`Programování` 1 pro matematiky

12. cvičení, pozdě v prosinci 2021

tags: `Programovani 1 2021`, `středa` `čtvrtek`

Obsah:

Farní oznamy

1. **Materiály k přednáškám** najdete v GitHub repozitáři https://github.com/PKvasnick/Programovani-1. Najdete tam také kód ke cvičením a pdf soubory textů cvičením.
2. **Kde se nacházíme** Tady jsem vložil několik věcí, ke kterým se mi zdálo užitečné vrátit. Je to text ke cvičení, které se těsně před Vánoci neuskutečnilo.

Opakování: Funkční objekty

Vysvětlili jsme si už, že funkce jsou plnoprávnými obyvateli Pythonu, a tedy je můžeme přiřazovat proměnným, ukládat do seznamů či slovníků, a používat je jako parametry funkcí.

Příklad 1. Součet položek ve dvou seznamech

Máme dva seznamy `a` a `b`, chceme seznam s položkami `a[0]+b[0], a[1]+b[1]` atd.

Možností je několik, liší se množstvím a čitelností kódu a také rychlostí.

```
# Součet položek ve dvou seznamech

a = [1, 2, 3, 4, 5]
b = [5, 4, 3, 2, 1]

# Definice funkce - příliš mnoho kódu pro jednoduchou věc
def soucet(x,y):
    return x+y

# List comprehension
print([soucet(x,y) for x,y in zip(a,b)])

# ale vlastně vůbec nepotřebujeme funkci!
print([x+y for x, y in zip(a,b)])

# Funkci ale potřebuje map - to je méně čitelné, ale výkonnější
print(list(map(soucet, a,b)))

# Čitelnější je použít namísto definované funkce soucet lambda funkci:
```

```
print(list(map(lambda x,y: x + y, a, b)))

# Základní funkce máme předdefinovány v modulu operator:
import operator
print(list(map(operator.add, a, b)))

```

Modul `operator` obsahuje funkční ekvivalenty pro běžné binární operátory:

Operation	Syntax	Function
Addition	`a + b`	`add(a, b)`
Concatenation	`seq1 + seq2`	`concat(seq1, seq2)`
Containment Test	`obj in seq`	`contains(seq, obj)`
Division	`a / b`	`truediv(a, b)`
Division	`a // b`	`floordiv(a, b)`
Bitwise And	`a & b`	`and_(a, b)`
Bitwise Exclusive Or	`a ^ b`	`xor(a, b)`
Bitwise Inversion	`~ a`	`invert(a)`
Bitwise Or	`a | b`	`or_(a, b)`
Exponentiation	`a ** b`	`pow(a, b)`
Identity	`a is b`	`is_(a, b)`
Identity	`a is not b`	`is_not(a, b)`
Indexed Assignment	`obj[k] = v`	`setitem(obj, k, v)`
Indexed Deletion	`del obj[k]`	`delitem(obj, k)`
Indexing	`obj[k]`	`getitem(obj, k)`
Left Shift	`a << b`	`lshift(a, b)`
Modulo	`a % b`	`mod(a, b)`
Multiplication	`a * b`	`mul(a, b)`
Matrix Multiplication	`a @ b`	`matmul(a, b)`
Negation (Arithmetic)	`- a`	`neg(a)`
Negation (Logical)	`not a`	`not_(a)`
Positive	`+ a`	`pos(a)`
Right Shift	`a >> b`	`rshift(a, b)`
Slice Assignment	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
Slice Deletion	`del seq[i:j]`	`delitem(seq, slice(i, j))`
Slicing	`seq[i:j]`	`getitem(seq, slice(i, j))`
String Formatting	`s % obj`	`mod(s, obj)`
Subtraction	`a - b`	`sub(a, b)`
Truth Test	`obj`	`truth(obj)`

Operation	Syntax	Function
Ordering	`a < b`	`lt(a, b)`
Ordering	`a <= b`	`le(a, b)`
Equality	`a == b`	`eq(a, b)`
Difference	`a != b`	`ne(a, b)`
Ordering	`a >= b`	`ge(a, b)`
Ordering	`a > b`	`gt(a, b)`

V modulu najdete i mnoho dalších užitečných věcí, takže neškodí nahlédnout do dokumentace.

Příklad 2: Třídění a další operace, vyžadující klíč

```
# Třídění a další operace, vyžadující klíč

# Klíč můžeme lehko definovat pomocí lambda funkce.
>>> k = ["kočka", "sedí", "na", "okně"]
>>> sorted(k, key=lambda x: len(x))

['na', 'sedí', 'okně', 'kočka']

# Pomocí lambda funkce můžeme definovat i složitější klíč:
>>> k = ["kočka", "sedí", "na", "okně"]
>>> sorted(k, key=lambda x: (len(x), x))

['na', 'okně', 'sedí', 'kočka']

# Funkce min také srovnává a tedy u ní můžeme definovat klíč:
>>> min(k, key=lambda x: len(x))

'na'

# Setřídění podle položky jsme už trénovali:
>>> p = [(1,'leden'), (2,'unor'), (4,'duben')]
>>> sorted(p, key=lambda x: x[1])

[(4, 'duben'), (1, 'leden'), (2, 'unor')]

# Konečně si můžeme vypomoct modulemt operator:
>>> import operator
>>> sorted(p, key = operator.itemgetter(1))

[(4, 'duben'), (1, 'leden'), (2, 'unor')]
```

Příklad 3: Implementace funkce `itemgetter`

Ukážeme si dvě možné implementace pro `oparator.itemgetter` . První možností je funkce, která vrací funkci:

```
# itemgetter as a function

```

```
def itemgetter(k):
    return lambda a: a[k]

def main():
    a = (1,2)
    print(itemgetter(1)(a))
    u = [(1,5), (2,4), (3,3)]
    print(sorted(u, key = itemgetter(1)))
    print(sorted(u, key = itemgetter(0)))

if __name__ == "__main__":
    main()

2
[(3, 3), (2, 4), (1, 5)]
[(1, 5), (2, 4), (3, 3)]
```

Druhou možností je implementovat `itemgetter` jako funktor, tedy objekt, který lze volat jako funkci:

```
# itemgetter as functor

class itemgetter:
    def __init__(self, k):
        self.k = k
        self.fun = lambda a: a[self.k]

    def __call__(self, a):
        return self.fun(a)

def main():
    a = (1,2)
    print(itemgetter(1)(a))
    u = [(1,5), (2,4), (3,3)]
    print(sorted(u, key = itemgetter(1)))
    print(sorted(u, key = itemgetter(0)))

if __name__ == "__main__":
    main()

2
[(3, 3), (2, 4), (1, 5)]
[(1, 5), (2, 4), (3, 3)]
```

Příklad 4: Kompozice funkcí

Mějme funkce `f(x)` a `g(x)`, chceme implementovat funkci `compose(f,g)`, která vrací složenou funkci `f∘g`, tedy funkci, vracející hodnotu `f(g(x))` a ne hodnotu této funkce pro nějaký argument.

```
def compose(f,g):
    def _comp(x):
        return f(g(x))
    return _comp

print(compose(int, abs)(-4.5))

```

Vnitřní funkce

Už jsme několikrát viděli, že vytváření funkcí jinými funkcemi je docela silná zbraň, zejména díky tomu, že vytvořené funkce si sebou nesou prostředí mateřské funkce ve stavu svého vzniku. Tím se podobají na třídy.

```
def f():
    n = 0
    def in_f():
        nonlocal n   # n pochází z nadřazeného jmenného prostoru
            n += 1   # musíme explicitně deklarovat, protože používáme
        return n     # na pravé straně výrazu
    return in_f

>>> a = f()
>>> a()
1
>>> a()
2
>>> b = f()
>>> b()
1
>>> b()
2
>>> a()
3
>>>
```

Zjevně funkce `a()` a `b()` mají nezávislé vnitřní stavy.