Zaawansowane programowanie w Pythonie

Wykład 3

dr Agnieszka Zbrzezny

30 października 2024

Funkcje anonimowe

Wyrażenia lambda

- Obok instrukcji **`def`** Python udostępnia również formę wyrażenia generującego obiekty funkcji. Ze względu na podobieństwo do narzędzia z języka LISP nosi ono nazwę **`lambda`**.
- Tak jak **`def`**, wyrażenie to tworzy funkcję, którą można wywołać później, jednak zwraca tę funkcję, zamiast przypisywać ją do nazwy.
- Z tego powodu wyrażenia lambda nazywane są czasami funkcjami anonimowymi (nienazwanymi).
- W praktyce często wykorzystywane są jako sposób skrótowego zapisania definicji funkcji lub opóźnienia wykonania fragmentu kodu.

Funkcje anonimowe

Wyrażenia lambda

- Na ogólną formę wyrażenia lambda składa się słowo kluczowe **`lambda`**, po którym następuje dowolna liczba argumentów, a po nich, po dwukropku, wyrażenie:

```
lambda arg1, ... argN : wyrażenie
```

- Obiekty funkcji zwracane przez wykonanie wyrażeń lambda działają dokładnie tak samo jak te utworzone i przypisane przez instrukcję **`def`**.
- Istnieje jednak kilka różnic sprawiających, że wyrażenie lambda staje się użyteczne w pewnych wyspecjalizowanych rolach:
  - Wyrażenie lambda nie jest instrukcją, lecz jest wyrażeniem. Z tego powodu może się pojawiać w miejscach, w których użycie **`def`** w składni Pythona nie jest dozwolone – wewnątrz literału listy czy w wywołaniu funkcji.
  - Ciałem lambda jest pojedyncze wyrażenie, a nie blok instrukcji.

Narzędzia programowania funkcyjnego

Przykład

```
>>> def cube(y):
...     return y * y * y
...
>>> # Przypisanie do zmiennej wyrażenia lambda
>>> # jest niepolecane wg PEP 8 -
>>> # przewodniku stylu kodowania w Pythonie
>>> lambda_cube = lambda y: y * y * y
>>>
>>> # Użycie funkcji zdefinowanej przez def
>>> print(cube(5))
125
>>>
>>> # Użycie funkcji zdefinowanej przez lambda
>>> print(lambda_cube(5))
125
```

Przykład

Użycie funkcji zwracającej klucz do porównywania

```
def year(item):
    return item[1]

d = {"Eve": 1999, "Ann": 2001, "Cay": 2000, "Bob": 2003}
print(max(d.items()))
print(max(d.items(), key=year))
print(sorted(d.items()))
print(sorted(d.items(), key=year))
```

Wynik działania powyższego programu:

```
('Eve', 1999)
('Bob', 2003)
[('Ann', 2001), ('Bob', 2003), ('Cay', 2000), ('Eve', 1999)]
[('Eve', 1999), ('Cay', 2000), ('Ann', 2001), ('Bob', 2003)]
```

Narzędzia programowania funkcyjnego

Przykład

Użycie wyrażenia lambda zwracającego klucz do porównywania

```
d = {"Eve": 1999, "Ann": 2001, "Cay": 2000, "Bob": 2003}
print(max(d.items()))
print(max(d.items(), key=lambda t: t[1]))
print(sorted(d.items()))
print(sorted(d.items(), key=lambda t: t[1]))
```

Wynik działania powyższego programu:

```
('Eve', 1999)
('Bob', 2003)
[('Ann', 2001), ('Bob', 2003), ('Cay', 2000), ('Eve', 1999)]
[('Eve', 1999), ('Cay', 2000), ('Ann', 2001), ('Bob', 2003)]
```

Funkcje anonimowe

Wyrażenia lambda

- Innym często wykorzystywanym zastosowaniem wyrażeń lambda jest definiowanie funkcji zwrotnych dla API graficznego interfejsu użytkownika tkinter Pythona.
- Przykładowo, poniższy kod tworzy przycisk wyświetlający po jego naciśnięciu komunikat w konsoli.

```
# button_lambda_demo.py

from tkinter import Tk, Frame, Button

root = Tk()
root.geometry("320x200")
frame = Frame(root).pack()
Button(frame, text = "Naciśnij mnie",
    command=(lambda: print("Jestem przyciskiem"))
).pack()
root.mainloop()
```

Funkcje anonimowe

Funkcje wyższego rzędu

- Funkcja wyższego rzędu jest to zwykła funkcja, z tą różnicą, że przyjmuje jako argument inną funkcję, lub zwraca funkcję.
- Przykładowo, funkcja wyższego rzędu, może wyglądać tak:

```
>>> def funkcja(func, number):
>>>     return func(number)
```

- Pierwszym argumentem jest funkcja `func`, natomiast drugim liczba `number`.
- Można wywołać naszą funkcję wyższego rzędu w następujący sposób:

```
>>> import math
>>> funkcja(math.sqrt, 2)
1.4142135623730951
>>>
```

Funkcje anonimowe

Funkcje wyższego rzędu

- Można zdefiniować zwykłą funkcję tylko i wyłącznie na potrzeby wywołania funkcji wyższego rzędu:

```
>>> def funkcja(func, number):
...     return func(number)
...
>>> def cube(x):
...     return x * x * x
...
>>> funkcja(cube, 3)
27
```

- Jest to sposób prawidłowy, jednak często taki zapis można skrócić poprzez zastosowanie funkcji lambda:

```
>>> funkcja(lambda x: x * x * x, 3)
27
```

Funkcje anonimowe

Funkcje wyższego rzędu

- Wyrażenia lambda można użyć, aby określić `klucz` sortowania:

```
>>> d = {'b': 42, 'c': 1, 'a': 2}
>>> sorted(d.items())
[('a', 2), ('b', 42), ('c', 1)]
>>> sorted(d.items(), key=lambda t: t[0])
[('a', 2), ('b', 42), ('c', 1)]
>>> sorted(d.items(), key=lambda t: t[0], reverse=True)
[('c', 1), ('b', 42), ('a', 2)]
>>> sorted(d.items(), key=lambda t: t[1])
[('c', 1), ('a', 2), ('b', 42)]
>>> sorted(d.items(), key=lambda t: t[1], reverse=True)
[('b', 42), ('a', 2), ('c', 1)]
ob = sorted(d.items(), key=lambda t: t[1], reverse=True)
>>> type(ob)
<class 'list'>
```

Iteratory

Uwagi ogólne

- Iteratorem nazywamy obiekt pozwalający na sekwencyjny dostęp do wszystkich elementów lub części zawartych w innym obiekcie, zwykle kolekcji lub łańcuchu znaków.
- Iterator można rozumieć jako rodzaj wskaźnika udostępniającego dwie podstawowe operacje: odwołanie się do konkretnego elementu w kolekcji (dostęp do elementu) oraz modyfikację samego iteratora tak, by wskazywał na kolejny element (sekwencyjne przeglądanie elementów).
- Musi także istnieć sposób utworzenia iteratora tak, by wskazywał na pierwszy element, oraz sposób określenia, kiedy iterator wyczerpał wszystkie elementy w kolekcji.

Iteratory

Uwagi ogólne

- W zależności od języka i zamierzonego zastosowania iteratory mogą dostarczać dodatkowych operacji lub posiadać różne dodatkowe zachowania.
- Podstawowym celem iteratora jest pozwolić użytkownikowi przetworzyć każdy element w kolekcji bez konieczności zagłębiania się w jej wewnętrzną strukturę.
- Pozwala to kolekcji przechowywać elementy w dowolny sposób, podczas gdy użytkownik może traktować ją jak zwykłą sekwencję lub listę.
- Klasa iteratora jest zwykle projektowana wraz z klasą odpowiadającej mu kolekcji i jest z nią ściśle powiązana.
- Zwykle to kolekcja dostarcza metod tworzących iteratory.

Iteratory w Pythonie

- Iteratory są jednym z podstawowych elementów Pythona i często są w ogóle niezauważalne, gdyż są niejawnie wykorzystywane w pętlach `for`.
- Wszystkie standardowe typy sekwencyjne w Pythonie, jak również wiele klas w bibliotece standardowej, udostępniają iterację.
- Iteratory można również definiować w sposób jawny.
- Do pobrania iteratora z kolekcji typu sekwencyjnego wykorzystuje się funkcję wbudowaną `iter()`.
- Wbudowana funkcja `next()` zwraca przy każdym wywołaniu kolejny element kolekcji oraz modyfikuje iterator tak, aby wskazywał on na następny element kolekcji.
- Gdy nie ma więcej elementów, funkcja ta wyrzuca wyjątek `StopIteration`.

Iteratory

Przykład

```
it = iter(sequence)
try:
    while True:
        val = next(it); print(val)
except StopIteration:
    pass
```

Iteratory w Pythonie

- Dowolna zdefiniowana przez użytkownika klasa może udostępniać standardową iterację (niejawną lub jawną), jeśli posiada metodę `__iter__()` zwracającą iterator.
- Zwrócony iterator musi posiadać również metodę `__iter__()` oraz metodę `__next__()`.

Iteratory

Obiekty iterowalne (ang. iterable)

- Obiekty iterowalne reprezentują ciągi obiektów. Przez obiekty iterowalne można iterować: przykładami są pętle for, konstrukcje list lub słowników składanych.
- Wiele metod lub funkcji z bibliotek standardowych (i nie tylko) jest napisanych tak, aby ich argumentami były dowolne obiekty iterowalne (np. `sorted(iterable)`).
- W Pythonie, aby `obj` był rozpoznany jako obiekt iterowalny, on i jego iteratory muszą realizować protokół iteratorów:
  - Obiekt iterowalny `obj` musi implementować metodę specjalną `__iter__()`, zwracającą iterator.
  - Iterator musi implementować:
    - metodę `__next__()`, która albo zwraca obiekt do skonsumowania, albo rzuca wyjątek `StopIteration`, reprezentujący koniec iteracji;
    - metodę `__iter__()`, zwracającą iterator (może nim być – i zazwyczaj jest – on sam).

Iteratory

Obiekty iterowalne (ang. iterable)

- Począwszy od Pythona 3.4, najdokładniejszym sposobem sprawdzenia, czy obiekt `ob` jest iterowalny, jest wywołanie funkcji `iter(ob)` i obsłużenie wyjątku `TypeError`, jeżeli tak nie jest.
- Przykłady obiektów iterowalnych:
  - łańcuchy znaków (obiekty klasy `str`)
  - sekwencje bajtów (obiekty klas `bytes` oraz `bytearray`)
  - listy (obiekty klasy `list`)
  - zbiory (obiekty klasy `set`)
  - krotki (obiekty klasy `tuple`)
  - słowniki (obiekty klasy `dict`)
  - pliki (obiekty zwracane przez funkcję `open`)
  - zakresy (obiekty klasy `range`)
- Zauważmy, że obiekty we wszystkich powyższych przykładach są obiektami iterowalnymi, ale nie są iteratorami.

Iteratory

Obiekty iterowalne (ang. iterable)

- Zbadajmy typy iteratorów wybranych obiektów iterowalnych:

```
for typ in (str, tuple, list, set, dict):
    obj = typ() # nowy obiekt danego typu
    it = iter(obj) # nowy iterator tego obiektu
    print(type(it))
print(type(iter(range(0)))) # range() niepoprawne
```

- Otrzymujemy następujace wyniki:

```
<class 'str_iterator'>
<class 'tuple_iterator'>
<class 'list_iterator'>
<class 'set_iterator'>
<class 'dict_keyiterator'>
<class 'range_iterator'>
```

Iteratory

Obiekty iterowalne (ang. iterable)

- Zwróćmy uwage na iterator słownika: `dict_keyiterator`, z nazwą sugerującą, że iterator służy do iterowania po kluczach słownika.
- Rzeczywiście, pętla iterująca po słowniku iteruje po jego kluczach:

```
>>> d = {'a': 1, 'b': 2, 'c': 42}
>>> for k in d:
...     print(k)
...
a
b
c
```

Iteratory

Obiekty iterowalne (ang. iterable)

- Przypomnijmy jednak, że słowniki mają które zwracają iterowalne obiekty reprezentujące wartości oraz pary klucz-wartość:

```
>>> values = d.values()
>>> print(type(values), type(iter(values)))
<class 'dict_values'> <class 'dict_valueiterator'>
>>> items = d.items()
>>> print(type(items), type(iter(items)))
<class 'dict_items'> <class 'dict_itemiterator'>
>>> # Poniżej iterujemy po d.items(), zwrócone
>>> # obiekty są konsumowane przez konstruktor listy
>>> print(list(d.items()))
[('a', 1), ('b', 2), ('c', 42)]
>>>
```

Implementowanie klasy iteratora

- Obiekty będące iteratorami w Pythonie są zgodne z protokołem iteracyjnym, co zasadniczo oznacza, że zapewniają dwie metody: `__iter__()` oraz `__next__()`.
- Iterator to obiekt reprezentujący strumień danych; ten obiekt zwraca ze strumienia danych jeden element na raz.
- Iterator Pythona musi obsługiwać metodę o nazwie `__next__()`, która nie przyjmuje argumentów i zawsze zwraca następny element strumienia.
- Jeżeli nie ma więcej elementów w strumieniu, metoda `__next__()` musi wyrzucić wyjątek `StopIteration`.
- Iteratory nie muszą być skończone; rozsądnie jest napisać iterator, który generuje nieskończony strumień danych.

Iteratory

Przykład klasy iteratora

```
class PowersOfTwo:
    def __init__(self):
        self.num = 1

    def __iter__(self):
        return self

    def __next__(self):
        num = self.num
        self.num *= 2
        return num
```

Wykorzystanie klasy iteratora

```
from sys import argv
from powersoftwo import PowersOfTwo

def main():
    try:
        limit = int(argv[1])
    except:
        limit = 10000
    it = iter(PowersOfTwo())
    a = next(it)
    while a < limit:
        print(a)
        a = next(it)

if __name__ == "__main__":
    main()
```

Narzędzia programowania funkcyjnego

Funkcja wbudowana `map`

- Jednym z najczęściej wykonywanych działań na listach i innych obiektach iterowalnych jest zastosowanie jakiejś operacji do każdego ich elementu i zebranie wyników.
- Ponieważ jest to tak często wykonywana operacja, Python udostępnia odpowiednią funkcję wbudowaną, która jest w stanie zrobić to za nas.
- Funkcja `map` służy do zastosowania przekazanej funkcji na każdym elemencie obiektu iterowalnego i zwraca obiekt iteratora zawierający wszystkie wyniki jej wywołania:
  `map(function, *iterables) -> map object`
- Obiekt ten może być przekształcony na listę przy pomocy funkcji wbudowanej `list`.

Narzędzia programowania funkcyjnego

Przykład (Funkcja wbudowana map)

```
>>> list(map(lambda x: x * x, range(6)))
[0, 1, 4, 9, 16, 25]
>>>
>>> # liczba obiektów iterowalnych musi być
>>> # równa liczbie argumentów funkcji
>>>
>>> potegi = map(pow, range(6), range(6))
>>> list(potegi)
[1, 1, 4, 27, 256, 3125]
>>>
>>> # działanie funkcji map kończy się po wyczerpaniu
>>> # najkrótszego z obiektów iterowalnych
>>> potegi = map(pow, range(6), range(9))
>>> list(potegi)
[1, 1, 4, 27, 256, 3125]
```

Narzędzia programowania funkcyjnego

Funkcja wbudowana `filter`

- Funkcja `filter` odfiltrowuje elementy obiektu iterowalnego w oparciu o funkcję testującą:
  `filter(function, iterable) -> filter object`
- Elementy obiektu iterowalnego, dla których funkcja testująca zwraca `True`, dodawane są do listy wyników.
- Funkcja `filter` zwraca obiekt iteratora zawierający odfiltrowane elementy.
- Obiekt ten może być przekształcony na listę przy pomoc funkcji wbudowanej `list`.

Narzędzia programowania funkcyjnego

Przykład (Funkcja wbudowana `filter`)

```
>>> liczby = [x for x in range(10)]
>>>
>>> def odd(x): return x % 2 == 1
...
>>> nieparzyste = filter(odd, liczby)
>>> print(list(nieparzyste))
[1, 3, 5, 7, 9]
>>>
>>> nieparzyste = filter(odd, range(10))
>>> print(list(nieparzyste))
[1, 3, 5, 7, 9]
>>>
>>> nieparzyste = filter(lambda x: x % 2 == 1, range(10))
>>> print(list(nieparzyste))
[1, 3, 5, 7, 9]
>>> print(list(nieparzyste))
[]
```

Narzędzia programowania funkcyjnego

Funkcja `reduce` z modułu `functools`

- Wywołanie funkcji `reduce(function, iterable)`, gdzie `function` jest funkcją dwuargumentową a `iterable` jest obiektem iterowalnym, zwraca pojedyńczą wartość obliczaną następująco:
  - funkcja `function` pobiera dwa pierwsze elementy obiektu `iterable` i oblicza wynik;
  - funkcja `function` pobiera poprzedni wynik oraz trzeci element z obiektu `iterable` i oblicza wynik;
  - .....
    .....
  - funkcja `function` pobiera poprzedni wynik oraz ostatni element z obiektu `iterable` i oblicza wynik;

Narzędzia programowania funkcyjnego

Funkcja `reduce` z modułu `functools`

- Funkcja `reduce` jest z grubsza równoważna następującej funkcji:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

Narzędzia programowania funkcyjnego

Przykład (Funkcja `reduce` z modułu `functools`)

```
# reduce_demo.py

from functools import reduce

print(reduce(lambda x, y: x + y, [1, 2, 3, 4]))
print(reduce(lambda x, y: x * y, [1, 2, 3, 4]))
print(reduce(lambda x, y: x * y, {1, 2, 3, 4}))
print(reduce(lambda x, y: x * y, range(1, 5)))

s = "Kiler skazany na dobre zmiany"
print(reduce(lambda x, y: y + x, s))
```