Poniższy rozdział pochodzi z książki:

Charles R. Severance - "Python dla wszystkich: Odkrywanie danych z Python 3"

Pełna wersja podręcznika znajduje się na stronie https://py4e.pl/book

# Rozdział 4

# Funkcje

## 4.1. Wywoływanie funkcji

W kontekście programowania *funkcja* to nazwana sekwencja instrukcji, która wykonuje pewne obliczenia. Gdy definiujesz funkcję, określasz jej nazwę i kolejność instrukcji. Później możesz "wywołać" funkcję po jej nazwie. Widzieliśmy już przykłady *wywołania funkcji*, np.:

```
>>> type(32)
<class 'int'>
```

Nazwa funkcji to `type()`. Wyrażenie w nawiasach nazywane jest *argumentem* funkcji. Argument jest wartością lub zmienną, którą przekazujemy do funkcji jako jej dane wejściowe. Wynikiem dla funkcji `type()` jest typ przekazanego argumentu.

Zwykle mówi się, że funkcja "przyjmuje" argument i "zwraca" wynik. Wynik jest nazywany *wartością zwracaną*.

## 4.2. Funkcje wbudowane

Python zapewnia szereg ważnych funkcji wbudowanych, z których możemy korzystać bez konieczności podawania definicji tych funkcji. Twórcy Pythona napisali zestaw funkcji do rozwiązywania popularnych problemów i dołączyli je do Pythona, tak abyśmy mogli z nich korzystać.

Funkcje `max()` i `min()` dają nam odpowiednio największe i najmniejsze wartości występujące w liście:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

Funkcja `max()` mówi nam o "największym znaku" w napisie (który okazuje się być literą "w"), a funkcja `min()` pokazuje nam "najmniejszy znak" (który okazuje się być spacją).

Inną bardzo często używaną funkcją jest funkcja `len()`, która mówi nam, ile elementów znajduje się w przekazanym argumencie. Jeśli argument przekazany do funkcji `len()` jest napis, to zwraca liczbę znaków w tym napisie.

```
>>> len('Hello world')
11
>>>
```

Funkcje te nie ograniczają się do analizowania wyłącznie napisów. Mogą one działać na dowolnym zestawie wartości, jak zobaczymy w późniejszych rozdziałach.

Nazwy wbudowanych funkcji należy traktować jako słowa zastrzeżone (tzn. unikać używania `max` jako nazwy zmiennej).

## 4.3. Funkcje konwersji typu

Python oferuje również funkcje wbudowane, które konwertują wartości z jednego typu na drugi. Funkcja `int()` przyjmuje dowolną wartość i konwertuje ją na liczbę całkowitą (jeśli to możliwe) lub skarży się, jeśli nie wie jak dokonać takiej konwersji:

```
>>> int('32')
32
>>> int('Halo')
ValueError: invalid literal for int() with base 10: 'Halo'
```

Funkcja `int()` może zamieniać wartości zmiennoprzecinkowe na liczby całkowite, ale ich nie zaokrągla (ucina część ułamkową):

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

Funkcja `float()` konwertuje liczby całkowite i napisy na liczby zmiennoprzecinkowe:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Natomiast funkcja `str()` konwertuje swój argument na napis:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

## 4.4. Funkcje matematyczne

Python posiada moduł (bibliotekę) `math`, który dostarcza większość znanych funkcji matematycznych. Zanim będziemy mogli skorzystać z tego modułu, musimy go zaimportować:

```
>>> import math
```

Powyższa instrukcja tworzy *obiekt modułu* o nazwie `math`. Jeśli spróbujesz wydrukować na ekranie obiekt modułu, otrzymasz kilka informacji na jego temat:

```
>>> print(math)
<module 'math' (built-in)>
```

Obiekt modułu zawiera funkcje i zmienne zdefiniowane w tym module. Aby uzyskać dostęp do jednej z tych funkcji, musisz podać nazwę modułu i nazwę funkcji, oddzielone kropką. Format ten nazywa się *notacją kropkową*.

```
>>> signal_power = 0.2
>>> noise_power = 0.000001
>>> ratio = signal_power / noise_power
```

```
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

Pierwszy przykład oblicza logarytm przy podstawie 10 ze stosunku sygnału (`signal_power`) do szumu (`noise_power`) przemnożony przez 10. Moduł matematyczny udostępnia również funkcję o nazwie `log()` ↪ , która oblicza logarytm naturalny.

Drugi przykład znajduje sinus z wartości w zmiennej `radians`. Nazwa zmiennej jest podpowiedzią, że `sin()` i inne funkcje trygonometryczne (`cos()`, `tan()` itp.) przyjmują argumenty w radianach. Aby przeliczyć wartość ze stopni na radiany, należy ją podzielić przez 360 i pomnożyć przez $2\pi$:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

Wyrażenie `math.pi` pobiera zmienną `pi` z modułu matematycznego. Wartość tej zmiennej jest przybliżeniem $\pi$ z dokładnością do około 15 cyfr.

Jeśli znasz trygonometrię, to możesz sprawdzić poprzedni wynik, porównując go z pierwiastkiem kwadratowym z dwóch podzielonym przez dwa:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

## 4.5. Liczby losowe

Na podstawie tych samych danych wejściowych większość programów komputerowych generuje za każdym razem te same dane wyjściowe, więc w takich przypadkach mówi się, że te programy są *deterministyczne*. Determinizm jest zazwyczaj dobrą rzeczą, ponieważ oczekujemy, że to samo obliczenie da ten sam wynik. W przypadku niektórych aplikacji chcemy jednak, by komputer był nieprzewidywalny. Gry są oczywistym przykładem, ale nie jedynym.

Uczynienie programu prawdziwie niedeterministycznym okazuje się wcale nie takie proste, ale są sposoby, by przynajmniej wydawał się niedeterministyczny. Jednym z nich jest użycie *algorytmów*, które generują liczby *pseudolosowe*. Liczby pseudolosowe nie są tak naprawdę losowe, ponieważ są generowane przez deterministyczne obliczenia, ale patrząc na te liczby, nie da się ich odróżnić od losowych.

Moduł `random` oferuje funkcje, które generują liczby pseudolosowe (które od tej chwili będę po prostu nazywał "losowymi").

Funkcja `random()` zwraca losową liczbę pomiędzy 0.0 a 1.0 (włączając w to 0.0, ale nie 1.0). Za każdym razem, gdy wywołujesz funkcję `random()`, otrzymujesz następną liczbę. Aby zobaczyć krótki przykład, uruchom poniższą pętlę:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Powyższy program tworzy następującą listę 10 losowych liczb z zakresu od 0.0 do 1.0 (z wyłączeniem 1.0).

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
```

```
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

**Ćwiczenie 1.** Uruchom program na swoim systemie i zobacz, jakie dostajesz liczby. Uruchom program więcej niż raz i zobacz, jakie tym razem dostajesz liczby.

---

Funkcja `random()` jest tylko jedną z wielu funkcji, które obsługują liczby losowe. Funkcja `randint()` pobiera parametry `low` oraz `high` i zwraca liczbę całkowitą pomiędzy `low` i `high` (włączając w to obydwie te liczby).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Aby wybrać losowo element z jakiejś sekwencji, możesz użyć `choice()`[1]:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

Moduł `random` zapewnia również funkcje do generowania wartości losowych z rozkładów ciągłych, w tym rozkładu Gaussa, wykładniczego, gamma i kilku innych.

## 4.6. Dodawanie nowych funkcji

Do tej pory korzystaliśmy tylko z funkcji dostarczanych wraz z Pythonem, ale możliwe jest również dodawanie nowych funkcji. *Definicja funkcji* określa nazwę nowej funkcji i kolejność poleceń, które wykonują się podczas jej wywołania. Kiedy już zdefiniujemy jakąś funkcję, będziemy mogli używać jej raz za razem w całym programie.

Oto krótki przykład:

```
def print_lyrics():
    print("Jestem sobie drwal i równy chłop.")
    print('Pracuję w dzień i śpię całą noc.')
```

`def` to słowo kluczowe, które wskazuje, że jest to definicja funkcji. Nazwa funkcji to `print_lyrics()`. Zasady dotyczące nazw funkcji są takie same jak dla nazw zmiennych: litery, cyfry i niektóre znaki interpunkcyjne są dozwolone, ale pierwszy znak nie może być liczbą. Nie możesz używać słowa kluczowego jako nazwy funkcji i powinieneś unikać zmiennej i funkcji o tej samej nazwie.

Puste nawiasy po nazwie oznaczają, że ta funkcja nie przyjmuje żadnych argumentów. Później zbudujemy funkcje, które przyjmują argumenty jako swoje dane wejściowe.

Pierwszy wiersz definicji funkcji nazywa się *nagłówkiem* funkcji; reszta nazywa się *ciałem* funkcji. Nagłówek musi się kończyć dwukropkiem, a ciało musi być wcięte. Zgodnie z konwencją, wcięciem są zawsze cztery spacje. Ciało może zawierać dowolną liczbę instrukcji.

Jeżeli wpiszesz definicję funkcji w trybie interaktywnym, interpreter wypisze trzy kropki (...), aby dać Ci znać, że definicja nie jest kompletna:

[1] W poniższym kodzie zmienna `t` jest listą; przyjrzymy się listom bardziej szczegółowo w dalszych rozdziałach.

```
>>> def print_lyrics():
...     print("Jestem sobie drwal i równy chłop.")
...     print('Pracuję w dzień i śpię całą noc.')
...
```

Aby zakończyć funkcję, musisz wpisać pustą linię (w skrypcie nie jest to konieczne).

Zdefiniowanie funkcji tworzy zmienną o tej samej nazwie.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

Wartość `print_lyrics` to *obiekt funkcji* o typie `function`.

Składnia wywoływania nowej funkcji jest taka sama jak dla funkcji wbudowanych:

```
>>> print_lyrics()
Jestem sobie drwal i równy chłop.
Pracuję w dzień i śpię całą noc.
```

Po zdefiniowaniu funkcji, możemy jej używać wewnątrz innej funkcji. Na przykład, aby powtórzyć poprzednią zwrotkę, możemy napisać funkcję o nazwie `repeat_lyrics()`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

Następnie wywołujemy `repeat_lyrics()`[2]:

```
>>> repeat_lyrics()
Jestem sobie drwal i równy chłop.
Pracuję w dzień i śpię całą noc.
Jestem sobie drwal i równy chłop.
Pracuję w dzień i śpię całą noc.
```

## 4.7. Definiowanie i używanie

Po złożeniu fragmentów kodu z poprzedniej sekcji, zauważymy, że cały program wygląda następująco:

```
def print_lyrics():
    print("Jestem sobie drwal i równy chłop.")
    print('Pracuję w dzień i śpię całą noc.')


def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()

# Kod źródłowy: https://py4e.pl/code3/lyrics.py
```

Program ten zawiera dwie definicje funkcji: `print_lyrics()` i `repeat_lyrics()`. Definicje te wykonywane są tak samo jak inne polecenia, ale efektem jest tworzenie obiektów funkcji. Polecenia wewnątrz funkcji są wykonywane dopiero po jej wywołaniu, a definicja funkcji nie generuje danych wyjściowych.

[2] Tak naprawdę piosenka o drwalu idzie zupełnie inaczej.

Jak pewnie się spodziewasz, musisz stworzyć funkcję, zanim będziesz mógł ją wykonać. Innymi słowy, definicja funkcji musi zostać wykonana przed jej pierwszym wywołaniem.

**Ćwiczenie 2.** Przesuń ostatnią linię programu na samą górę, tak aby wywołanie funkcji pojawiło się przed definicjami. Uruchom program i zobacz, jaki komunikat o błędzie otrzymasz.

**Ćwiczenie 3.** Przesuń wywołanie funkcji na sam dół i przenieś definicję `print_lyrics()` po definicji `repeat_lyrics()`. Co się stanie, gdy uruchomisz taki program?

## 4.8. Przepływ sterowania

Aby zapewnić, że funkcja zostanie zdefiniowana przed jej pierwszym użyciem, musisz znać kolejność, w jakiej wykonywane są instrukcje – tę kolejność nazywa się *przepływem sterowania*.

Wykonanie poleceń rozpoczyna się zawsze od pierwszej instrukcji programu. Instrukcje są wykonywane pojedynczo, w kolejności od góry do dołu.

*Definicje* funkcji nie zmieniają przepływu sterowania programu, ale pamiętaj, że instrukcje wewnątrz funkcji nie są wykonywane, dopóki funkcja nie zostanie wywołana.

Wywołanie funkcji można traktować w przepływie sterowania jak objazd. Zamiast przejść do następnego polecenia, przepływ sterowania przeskakuje do ciała funkcji, wykonuje tam wszystkie polecenia, a następnie wraca do miejsca, w którym przerwał.

Brzmi to całkiem prosto, dopóki nie przypomnisz sobie, że jedna funkcja może wywołać inną. Będąc w środku jednej funkcji, program może być zmuszony do wykonania poleceń w innej funkcji. Co więcej, być może podczas wykonywania tej nowej funkcji program będzie musiał wykonać jeszcze inną funkcję!

Na szczęście Python jest dobry w śledzeniu, gdzie się znajduje, więc za każdym razem, gdy funkcja się kończy, program wznawia działanie tam, gdzie przerwał, w funkcji, która ją wywołała. Po dojściu do końca programu, jego działanie zostaje zakończone.

Jaki jest morał tej strasznej opowieści? Kiedy czytasz kod programu, nie zawsze chcesz go czytać od góry do dołu. Czasami większy sens ma podążanie za jego przepływem sterowania.

## 4.9. Parametry i argumenty

Niektóre z wbudowanych funkcji, które widzieliśmy, wymagają argumentów. Na przykład, gdy wywołujesz `math.sin()`, podajesz liczbę jako argument. Niektóre funkcje przyjmują więcej niż jeden argument: `math.pow()` przyjmuje dwa, tj. podstawę i wykładnik.

Wewnątrz funkcji argumenty są przypisane do zmiennych nazywanych *parametrami*. Oto przykład zdefiniowanej przez użytkownika funkcji, która przyjmuje argument:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

Funkcja ta przypisuje argument do parametru o nazwie `bruce`. Gdy funkcja zostanie wywołana, wypisuje wartość parametru (czymkolwiek on jest) dwukrotnie.

Funkcja ta działa z dowolną wartością, która może być wypisana.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
```

```
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

Te same zasady składania wyrażeń lub instrukcji, które odnoszą się do funkcji wbudowanych, dotyczą także funkcji zdefiniowanych przez użytkownika, więc możemy użyć dowolnego rodzaju wyrażenia jako argumentu w `print_twice()`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

Argument jest ewaluowany przed wywołaniem funkcji, więc w powyższych przykładach wyrażenia `'Spam '*4` i `math.cos(math.pi)` są ewaluowane tylko raz.

Możesz również użyć zmiennej jako argumentu:

```
>>> michael = 'Eryk Pół-Ćma.'
>>> print_twice(michael)
Eryk Pół-Ćma.
Eryk Pół-Ćma.
```

Nazwa zmiennej, którą przekazujemy jako argument (`michael`), nie ma nic wspólnego z nazwą parametru (`bruce`). Nie ma znaczenia, jak na tę wartość wołano w domu (w miejscu wywoływania); tutaj w `print_twice()` wszystkich nazywamy `bruce`.

## 4.10. Funkcje owocne i funkcje puste

Niektóre z używanych przez nas funkcji to np. funkcje matematyczne, które zwracają wyniki; z braku lepszej nazwy nazywam je *funkcjami owocnymi* (ang. *fruitful functions*). Inne funkcje, takie jak `print_twice()`, wykonują jakąś akcję, ale nie zwracają wartości. Nazywa się je *pustymi funkcjami*[3] (ang. *void functions*).

Kiedy wywołujesz owocną funkcję, prawie zawsze chcesz zrobić coś ze zwróconym wynikiem. Na przykład możesz przypisać go do zmiennej lub użyć go jako części wyrażenia:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

Kiedy wywołujesz funkcję w trybie interaktywnym, Python wyświetla wynik:

```
>>> math.sqrt(5)
2.23606797749979
```

Ale w skrypcie, jeśli wywołasz owocną funkcję i nie zapiszesz wyniku funkcji w zmiennej, to zwracana wartość zniknie jak we mgle!

```
math.sqrt(5)
```

Powyższy skrypt oblicza pierwiastek kwadratowy z 5, ale ponieważ nie zapisuje wyniku w zmiennej ani nie wyświetla zwróconego wyniku, nie jest on zbyt użyteczny.

[3] W niektórych językach programowania tego typu funkcje nazywa się procedurami.

Funkcje puste mogą wyświetlać coś na ekranie lub mieć jakiś inny efekt. W związku z tym, że nie zwracają żadnej wartości, jeśli spróbujesz przypisać wynik takiej funkcji do zmiennej, otrzymasz specjalną wartość o nazwie `None`.

```
>>> result = print_twice('Ping')
Ping
Ping
>>> print(result)
None
```

Wartość `None` nie jest taka sama jak napis `'None'`. Jest to specjalna wartość, która ma swój własny typ:

```
>>> print(type(None))
<class 'NoneType'>
```

Aby zwrócić wynik z funkcji, używamy w niej instrukcji `return`. Na przykład możemy napisać bardzo prostą funkcję o nazwie `addtwo()`, która dodaje dwie liczby i zwraca wynik dodawania.

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)

# Kod źródłowy: https://py4e.pl/code3/addtwo.py
```

Gdy powyższy skrypt zostanie wykonany, funkcja `print()` wypisze `8`, ponieważ funkcja `addtwo()` została wywołana z argumentami 3 i 5. Wewnątrz funkcji parametry `a` i `b` miały wartości odpowiednio 3 i 5. Funkcja obliczyła sumę tych dwóch liczb i umieściła ją w zmiennej lokalnej o nazwie `added`. Następnie użyła instrukcji `return`, by wysłać wyliczoną wartość z powrotem do kodu wywołującego funkcję, gdzie zwrócona wartość została przypisana do zmiennej `x` i wypisana.

## 4.11. Dlaczego potrzebujemy funkcji?

Na początku może nie być jasne, dlaczego warto podzielić program na funkcje. Jest kilka powodów:

- Stworzenie nowej funkcji daje Ci możliwość nazwania grupy poleceń, co sprawia, że Twój program jest łatwiejszy do odczytania, zrozumienia i debugowania.
- Funkcje mogą sprawić, że program będzie mniejszy poprzez wyeliminowanie powtarzającego się kodu. Później, jeśli dokonasz zmiany kodu, będziesz musiał zrobić to tylko w jednym miejscu.
- Dzielenie długiego programu na funkcje pozwala na debugowanie po kolei jego części, a następnie złożenie ich w działającą całość.
- Dobrze zaprojektowane funkcje są często przydatne w wielu programach. Kiedy już napiszesz i zdebugujesz jedną z nich, możesz jej ponownie użyć w innym programie.

W pozostałej części książki często będziemy używać definicji funkcji, tak by wyjaśnić pewne pojęcie. Częścią umiejętności tworzenia i korzystania z funkcji jest posiadanie funkcji, która właściwie uchwyci cel, taki jak "znajdź najmniejszą wartość na liście wartości". Później pokażemy Ci kod, który znajduje najmniejszą wartość na liście i przedstawimy go jako funkcję o nazwie `min()`, która przyjmuje listę wartości jako swój argument i zwraca najmniejszą wartość występującą na tej liście.

## 4.12. Debugowanie

Jeśli używasz edytora tekstu do pisania swoich skryptów, możesz mieć problemy ze spacjami i tabulacjami. Najlepszym sposobem na uniknięcie tych problemów jest użycie wyłącznie spacji (bez tabulacji). Większość (ale nie wszystkie) edytorów tekstowych, które znają specyfikę Pythona, robi to domyślnie.

Zazwyczaj ciężko odróżnić pojedynczą tabulację od grupy czterech spacji, co utrudnia proces debugowania, więc postaraj się znaleźć taki edytor, który zarządza wcięciami za Ciebie.

Nie zapomnij również zapisać swojego programu, zanim go uruchomisz. Niektóre środowiska programistyczne robią to automatycznie, a niektóre nie. W takim przypadku program, który oglądasz w edytorze tekstu, nie jest taki sam jak program, który uruchamiasz.

Jeśli będziesz ciągle uruchamiać ten sam nieprawidłowy program, to debugowanie może zająć Ci bardzo dużo czasu!

Upewnij się, że kod, na który patrzysz, jest tym samym kodem, który uruchamiasz. Jeśli nie jesteś pewien, umieść coś w rodzaju `print("hello")` na początku programu i uruchom go ponownie. Jeśli nie widzisz `hello`, to nie uruchamiasz właściwego programu![4]

## 4.13. Słowniczek

**algorytm** Ogólny proces rozwiązywania pewnej kategorii problemów.
**argument** Wartość przekazywana do funkcji w momencie jej wywołania. Wartość ta jest przypisana do odpowiadającego jej parametru w funkcji.
**biblioteka** Inaczej moduł.
**ciało funkcji** Sekwencja instrukcji wewnątrz definicji funkcji.
**definicja funkcji** Instrukcja, która tworzy nową funkcję, określająca jej nazwę, parametry i polecenia, które wykonuje.
**deterministyczny** Dotyczy programu, który robi to samo za każdym razem, gdy dostarczy mu się te same dane wejściowe.
**funkcja** Nazwany ciąg instrukcji wykonujący jakąś użyteczną operację. Funkcje mogą, ale nie muszą, przyjmować argumenty i mogą, ale nie muszą, zwracać wyniki.
**funkcja pusta** Funkcja, która nie posiada zwracanej wartości.
**funkcja owocna** Funkcja zwracająca wartość.
**import** Instrukcja, która wczytuje plik modułu i tworzy obiekt modułu.
**nagłówek funkcji** Pierwszy wiersz definicji funkcji.
**notacja kropkowa** Składnia do wywołania funkcji w innym module poprzez określenie nazwy modułu, po której następuje kropka i nazwa funkcji.
**obiekt funkcji** Wartość utworzona przez definicję funkcji. Nazwa funkcji jest zmienną, która odnosi się do obiektu funkcji.
**obiekt modułu** Wartość utworzona przez instrukcję `import`, która zapewnia dostęp do danych i kodu zdefiniowanego w module.
**parametr** Nazwa używana wewnątrz funkcji do odwoływania się do wartości przekazanej jako argument.
**przepływ sterowania** Kolejność, w której polecenia są wykonywane podczas uruchamiania programu.
**pseudolosowość** Odnosi się do ciągu liczb, które wydają się być losowe, ale są generowane przez deterministyczny program.
**składanie** Użycie wyrażenia jako części większego wyrażenia lub instrukcji jako części większej instrukcji.
**wartość zwracana** Wynik działania funkcji. Jeżeli wywołanie funkcji zostanie użyte jako wyrażenie, to wartość zwracana jest wartością tego wyrażenia.
**wywołanie funkcji** Polecenie, które uruchamia funkcję. Składa się z nazwy funkcji, po której następuje lista argumentów.

[4] Ta technika debugowania nosi nazwę *caveman debugging* (z ang. debugowanie metodą jaskiniowca) lub *print debugging* i nie jest zalecana podczas profesjonalnego programowania, jednak dobrze sprawdza się podczas pierwszych kroków w nauce programowania.

## 4.14. Ćwiczenia

**Ćwiczenie 4.** Jaki jest cel słowa kluczowego `def` w Pythonie?

a. Jest to slang, który oznacza: "następujący kod jest naprawdę fajny".
b. Oznacza początek funkcji.
c. Oznacza, że następujące wcięcie sekcji kodu ma być przechowywane na później.
d. Odpowiedzi b. i c. są poprawne.
e. Żadne z powyższych.

**Ćwiczenie 5.** Co wypisze następujący program Pythona?

```
def fred():
    print("Zap")

def jane():
    print("ABC")

jane()
fred()
jane()
```

(Dla uproszenia zapisu każda odpowiedź jest podana w jednej linii.)

a. Zap ABC jane fred jane
b. Zap ABC Zap
c. ABC Zap jane
d. ABC Zap ABC
e. Zap Zap Zap

**Ćwiczenie 6.** Przepisz ponownie swoje obliczenie wynagrodzenia z dodatkiem za nadgodziny i stwórz funkcję o nazwie `computepay()`, która przyjmuje dwa argumenty (parametry `hours` i `rate`).

```
Podaj liczbę godzin: 45
Podaj stawkę godzinową: 10
Wynagrodzenie: 475.0
```

**Ćwiczenie 7.** Napisz ponownie program z poprzedniego rozdziału do wyliczania ocen za pomocą funkcji o nazwie `computegrade()`, która przyjmuje jako argument wartość i zwraca ocenę jako napis.

```
Wartość   Ocena
>= 0.9     5,0
>= 0.8     4,5
>= 0.7     4,0
>= 0.6     3,5
>= 0.5     3,0
 < 0.5     2,0
```

```
Podaj wartość: 0.95
5,0
```

```
Podaj wartość: doskonała
Niepoprawna wartość
```

```
Podaj wartość: 10.0
Niepoprawna wartość
```

```
Podaj wartość: 0.75
4,0
```

```
Podaj wartość: 0.5
3,0
```

```
Podaj wartość: 0.46
2,0
```

Uruchom program kilkukrotnie, tak aby przetestować różne wartości.