Wstęp do programowania

Funkcje

Paweł Daniluk

Wydział Fizyki

Jesień 2013

Funkcje

Funkcje w matematyce

$$f : D \longrightarrow W$$

D – dziedzina
W – zbiór wartości

Funkcja może być wieloargumentowa

$$f : D_1 \times D_2 \times \cdots \times D_n \longrightarrow W$$

Funkcje w Pythonie

```
def f(arg1, arg2, ..., argN):
    compute
    compute
    ...
    compute

    return result
```

Podawanie argumentów

Domyślne wartości argumentów

```
def work(name="Jack"):
    print "All␣work␣and␣no␣play␣makes", name, "a␣dull␣boy."
```

Jeżeli w wywołaniu funkcji nie zostanie podany argument zostanie użyta domyślna wartość.

```
>>> work("Kevin")
All work and no play makes Kevin a dull boy.
>>> work()
All work and no play makes Jack a dull boy.
>>>
```

Podawanie argumentów c.d.

Argumenty nazwane

```
def parrot(voltage, state='a stiff', action='voom',
        type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

```
parrot(1000)                          # 1 positional argument
parrot(voltage=1000)                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM') # 2 keyword args
parrot(action='VOOOOOM', voltage=1000000) # 2 keyword args
parrot('a million', 'bereft of life', 'jump')
# 3 positional arguments
parrot('a thousand', state='pushing up the daisies')
# 1 positional, 1 keyword
```

Argumenty pozycyjne nie mogą następować po nazwanych.

Podawanie argumentów c.d.

Dowolna liczba argumentów pozycyjnych

```
def fun(*args):
    print args
```

args jest krotką zawierającą wszystkie argumenty pozycyjne.

```
>>> fun()
()
>>> fun('aa')
('aa',)
>>> fun(1,'aa',2,'bb',3,'cc')
(1, 'aa', 2, 'bb', 3, 'cc')
>>> fun(1,par=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fun() got an unexpected keyword argument 'par'
```

Podawanie argumentów c.d.

Dowolna liczba argumentów nazwanych

```
def fun(**kwargs):
    print kwargs
```

kwargs jest słownikiem zawierającym wszystkie argumenty nazwane.

```
>>> fun(a=1,b=2,c=3)
'a': 1, 'c': 3, 'b': 2
>>> fun(1,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fun() takes exactly 0 arguments (2 given)
```

Wszystkie metody specyfikowania argumentów można łączyć

```
def fun(par1, a=0, *args, **kwargs):
    ....
```

Podawanie argumentów c.d.

```
>>> def fun(a,b):
...     print a,b
...
>>> l=range(2)
>>> fun(*l)
0 1
>>> fun(*(range(3)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fun() takes exactly 2 arguments (3 given)
>>> d='b':3, 'a':2
>>> fun(**d)
2 3
>>> fun(*d)
a b
>>>
```

Argumenty – przez wartość, czy przez referencję?

Przekazywanie przez wartość

Zmiana wartości argumentu wewnątrz funkcji nie propaguje się na zewnątrz.

Przekazywanie przez referencję

Zmiana wartości argumentu wewnątrz funkcji powoduje zmianę wartości zmiennej podanej jako argument.

Argumenty – przez wartość, czy przez referencję? c.d.

Przykład (język E)

```
def modify(var p, &q) {
    p := 27 # passed by value - only the local parameter is modified
    q := 27 # passed by reference - variable used in call is modified
}

? var a := 1
# value: 1
? var b := 2
# value: 2
? modify(a,&b)
? a
# value: 1
? b
# value: 27
```

Argumenty – przez wartość, czy przez referencję?

Przykład (Python)

```
>>> def f(l):
...     l.append(1)
...
>>> m = []
>>> f(m)
>>> print m
[1]
>>> def f(l):
...     l += [1]
...
>>> m = []
>>> f(m)
>>> print m
[1]
>>> def f(l):
...     l = l + [1]
...
>>> m = []
>>> f(m)
>>> print m
[]
```

Zwracana wartość

Instrukcja `return result` powoduje natychmiastowe wyjście z funkcji. Funkcja zwraca wartość `result`.

Aby zwrócić wiele wartości na raz, trzeba zwrócić krotkę. Można korzystać z pakowania/rozpakowywania krotek.

Przykład

```
>>> def f():
...     return 1,2,3
...
>>> f()
(1, 2, 3)
>>> a,b,c=f()
```

Funkcje mogą wywoływać kolejne funkcje

```
def f():
    print "Jestem␣f."

def g():
    f()
    print "Jestem␣g."
```

...albo siebie same

```
def f():
    f()
    print "Jestem␣f."
```

Rekurencja

Recursion

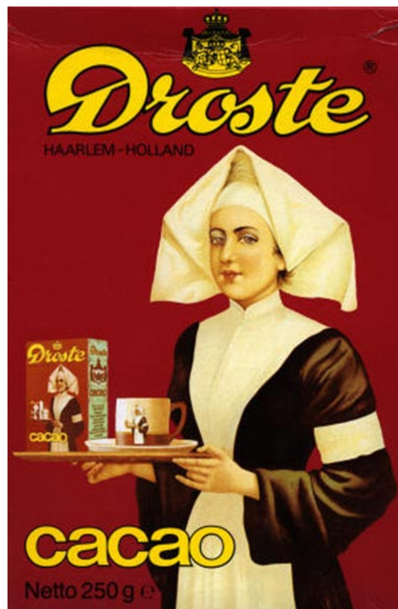
See "Recursion".

Rekurencja

Recursion

If you still don't get it, see "Recursion".

Efekt Droste



Silnia

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

$$silnia(i) = \begin{cases} 1 & i = 1 \\ silnia(i-1) \cdot i & \text{w p.p.} \end{cases}$$

Silnia

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

$$silnia(i) = \begin{cases} 1 & i = 1 \\ silnia(i-1) \cdot i & \text{w p.p.} \end{cases}$$

Rozwiązanie rekurencyjne

```
def silnia(n):
    if n==1:
        return 1
    else:
        return silnia(n-1)*n
```

Silnia c.d.

Rozwiązanie rekurencyjne

```
def silnia(n):
    if n==1:
        return 1
    else:
        return silnia(n-1)*n
```

Rozwiązanie iteracyjne

```
def silnia(n):
    res=1
    for i in range(1, n+1):
        res *= i

    return res
```

Uwagi

Zalety rekurencji

- prostota implementacji
- dobra do rozwiązywania problemów, które wymagają stosu
- często stosowana w metodzie "dziel i zwyciężaj"

Uwagi

Zalety rekurencji

- prostota implementacji
- dobra do rozwiązywania problemów, które wymagają stosu
- często stosowana w metodzie "dziel i zwyciężaj"

Wady rekurencji

- wywołanie funkcji jest drogie w językach preferujących konstrukcje iteracyjne
- może łatwo nastąpić przepełnienie

Uwagi

Zalety rekurencji

- prostota implementacji
- dobra do rozwiązywania problemów, które wymagają stosu
- często stosowana w metodzie "dziel i zwyciężaj"

Wady rekurencji

- wywołanie funkcji jest drogie w językach preferujących konstrukcje iteracyjne
- może łatwo nastąpić przepełnienie

Iteracja i rekurencja są wzajemnie równoważne.

Uwagi c.d.

Sensowne zastosowania

- Quicksort
- obchodzenie drzew (grafów)
- wieże Hanoi (ćw.)
- algorytmy "dziel i zwyciężaj"

Uwagi c.d.

Sensowne zastosowania

- Quicksort
- obchodzenie drzew (grafów)
- wieże Hanoi (ćw.)
- algorytmy "dziel i zwyciężaj"

Istnieją również rekurencyjne struktury danych (np. listy, drzewa).

Funkcje jako wartości

W Pythonie funkcje są traktowane jak wartości...

```
>>> f()
par: default
>>> f('aa')
par: aa
>>>
>>> g=f
>>> g()
par: default
>>> g('bb')
par: bb
>>> g
<function f at 0x10e5dc410>
>>> f
<function f at 0x10e5dc410>
>>> g
<function f at 0x10e5dc410>
>>> del f
>>> g()
par: default
```

Funkcje jako wartości c.d.

... i mogą być argumentami innych funkcji.

```
>>> def h(n, fun):
...     for i in range(n):
...         fun(i)
...
>>> h(5,f)
par: 0
par: 1
par: 2
par: 3
par: 4
>>>
```

Przykładowe zastosowania

- aplikowanie operacji do elementów sekwencji
- kryterium porównujące w sortowaniu

Projektowanie – dwie strategie

Top-down

Problem jest rozbijany na podproblemy, które można rozwiązać niezależnie. Podproblemy są dalej sukcesywnie dzielone do poziomu łatwo implementowalnych funkcji.

Bottom-up

Najpierw definiowane są podstawowe elementy, następnie składane są z nich coraz większe komponenty.

Projektowanie – dwie strategie c.d.

Top-down

- Brakujące elementy całości trzeba tymczasowo uzupełnić przed uruchomieniem.
- W naturalny sposób przechodzi od ogólnego zarysu do szczegółowych rozwiązań.
- Wymaga kompletnego projektu.
- Ułatwia delegowanie zadań i programowanie zespołowe.
- Pozwala poprawnie zaprojektować system od podstaw.

Bottom-up

- Szybko uzyskuje się fragmenty, które można testować.
- Trzeba z góry przewidzieć jakie komponenty są niezbędne.
- Można rozpocząć kodowanie bez ostatecznej koncepcji.
- Zwiększa prawdopodobieństwo, że bloki funkcjonalne będą uniwersalne.
- Pozwala rozwijać istniejący program.

Przykłady

Top-down

1. Wczytaj dane
2. Przetwórz
   1. Sprawdź poprawność danych
   2. Wykonaj obliczenie
3. Zapisz wynik

Przykłady

Bottom-up

1. Proste operacje (np. jakaś algebra - dodawanie wektorów, iloczyn skalarny)
2. Operacje na podzbiorach dziedziny (mnożenie wektora przez macierz, mnożenie macierzy, wyznacznik)
3. Bardziej złożone operacje (rozkłady macierzy)
4. Program rozwiązujący równania liniowe

Zadanie 1

Zaimplementuj funkcję obliczającą k-tą liczbę Fibonacciego.

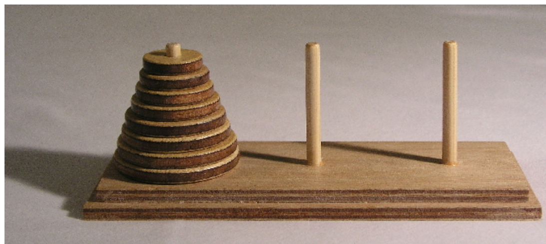
Zadanie 2

Napisz program losujący k liczb Fibonacciego nie większych od n.

Zadanie 3

Zadanie

Zaimplementuj algorytm rozwiązujący problem przenoszenia n krążków z pierwszego kołka na trzeci zgodnie z regułami.



Zadanie 4

Zaimplementuj sortowanie według zadanego porządku.

Zadanie 5

Zaimplementuj sortowanie według zadanego porządku algorytmem Quicksort.

Strona wykładu

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