Wstęp do programowania

Programowanie funkcyjne

Paweł Daniluk

Wydział Fizyki

Jesień 2013

Paradygmaty programowania

Programowanie imperatywne

Program składa się z ciągu instrukcji, które zmieniają stan programu.

Stan wykonania programu

- wartości zmiennych globalnych
- wartości zmiennych lokalnych (wewnętrznych dla funkcji)
- stos wykonania
- aktualnie wykonywana instrukcja

Paradygmaty programowania

Programowanie funkcyjne

Program jest definicją funkcji (w sensie matematycznym) ze zbioru danych wejściowych w zbiór wyników.

Funkcja czysta (ang. *pure function*)

Funkcja (w rozumieniu języka programowania), która nie ma żadnych efektów ubocznych w szczególności:

- nie zmienia wartości zmiennych
- nie wykonuje operacji wejścia/wyjścia

Prosty przykład

Zadanie

Obliczyć wartość częściowo niepoprawnego wyrażenia postaci:

$$28+32+++32++39$$

Rozwiązanie imperatywne

```
expr, res = "28+32+++32++39", 0
for t in expr.split("+"):
    if t != "":
        res += int(t)

print res
```

Wykonanie

```
"28+32+++32++39", 0
"28", 0
"32", 28
"", 60
"", 60
"32", 60
"", 92
```

Prosty przykład

Zadanie

Obliczyć wartość częściowo niepoprawnego wyrażenia postaci:

$$28+32+++32++39$$

Rozwiązanie funkcyjne

```
from operator import add
expr = "28+32+++32++39"
print reduce(add, map(int, filter(bool, expr.split("+"))))
```

Wykonanie

```
"28+32+++32++39"
["28","32","","","32","","39"]
["28","32","32","39"]
[28,32,32,39]
131
```

Funkcje w Pythonie

Funkcje nazwane

```
def f(x):
    return x+1
```

Funkcje anonimowe

```
f=lambda x: x+1
```

λ-wyrażenia

Składnia

```
lambda_expr ::= "lambda"[parameter_list]: expression
```

Napis:

```
name=lambda arguments: expression
```

jest równoważny napisowi:

```
def name(arguments):
    return expression
```

Argumenty

Definicja argumentów λ-wyrażeń jest taka jak funkcji. Dozwolone są argumenty pozycyjne i nazwane, wartości domyślne, etc.

Funkcje to wartości

First-class functions

Python traktuje funkcje jako wartości. Można je przypisywać na zmienne, przekazywać jako argumenty i zwracać z funkcji.

Moduł operator

Moduł operator zawiera funkcje odpowiadające operatorom języka Python np.:

- `operators.add(a,b)` $\longleftarrow$ `a+b`
- `operators.mul(a,b)` $\longleftarrow$ `a*b`
- `operators.lt(a,b)` $\longleftarrow$ `a<b`
- `operators.getitem(a,b)` $\longleftarrow$ `a[b]`

Pętle

W "czystym" programowaniu funkcyjnym nie stosuje się pętli.

Przykład

```
name = None
while name is None:
    name = raw_input()
    if len(name) < 2:
        name = None
```

```
def get_name():
    name = raw_input()
    return name if len(name) >= 2 else get_name()
```

`raw_input()` nie jest czystą funkcją.

Pętle c.d

Przykład

```
sum=0
for i in list:
    sum+=i
```

```
def sum(list, acc=0):
    return acc if len(list)==0 else sum(list[1:], acc+list[0])
```

Pętle c.d

Przykład

```
sum=0
for i in list:
    sum+=i
```

```
def sum(list, acc=0):
    return acc if len(list)==0 else sum(list[1:], acc+list[0])
```

"Prawdziwe" języki funkcyjne mają składnię pozwalającą dopasowywać definicję funkcji do parametrów.

SML

```
fun sum (nil) = 0 | sum (head::tail) = head+sum(tail);
```

Funkcja map

$$map : (A \to B) \times A^* \to B^*$$

`map(function, iterable, ...)` aplikuje funkcję do wszystkich elementów listy.

Przykłady

```
>>> map(lambda x:x**2, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> map(operator.add, range(10), range(9,-1,-1))
[9, 9, 9, 9, 9, 9, 9, 9, 9, 9]
```

Wszystkie listy przekazywane do `map` muszą być jednakowej długości.

Funkcja map

$$map : (A \to B) \times A^* \to B^*$$

`map(function, iterable, ...)` aplikuje funkcję do wszystkich elementów listy.

Przykłady

```
>>> map(lambda x:x**2, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> map(operator.add, range(10), range(9,-1,-1))
[9, 9, 9, 9, 9, 9, 9, 9, 9, 9]
```

Alternatywnie

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [x[0]+x[1] for x in zip(range(10), range(9,-1,-1))]
[9, 9, 9, 9, 9, 9, 9, 9, 9, 9]
>>> zip(range(10), range(9,-1,-1))
[(0, 9), (1, 8), (2, 7), (3, 6), (4, 5), (5, 4), (6, 3), (7, 2), (8, 1), (9
```

Funkcja `filter`

$$filter : (A \to \{True, False\}) \times A^* \to A^*$$

`filter(function, iterable)` zwraca elementy listy, dla których funkcja daje wartość `True`.

Przykłady

```
>>> filter(lambda x:x%2==0, range(10))
[0, 2, 4, 6, 8]
```

Alternatywnie

```
>>> [x for x in range(10) if x%2==0]
[0, 2, 4, 6, 8]
```

Funkcja `reduce`

$$reduce : (B \times A \to B) \times A^* \times B \to B$$

`reduce(function, iterable[, initializer])` używa funkcji do kumulatywnego obliczenia pojedynczej wartości na podstawie listy.

$$reduce(f, [l_1, l_2, \ldots, l_n], a) = f\left(f\left(f\left(f\left(a, l_1\right), l_2\right), \ldots\right), l_n\right)$$

```
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) == ((((1+2)+3)+4)+5)
```

Przydatne gadżety

all, any

$$all, any : \{True, False\}^* \to \{True, False\}$$

$$all([l_0, \ldots, l_{n-1}] = \forall_{0 \le i < n} l_i$$
$$any([l_0, \ldots, l_{n-1}] = \exists_{0 \le i < n} l_i$$

enumerate

$$enumerate : A^* \to (\mathbb{N} \times A)^*$$

$$enumerate([l_0, l_1, \ldots, l_{n-1}]) = [(0, l_0), (1, l_1), \ldots, (n-1, l_{n-1})]$$

zip

$$zip : A^* \times B^* \to (A \times B)^*$$

$$zip([a_0, a_1, \ldots, a_{n-1}], [b_0, b_1, \ldots, b_{n-1}]) = [(a_0, b_0), (a_1, b_1), \ldots, (a_{n-1}, b_{n-1})]$$

Funkcje wyższego rzędu

Funkcje, które przyjmują funkcje jako argumenty (np. `map`), albo zwracają funkcje.

Przykład

$$pow : \mathbb{N} \to (\mathbb{R} \to \mathbb{R})$$

$$pow(n) = \lambda x.x^n$$

W Pythonie

```
def pow(n):
    return lambda x: x**n
```

albo

```
def pow(n):
    def f(x):
        return x**n

    return f
```

Częściowa aplikacja

$$partial : ((A \times B) \to C) \times A \to (B \to C)$$

$$partial(f, a) = \lambda b.f(a, b)$$

Przykład

```
def log(level, message):
    print "[{level}]: {msg}".format(level=level, msg=message)

from functools import partial
debug = partial(log, "debug")

debug("Start doing something")
debug("Continue with something else")
debug("Finished. Profit?")
```

Zamiast

```
def debug(message):
    log("debug", message)
```

Currying – rozwijanie funkcji

$$curry : (A \times B \times C \to D) \to (A \to (B \to (C \to D)))$$

$$curry(f) = \lambda a.\lambda b.\lambda c.f(a, b, c)$$

Przykład

```
def curry(f):
return lambda a: lambda b: f(a,b)

curriedadd=curry(operator.add)

addone=curriedadd(1)
addtwo=curriedadd(2)
```

Zadanie 1 – `my_max`, `my_min`

Zaimplementuj funkcje `my_max` i `my_min` znajdujące odpowiednio największy i najmniejszy element w liście.

Zadanie 2 – `my_zip`

Wykonaj własną implementację funkcji `zip`. Czy da się do zrobić bez użycia pętli?

Zadanie 3 – złożenia funkcji

Zdefiniuj funkcję realizującą operator złożenia dwóch funkcji:

$$(g \circ f)(x) = g(f(x))$$

Zdefiniuj funkcję obliczającą złożenie dowolnej liczby funkcji.

Zadanie 4 – sumy

Napisz możliwie najprostszy zestaw funkcji obliczających sumy:

- kolejnych k liczb naturalnych
- kwadratów kolejnych k liczb naturalnych
- logarytmów kolejnych k liczb naturalnych
- czegokolwiek kolejnych k liczb naturalnych

Zadanie 5* – Lata przestępne

Napisz funkcję obliczającą zadaną liczbę lat przestępnych począwszy od podanego roku.

Wskazówka

Pomocne mogą być funkcje z modułu `itertools`.

Strona wykładu

http://bioexploratorium.pl/wiki/Wstęp_do_programowania_
_2013z