Rafał Korzeniewski

Wstęp do języka Python. Część 8: funkcje (część 1)

Funkcje to potężny oręż w programistycznym arsenale. Dzięki nim kod programu może być bardziej zwięzły, czytelny i uniwersalny. Raz napisany kod może być potem łatwo wykorzystywany w innych miejscach programu. Dlatego warto je poznać i się z nimi zaprzyjaźnić.

10+

DOWIESZ SIĘ

- Jak definiować funkcje i ich używać.

POTRZEBNA WIEDZA

- Znajomość podstawowych struktur, typów danych, składni Pythona.

CZYM JEST FUNKCJA I JAK SIĘ JĄ DEFINIUJE?

Funkcja jest fragmentem kodu ujętym w specjalnej definicji. Fragment ten może być wielokrotnie używany w innych miejscach programu poprzez odwołanie do nazwy funkcji. Do nazw funkcji odnoszą się te same reguły, co do nazw zmiennych. Przyjęło się też, że nazwy funkcji w Pythonie są pisane małymi literami, a słowa rozdziela się znakiem podkreślenia.

Definicja funkcji zaczyna się od słówka kluczowego `def`, po którym umieszczamy nazwę funkcji. Po nazwie umieszczamy nawiasy okrągłe oraz dwukropek, po którym powinniśmy przejść do kolejnej linii i zrobić wcięcie. Od tej kolejnej linii zaczyna się ciało funkcji, które powinno być przesunięte w prawo zgodnie z regułami wcięć w Pythonie. Przykład takiej definicji znajdziemy w Listingu 1.

Listing 1. Definicja prostej funkcji

```
def hello():
    print("Hello World!")
```

Ciało funkcji jest w tym przypadku banalne, jest to po prostu wywołanie funkcji `print`. Zdefiniowaną funkcję możemy następnie wywołać w innych miejscach kodu. Za każdym takim wywołaniem kod zawarty w funkcji będzie wykonywany:

Listing 2. Wielokrotne wywołanie funkcji „hello"

```
hello()
hello()
hello()
```

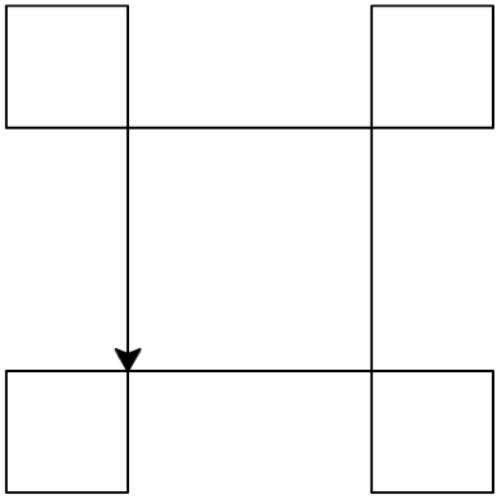
co da nam rezultat:

Listing 3. Wynik wielokrotnego wywołania funkcji „hello"

```
Hello World!
Hello World!
Hello World!
```

Taka funkcja może zawierać w zasadzie dowolny kod Pythona i przydatna jest szczególnie wtedy, gdy dany kod wykonywany jest w programie wielokrotnie. Zamiast wiele razy pisać taki sam kod w różnych miejscach programu, możemy zdefiniować go raz, a potem przywoływać, gdy jest potrzebny.

Powiedzmy, że chcielibyśmy narysować przy pomocy Pythona figurę taką jak poniżej:



Ilustracja 1. Figura narysowana przy pomocy „turtle"

Spójrzmy na poniższy przykład:

Listing 4. Rysowanie figury przy pomocy „turtle"

```
import turtle

def square(len=10):
   for i in range(4):
       turtle.forward(len)
       turtle.right(90)

square(50)
turtle.forward(150)
square(50)
turtle.left(90)
turtle.forward(100)
square(50)
turtle.left(90)
turtle.forward(100)
square(50)
turtle.left(90)
turtle.forward(100)

turtle.exitonclick()
```

W przykładzie tym korzystamy z modułu `turtle`, który umożliwia rysowanie różnego rodzaju figur. Działa to tak, jakbyśmy przyłożyli ołówek do kartki i realizowali proste instrukcje. Przesuń ołówek do przodu o tyle kroków, zmień kierunek rysowania o tyle stopni i tak dalej. Aby narysować kwadrat o boku o zadanej długości (`len`), musimy powtórzyć 4-krotnie następujące kroki – idź do przodu o długość `len`, obróć się o 90 stopni w prawo (może być też w lewo – ale konsekwentnie). Po czterech iteracjach wrócimy do punktu wyjścia.

W turtle służą do tego metody takie jak `forward`, `right`, `left`. Metody te to w zasadzie też funkcje. W tym przypadku zdefiniowane przez twórców modułu. W jaki sposób je wywołujemy, widać w Listingu 4. By ułatwić sobie życie, możemy kroki potrzebne do narysowania kwadratu ująć w funkcji (zobacz funkcję `square`), a następnie wywoływać tę funkcję w odpowiednich miejscach programu. Dzięki temu nasz kod jest znacznie krótszy i czytelniejszy.

PARAMETRY (ARGUMENTY) FUNKCJI

Definiując funkcję, możemy uwzględnić dodatkowe parametry, które będą potrzebne w ciele funkcji do wykonania jakichś zadań. Takie parametry w definicji umieszczamy wewnątrz nawiasów występujących po nazwie funkcji. Możemy nieznacznie zmodyfikować naszą funkcję `hello`, tak by wypisywała spersonalizowane przywitania:

Listing 5. Definicja funkcji z parametrem

```
def hello(name):
   print(f"Hello {name}!")

hello("World")
hello("Gabriela")
hello("R2D2")
```

Rezultat tych wywołań naszej funkcji znajdziemy w Listingu 5:

Listing 6. Rezultat wywołania funkcji „hello" z różnymi argumentami

```
Hello World!
Hello Gabriela!
Hello R2D2!
```

Jak widać, parametr zdefiniowany w funkcji przyjmuje w czasie wywołania podaną wartość. Tę wartość nazywamy też argumentem. W powyższym przykładzie `name` jest więc parametrem, a `World`, `Gabriela`, `R2D2` to argumenty. Parametry możemy traktować jako zmienne

i do tworzenia ich nazw stosujemy te same zasady jak w tworzeniu nazw zmiennych. Argumentem może być właściwie dowolny obiekt Pythona – w tym także inne funkcje. Do tego wrócimy w kolejnych artykułach. Czasem dla wygody słowa **parametr** i **argument** mogą też być stosowane zamiennie, warto jednak dostrzegać subtelną różnicę między nimi.

W definicji funkcji możemy stosować wiele parametrów. Rozdzielamy je wtedy przecinkami. Podobnie przecinkami oddzielamy od siebie argumenty w wywołaniu:

Listing 7. Więcej parametrów w definicji funkcji oraz jej wywołanie

```
def hello(name, surname):
    print(f"Hello {name} {surname}!")

hello("Gabriela", "Korzeniewska")
```

Argumenty podane w wywołaniu funkcji są wtedy przypisywane automatycznie do odpowiednich parametrów zgodnie z kolejnością ich występowania w definicji.

Wywołując funkcję, możemy też podawać argumenty, posługując się nazwami parametrów. W Listingu 8 przedstawiono kilka takich możliwości.

Listing 8. Wywołanie funkcji z wykorzystaniem nazw parametrów

```
hello("Gabriela", surname="Korzeniewska")
hello(name="Gabriela", surname="Korzeniewska")
hello(surname="Korzeniewska", name="Gabriela")
```

Jak widać, jeśli używamy nazw, to możemy nawet zamieniać kolejność. Co jednak istotne – jeśli użyjemy nazwy parametru, to dla parametrów występujących dalej też powinniśmy użyć nazwy. Inaczej dostaniemy błąd:

Listing 9. Przykład nieprawidłowego wywołania funkcji z użyciem nazwy parametru

```
hello(name="Gabriela", "Korzeniewska")
```

```
  File "main.py", line 29
    hello(name="Gabriela", "Korzeniewska")
                           ^
SyntaxError: positional argument follows
keyword argument
```

Co ważne – jeśli zdefiniowaliśmy funkcję tak jak w Listingu 7, to wywołując ją, musimy podać dokładnie tyle argumentów, ile jest w definicji. Jeśli podamy ich zbyt mało lub zbyt dużo, to dostaniemy błąd informujący nas o tym:

Listing 10. Przykłady błędnych wywołań funkcji z Listingu 7

```
hello("Gabi")
TypeError: hello() missing 1 required
positional argument: 'surname'

hello("Gabi", "X", "Y")

TypeError: hello() takes 2 positional
arguments but 3 were given
```

Jak widzimy, interpreter Pythona informuje nas w pierwszym przykładzie, że zabrakło argumentu `surname`, a w drugim, że podaliśmy 3 argumenty, ale definicja przewiduje tylko 2.

W błędach z Listingów 9 i 10 pojawia się sformułowanie *positional* oraz *keyword argument*. Po polsku nazwalibyśmy te argumenty pozycyjnymi oraz kluczowymi. Argumenty pozycyjne to takie, które znamy z poprzednich przykładów. Argumenty kluczowe to takie, które mają jakąś wartość domyślną. Wartość tę przypisuje się przy pomocy znaku =. Definiując funkcję, musimy pamiętać o tym, że argumenty pozycyjne powinny występować przed argumentami kluczowymi. Tak jak w poniższym przykładzie:

Listing 11. Użycie argumentów kluczowych i pozycyjnych, przykłady wywołań i ich rezultaty

```
def foo(pos1, pos2, kw1="x", kw2="y"):
    print(f"pos1: {pos1}, pos2: {pos2}"
          f", kw1: {kw1}, kw2: {kw2}")

foo(1, 2)
foo(1, 2, 3)
foo(1, 2, kw2=3)
foo(1, 2, 3, 4)
foo(1, 2, kw1=3, kw2=4)

# rezultaty wywołań

pos1: 1, pos2: 2, kw1: x, kw2: y
pos1: 1, pos2: 2, kw1: 3, kw2: y
```

```
pos1: 1, pos2: 2, kw1: x, kw2: 3
pos1: 1, pos2: 2, kw1: 3, kw2: 4
pos1: 1, pos2: 2, kw1: 3, kw2: 4
```

Jak widzimy – jeśli mamy argumenty kluczowe, to możemy je pominąć w wywołaniu – mają one bowiem zapewnione wartości domyślne. Jeśli przekażemy taki argument w wywołaniu, to nowy argument zastępuje wartość domyślną.

W definicji parametry pozycyjne powinny występować przed tymi z wartościami domyślnymi. Jeśli spróbujemy zmienić kolejność, dostaniemy błąd:

Listing 12. Argumenty kluczowe w definicji powinny występować po pozycyjnych

```
def foo(kw1=1, pos1):
   ...
```

```
File "main.py", line 14
    def foo(kw1=1, pos1):
            ^
SyntaxError: non-default argument
follows default argument
```

ZWRACANA WARTOŚĆ

Kolejnym istotnym faktem dotyczącym funkcji jest to, że zwracają one zawsze jakąś wartość. Nasze funkcje `hello` też zwracają wartość. Jeśli nie powiemy Pythonowi, co powinno zostać zwrócone, to taka funkcja zawsze zwróci wartość `None`.

O tym, co zwraca funkcja, możemy przekonać się, przypisując jej wywołanie do jakiejś zmiennej, albo bezpośrednio przekazując ją do funkcji `print`.

Listing 13. Sprawdzenie, co zwraca funkcja. Pierwszy napis pochodzi z wywołania funkcji „print" wewnątrz funkcji. „None" jest zwracane domyślnie przez funkcję „hello"

```
def hello():
   print("Hello World!")

result = hello()
print(result)

# rezultat:

Hello World!
None
```

Zamiast domyślnego działania możemy użyć słowa `return` wewnątrz funkcji i podać po nim, co ma być zwrócone. Tak jak w Listingu 14.

Listing 14. Funkcja „add" zwraca sumę argumentów

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

result = add(1, 2)
print(result)

# rezultat to:

3
```

Podobne zachowania obserwowaliśmy już w przykładach z poprzednich artykułów. Na przykład gdy zmienialiśmy napis na liczbę lub gdy sprawdzaliśmy długość napisu czy listy. Tam też funkcje zwracały jakieś konkretne wartości. Poznaliśmy też funkcje, które zwracały `None` – na przykład `print` jest taką funkcją. Co prawda drukuje coś na ekranie, ale zwraca też `None`. Przekonaj się o tym samodzielnie. To, co funkcja zwraca, może być w zasadzie dowolnym obiektem. Nie musi być to też jeden obiekt, możemy je oddzielać od siebie przecinkiem – tak jak w przykładzie z Listingu 15, wtedy funkcja zwraca krotkę, która te obiekty będzie zawierać:

Listing 15. Jeśli po słowie „return" umieścimy więcej obiektów, to funkcja zwróci krotkę, która będzie zawierać te obiekty

```
def foo():
   return 1, 2

baz = foo()

print(baz)

baz, bar = foo()

print(baz)
print(bar)

# wynik działania:

(1, 2)
1
2
```

CIEKAWOSTKA

Nazwy takie jak „foo" i „bar" stosowane są jako tak zwane zmienne metasynktatyczne. Stosowane są one najczęściej w przykładach programistycznych. Dzięki ich użyciu nie musimy zastanawiać się nad wymyślaniem nazw do takich przykładów, a jedynie skupiamy się na przedstawieniu danego problemu. Pochodzenie (etymologia) tych słów wyjaśnione jest w dokumencie: https://www.ietf.org/rfc/rfc3092.txt.

Jak widzimy, pierwszy `print` wydrukował nam krotkę. Możemy wartości z niej przypisać do różnych zmiennych w jednej linijce.

W artykule celowo używaliśmy bardzo prostych przykładów. Funkcje dają jednak potężne możliwości i zachęcam do samodzielnego eksperymentowania z nimi.

W następnym numerze PJR będziemy kontynuować poznawanie funkcji. Dowiemy się o przestrzeniach nazw, nauczymy się tworzyć funkcje, które przyjmują dowolną liczbę argumentów – zarówno pozycyjnych, jak i kluczowych. Dowiemy się też, że możemy w Pythonie definiować takie funkcje, które wcale nie mają nazwy.

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

- W zależności od definicji funkcja może przyjąć zero lub wiele argumentów.
- Niektóre argumenty mogą mieć wartości domyślne — są to tak zwane argumenty kluczowe.
- Argumenty kluczowe w definicji zawsze występują po pozycyjnych.
- Funkcja zawsze zwraca jakąś wartość. Jeśli nie wskażemy słowem kluczowym „return", co ma być zwracane, funkcja zwróci wartość „None".

ĆWICZ W DOMU

- Spróbuj samodzielnie napisać funkcję, która przyjmie jako argument jakiś napis i wypisze go kolejno zadaną ilość razy (domyślnie 3). Funkcja taka powinna przyjąć dwa argumenty, z czego drugi (ilość powtórzeń) ma wartość domyślną 3.
- Spróbuj napisać funkcję, która sprawdzi, czy dana liczba jest liczbą pierwszą. Funkcja taka powinna przyjąć jako argument liczbę i zwrócić „True" lub „False" w zależności od tego, czy dana liczba jest, czy nie jest liczbą pierwszą (liczby pierwsze to takie, które są podzielne tylko przez 1 i samą siebie: a więc na przykład 2, 3, 5, 7, 11 są liczbami pierwszymi, ale 6 już nie, ponieważ dzieli się ona także przez 2 i 3.
- Spróbuj samodzielnie narysować trójkąt, prostokąt i inne figury przy pomocy modułu „Turtle".