcja higher order cechuje się tym, że zwraca lub przyjmuje jako argument inną funkcję lub lambdę).

**Przykład 1:** Lambda jako argument funkcji wykonującej określone operacje



**Listing 4**



Co ciekawe, twórcy Pythona wykorzystali mniej więcej powyżej zaprezentowany sposób na kierowanie przebiegiem programu i zaimplementowali **"higher-order methods"**, które właśnie opierają się na otrzymywaniu wyrażeń lambda w postaci argumentów.

Wpisane już w kanwę programów wykorzystujących paradygmat programowania deklaratywnego i imperatywnego w Pythonie, metody te to:

* **map()**
* **filter()**
* **reduce()**

To właśnie w dalszej części tego tematu, rozważymy, jak funkcje lambda są wykorzystywane przy niektórych wbudowanych metodach w Pythonie.

**IIFE**

Rozwijając skrót IIFE, otrzymujemy **immediately invoked function execution**, czyli natychmiastowo wywoływane funkcje.

Potraktuj ten podpunkt jako tylko wtrącenie, ale chciałbym Cię uświadomić, że jest możliwe wywoływanie funkcji lambda zaraz po jej zdefiniowaniu, np.

(lambda x: x + x)(2) # 4

Wspominam o tym, ponieważ jest to wykorzystywane we wspomnianych metodach i umożliwia **automatyczne wywołanie lambdy** w metodzie **map, reduce czy filter**.

Co prawda, nie musiałem o tym wspominać, bo wszystko to dzieje się w kodzie pod przykrywką i IIFE jest wykonywane w tym przypadku niejawnie, ale chcę budować w Tobie jak największą świadomość programisty i poruszać wszystkie szczegółowe zagadnienia.

**Filter()**

Metoda ta służy do **filtrowania elementów** dowolnej struktury danych (lista, tuple zbiory etc.) według **pewnego warunku** (możliwego do zdefiniowania w lambdzie).

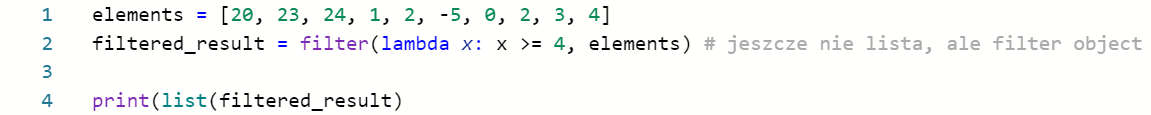
Syntax:

filter(func, iterable)

Omawiana metoda przyjmuje 2 argumenty:

* Funkcję filtrującą (którą będziemy wyrażali przez lambdę)
* Dowolną sekwencję danych

**Przykład 1:** Przefiltruj listę w poszukiwaniu liczb większych lub równych 4.



**Listing 5**



**Wyjaśnienie:**

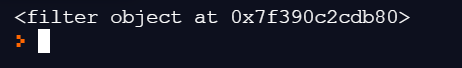
1. Linia 1 to definicja listy o elementach będącymi dowolnymi liczbami całkowitymi.

2. Linia 2 to utworzenie zmiennej filtered\_result, która przechowywać będzie obiekt **<filter result>** zwracany przez **filter(lambda x: x >= 4, elements)**.

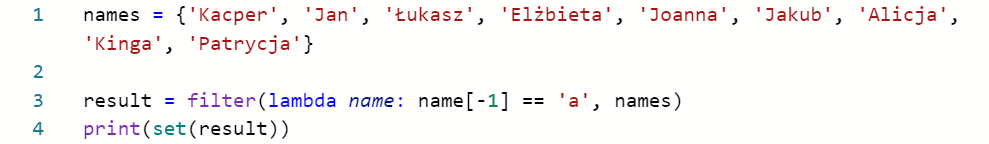
3. Metoda filter przyjmuje pierwszy argument w postaci wyrażenia lambda (nazywane też **predykatywą**, ponieważ służy do zwracania wartości **True/False** w zależności od warunku).  
Drugim argumentem jest utworzona lista, której wartości chcemy przefiltrować według przekazanej lambdy.

4. Tak więc filter wyłuska z listy elements tylko te liczby, dla których **lambda x: x >= 4** zwróci **True** (wszystkie elementy większe lub równe 4). W tym przypadku parametr lambdy nazwaliśmy jako x, ale równie dobrze może ona przyjąć postać lambda **val**: val >= 4 etc.

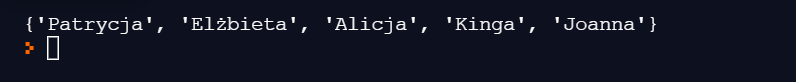
5. Linia 4. to wydrukowanie na ekranie rezultatu w postaci listy. Co jest tutaj bardzo ważne, to fakt, że chcąc otrzymać czytelną formę przefiltrowanych wartości, obiekt **filtered\_result** musimy przekonwertować na listę!   
W przeciwnym razie, zapisana tak linia 4: **print(filtered\_result)** wyświetliłaby:



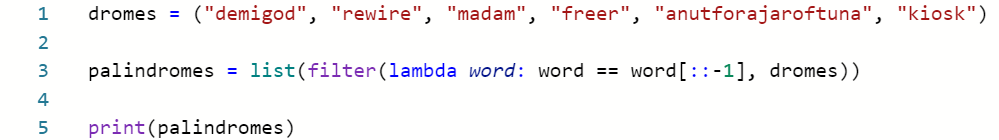
**Przykład 2:** *Wybierz z zbioru unikalne imiona kończące się na literę 'a'.*



**Listing 6**



**Przykład 3:** *Wyświetl palindromy z podanej listy.*



**Listing 7**



**Map()**

Metoda, którą właśnie zaczynamy omawiać ma następujący syntax:

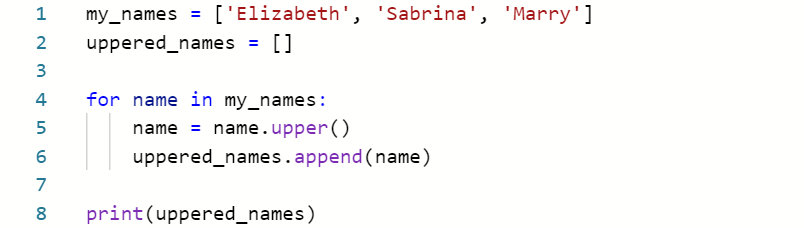
map(func, \*iterables)

Func to dowolny rodzaj funkcji (my użyjemy lambdy), która zostanie wykonana na rzecz obiektów (lub obiektu) przesłanych w postaci **\*args** - **\*iterables**.

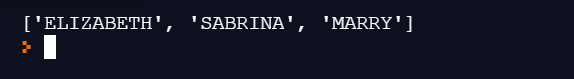
Tak jak w przypadku metody **filter()**, map() zwraca **obiekt generatorowy**, dlatego konieczne będzie jego **rzutowanie na odpowiedni typ**, np. **list(map(func, \*iterables)).**

Map ma proste zastosowanie. **Służy do kopiowania i modyfikowania** (np. zmienienia wszystkich liter na wielkie) elementów obiektów iterables.

**Przykład 1:** Zamiana wszystkich liter elementów z listy na wielkie. **Standardowe podejście.**

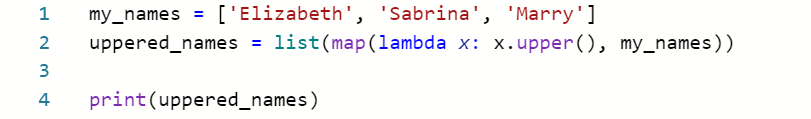


**Listing 8**

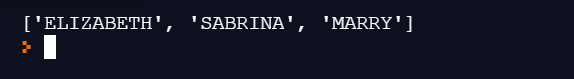


Powyższy przykład można jednak rozwiązać w o wiele łatwiejszy sposób. Możemy bowiem skrócić implementację powyższej funkcjonalności przez zastosowanie map(). Dzięki temu otrzymamy bardziej przejrzysty i elastyczny kod.

**Przykład 1.1:** Zamiana wszystkich liter elementów z listy na wielkie. Podejście z map().



**Listing 9**



W powyższym przykładzie wysłaliśmy tylko jeden obiekt - **my\_names** - dla argumentu **\*iterables**.

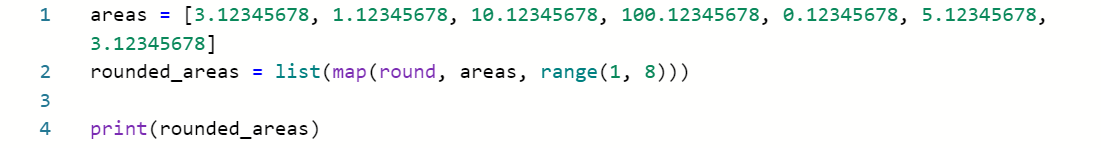
Przykład dość trywialny, widzimy, jak map w połączeniu z lambdą sprawnie zwróciło Nam do listy **uppered\_names zmodyfikowane wartości.**

Potencjał map() sprawdzić się może jednak najbardziej w momencie, gdy chcemy przeprowadzić pewne operacje dla kilku obiektów równocześnie.

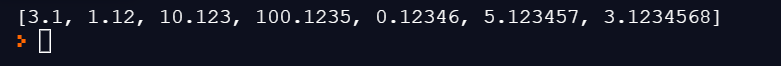
Ponadto, niekoniecznie musimy definiować własną funkcję (lub lambdę), bo możemy podać referencje do metody wbudowanej, tzw. built-in.

Już spieszę z przykładem!

**Przykład 2**: *Mając listę wartości typu float, przeprowadź proces zaokrąglania wartości do jednego miejsca po przecinku dla 1. wartości z listy, dwóch miejsc po przecinku dla 2. wartości z listy, trzech miejsc po przecinku dla 3. elementu itd.*



**Listing 10**



Zdziwiony, że to zadziałało? To właśnie jedna z potężnych możliwości metody map!

**Wyjaśnienie:**

Map() w powyższym przykładzie otrzymało w sumie **3 argumenty**. Pierwszy argument to **referencja do wbudowanej metody round** (która na dobrą sprawę mogłaby być dowolną inną funkcją 3-argumentową).

Następne argumenty to **\*iterables,** czyli **lista areas** oraz **range(1, 8)**. I choć odbywa się to niejawnie, to ich elementy przekazywane są do pierwszego argumentu map, czyli metody **round**.

Co więcej, iteracja po **\*iterables** następuje **automatycznie**. Czyli do metody round przekazywany jest kolejno element **pierwszy z areas**, a wraz z nim **wartość 1** z range(1, 8).

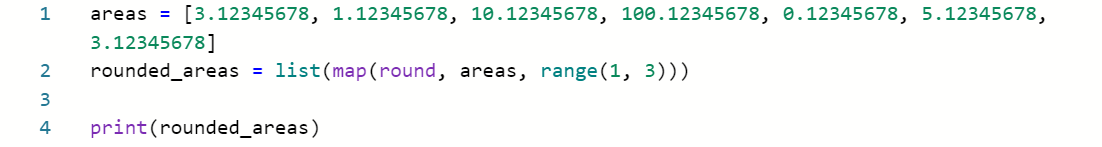
Następuje zaokrąglanie, wartość jest zapamiętywana i map przechodzi do następnych elementów - wywołuje metodę **round(areas[1], 2)**, potem **round(areas[2], 3)** itd.

Na pewno zastanawiasz się, a co w momencie, gdybyśmy przekazali do map obiekty o **różnej ilości elementów**?

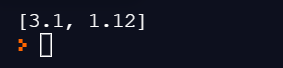
Jak mogliśmy zauważyć, przechodzenie po elementach odbywało się równolegle i pojawia się zagwostka, co się wydarzy, gdy prześlemy, np. niezmienioną **listę areas,** ale inny przedział, bo w postaci **range(1, 3)**?

Odpowiedź jest prosta: zmapowana zostanie taka ilość, jaka odpowiada długości **krótszego** obiektu, po którym następuje równoległa iteracja.

**Przykład 2.1:** *Argumenty \*iterables różnej długości przesyłane do metody map().*



**Listing 10.1**



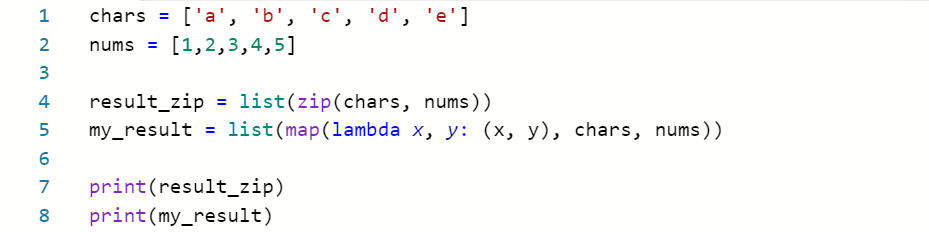
Na podsumowanie jeszcze jeden przykład:

**Przykład 3:** *Map() służące do zaimplementowania własnej funkcji* ***zipującej*** *dwie dowolne listy.*

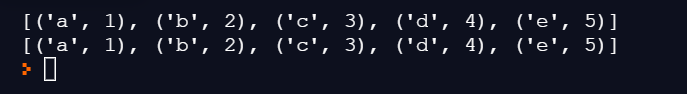
**Przypomnienie:**

Zip() jest już metodą wbudowaną w standard Pythona, ale na rzecz ćwiczenia, spróbujemy zaimplementować ją od podstaw. Metoda ta służy **do łączenia kolejnych elementów dwóch struktur danych**, np.

list(zip([1, 2, 3], [‘a’, ‘b’, ‘c’])) == [(1, ‘a’), (2, ‘b’), (3, ‘c’)]



**Listing 11**



**Reduce**

I na koniec metoda służąca do **redukowania (zamieniania)** dowolnego obiektu, po którym możemy iterować, na **pojedynczą wartość**.

Jaka to będzie wartość i od czego ona zależy? To wartość obliczana w wyniku dowolnej operacji przeprowadzonej na wszystkich elementach redukowanego obiektu (np. sumowanie, odejmowanie, mnożenie).

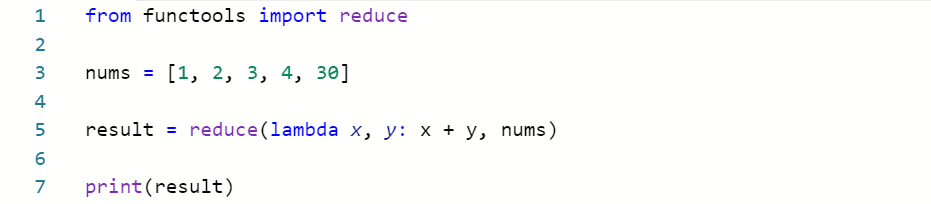
Ponadto, reduce pozwala określić wartość opcjonalnego argumentu **initial** i w ten sposób rozpoczynać operacje od dowolnej, skumulowanej już, wartości.

Pamiętaj jednak, że w przeciwieństwie do omówionych wcześniej przykładów, konieczne jest zaimportowanie metody reduce z modułu **functools**.

Syntax:

| from functools import reduce  reduce(func, iterable, initial=optional) |
| --- |

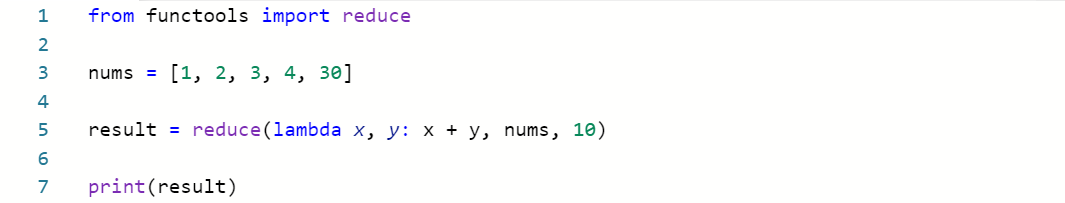
**Przykład 1**: *Własna implementacja metody sum().*



Listing 12



**Przykład 1.1:** *To samo, ale z argumentem initial.*



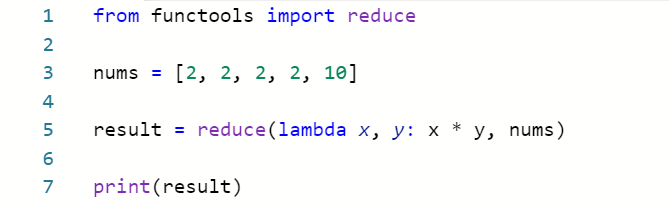


**Wyjaśnienie:**

Reduce pobiera kolejne elementy z listy nums i przekazuje je do lambdy, która sumuje kolejne pary. Domyślnie sumowanie rozpoczyna się od zwiększania wartości 0, w przykładzie 1.1 zmieniamy wartość **startową wyniku na 10**.

Proces pobierania i sumowania kolejnych elementów odbywa się do momentu, kiedy nie przejdziemy po wszystkich elementach w rozpatrywanej liście nums.

**Przykład 2:** *Obliczenie iloczynu wszystkich elementów listy.*



**Listing 13**

To tyle! Jestem przekonany, że po przeanalizowaniu powyższej teorii, będziesz w stanie gładko wejść w temat programowania funkcyjnego. Do poznanych metod map() i filter() odniesiemy się jeszcze w sekcji poświęconej listom składanym, także nie idź dalej z materiałem, dopóki nie będziesz pewny, że w pełni zrozumiałeś omówione zagadnienia!   
: )