`Programování` 1 pro matematiky

12. cvičení, 17-12-2024

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. **Domácí úkoly** - přišla dvě řešení úlohy o inverzní permutaci. Nebudu zadávat další domácí úkoly.
3. **Kde se nacházíme** Dnes soubory a výjimky, cvičení v lednu uděláme dálkovou formou: vyložím do repozitáře materiál ke studiu. Na konci semestru zapíšu všem zápočet.

Domácí úkoly

Inverzní permutace

> **Inverzní permutace** pro danou permutaci je takové promíchání čísel, které čísla vrátí do původního pořadí. Příklad: inverzní permutace pro permutaci
>
> ```
> 1 -> 1
> 2 -> 3
> 3 -> 6
> 4 -> 2
> 5 -> 5
> 6 -> 4
> ```
>
> je
>
> ```
> 1 -> 1
> 2 -> 4
> 3 -> 2
> 4 -> 6
> 5 -> 5
> 6 -> 3
> ```

Řešení je zřejmé, je potřeba vyměnit indexy a hodnoty v poli.

Řešení od Martina Bláhy:

```
data = list(map(int, input().split()))

n = data[0]
p = data[1:]

inv_p = [0] * n

for i in range(n):
    inv_p[p[i] - 1] = i + 1

print(" ".join(map(str, inv_p)))
```

Malé vylepšení by se dalo dosáhnout zahrnutím položek s indexem 0 tak, aby se nemuseli posouvat indexy o +/- 1:

```
p = list(map(int, input().split()))

n = p[0] # na indexu 0 máme n, ale to nepřekáží

inv_p = [0] * (n+1) # o 1 delší, aby položky šly na indexy 1..n.

for i in range(1,n+1):
    inv_p[p[i]] = i  # nic nemusíme posouvat

print(" ".join(map(str, inv_p[1:]))) # nakonec nedestruktívně vynecháme položku 0.
```

Maticová třída

Tento domácí úkol měl sloužit k procvičení vytváření tříd a jejich metod, a řešení mohlo ve velice zjednodušené verzi vypadat nějak takto:

```
class Matrix:

    def __init__(self, ll):
        self.matrix = ll

    def __repr__(self):
        return '\n'.join(' '.join(str(val) for val in row) for row in self.matrix)

    def vals(self):
        return self.matrix

    def dims(self):
        return len(self.matrix), len(self.matrix[0])

    def __add__(self, other):
        if self.dims() != other.dims():
            raise ValueError("Sečítat lze pouze matice stejných rozměrů")
        result = []
        for i, row in enumerate(self.matrix):
            result.append([x + y for x, y in zip(row, other.matrix[i])])
        return Matrix(result)

    def __sub__(self, other):
```

```
        if self.dims() != other.dims():
            raise ValueError("Odečítat lze pouze matice stejných rozměrů")
        result = []
        for i, row in enumerate(self.matrix):
            result.append([x - y for x, y in zip(row, other.matrix[i])])
        return Matrix(result)

    def __mul__(self, other):
        if isinstance(other, Matrix):
            if self.dims()[1] != other.dims()[0]:
                raise ValueError("Násobit lze pouze matice kompatibilních rozměrů")
            result = [[0] * len(other.matrix[0]) for _ in range(len(self.matrix))]
            for i in range(len(self.matrix)):
                for j in range(len(other.matrix[0])):
                    for k in range(len(other.matrix)):
                        result[i][j] += self.matrix[i][k] * other.matrix[k][j]
            return Matrix(result)
        else:
            result = [[x * other for x in row] for row in self.matrix]
            return Matrix(result)

    def is_symmetric(self):
        if len(self.matrix) != len(self.matrix[0]):
            return False
        for i in range(len(self.matrix)):
            for j in range(i + 1, len(self.matrix[0])):
                if self.matrix[i][j] != self.matrix[j][i]:
                    return False
        return True


def zero_matrix(r, c):
    return Matrix([[0] * c for _ in range(r)])


def identity_matrix(n):
    return Matrix([[1 if i == j else 0 for j in range(n)] for i in range(n)])
```

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)`

Operation	Syntax	Function
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)`
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']

# 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.

Soubory: čtení a zápis

Souborem myslíme nějakou skupinu bajtů, uloženou pod svým názvem v souborovém systému.

Budeme se zabývat **textovými soubory**, v nichž bajty reprezenutjí znaky v nějakém kódování.

- *ASCII* (”anglická abeceda“ o 95 znacích)
- *iso-8859-2* (navíc znaky východoevropských jazyků)

- *cp1250* (něco podobné, specifické pro Windows)
- *UTF-8* (vícebajtové znaky, pokrývají většinu glyfů a jazyků světa

Kódování je všudypřítomné, nevyhnete se problémům s explicitním uvedením kódování nebo s převodem. Neexistuje nic takového jako defaultní kódování textového souboru, i když například pro kód v Pythonu je defaultním kódováním UTF-8.

Python má rozsáhlou podporu kódování a většinu problémů jde jednoduše řešit.

Python načte textový soubor jako kolekci řádků. Naopak, při zápisu musíme konce řádků zapsat tam, kam patří:

```
f = open("soubor.txt", "w") # "w" jako write, "r" jako read
f.write("Hej, mistře!\n")
f.close()
```

Protože komunikujeme se systémovými službami a operačním systémem, může se při zápisu nebo čtení lehce stát něco neočekávaného - nejde vytvořit soubor, do kterého chcete zapisovat, soubor na čtení neexistuje tam, kde ho hledáte a podobně. Pokud chceme, aby nám v takovýchto situacích neskončil program s chybou, ale nějak se se situací graciézně vypořádal, potřebujeme nástroje na *obsluhu výjimek* a o nich budeme mluvit za chvíli.

U čtení zápisu bychom rádi měli jistotu, že ať se stane cokoli, soubor se zavře. Proto standardně obsluhujeme soubory pomocí **kontexotvého manažéra** takto:

```
with open("soubor.txt", "w") as f:
    f.write("Hej, mistře!\n")
```

`f.close()` se zavolá automaticky po opuštění bloku `with` a to i v případě, že se stane něco neočekávaného.

Metody souborů

`f.write(text)` – zapíše text
`f.read(n)` –přečte dalších n znaků, na konci " ".
`f.read()` – přečte zbývající znaky souboru
`f.readline()` – přečte další řádek (včetně "\n") nebo " ".
`f.seek(...)` – přesune se na další pozici v souboru

Další operace:
`print(..., file=f)`
`for line in f:` – cyklus přes řádky souboru
Pozor, řádky končí "\n", hodí se zavolat `rstrip()`.

Vždy je k dispozici:

`sys.stdin` – standardní vstup (odtud čte `input()`)
`sys.stdout` – standardní vstup (sem píše `print()`)
`sys.stderr` – standardní chybový výstup

```
>>> sys.stdout.write("Hej, mistre!\n")
Hej, mistre!
13
```

Modul `pathlib`

Toto je základní modul pro pohodlný přístup a manipulaci se souborovým systémem na Unixu nebo ve Windows.

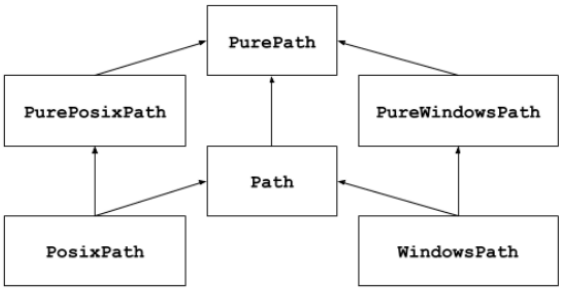
Table of Contents

`pathlib` — Object-oriented filesystem paths ¶

Added in version 3.4.

Source code: Lib/pathlib/

This module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between pure paths, which provide purely computational operations without I/O, and concrete paths, which inherit from pure paths but also provide I/O operations.



```
>>> from pathlib import Path

>>> p = Path(".")
>>> p
WindowsPath('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[WindowsPath('.idea')]
>>> list(p.glob('**/*.py'))
[WindowsPath('eqn_solve.py'), WindowsPath('je_prvocislo.py'),
WindowsPath('zeros_back.py'), WindowsPath('zvire.py')]

>>> p = Path("/")
[x for x in p.iterdir() if x.is_dir()]
[WindowsPath('/$Recycle.Bin'), WindowsPath('/Config.Msi'), WindowsPath('/Documents
and Settings'), WindowsPath('/eSupport'), WindowsPath('/OneDriveTemp'),
WindowsPath('/PerfLogs'), WindowsPath('/Program Files'), WindowsPath('/Program Files
(x86)'), WindowsPath('/ProgramData'), WindowsPath('/R'), WindowsPath('/Recovery'),
WindowsPath('/System Volume Information'), WindowsPath('/Users'),
WindowsPath('/Windows')]
>>> q = p / "Users" / "kvasn"
>>> q
WindowsPath('/Users/kvasn')
```

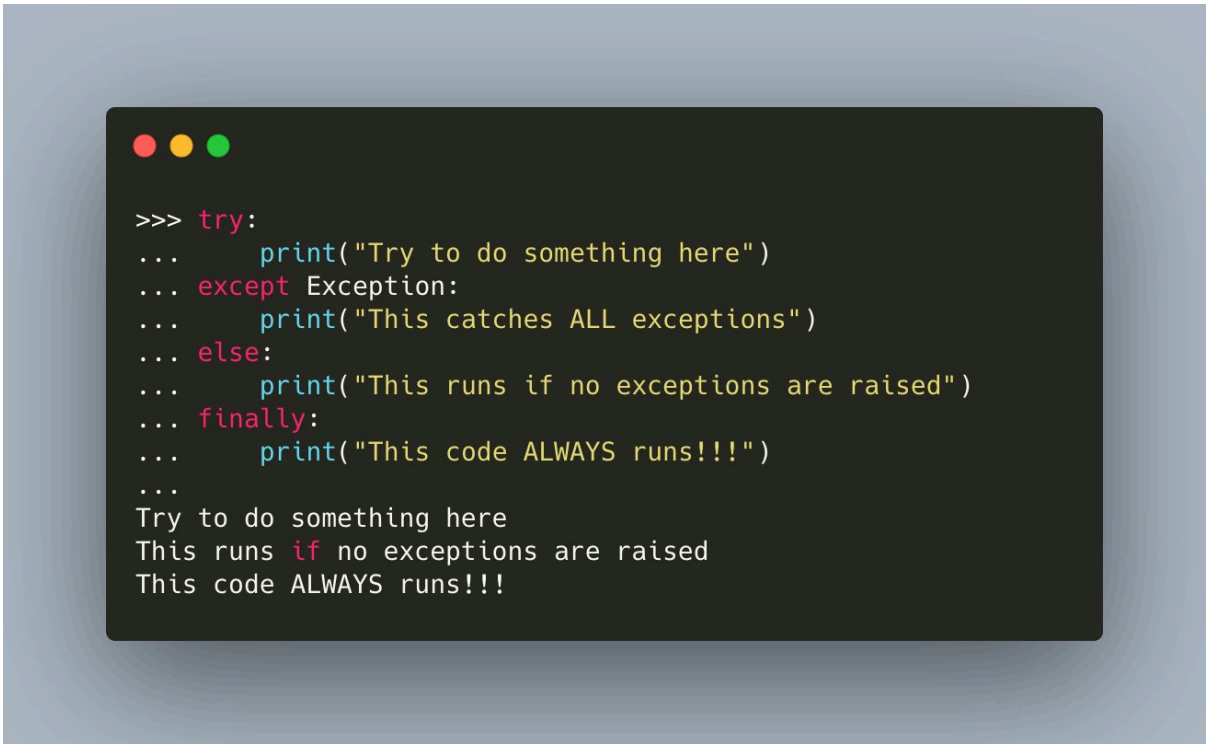
Chyby a výjimky

```
def divide(x, y):
    return x/y

divide(1, 0)

Traceback (most recent call last):
  File "<pyshell#73>", line 1, in <module>
    divide(1,0)
  File "<pyshell#72>", line 2, in divide
    return x/y
ZeroDivisionError: division by zero
```

Chyba vygeneruje výjimku, např.
`ZeroDivisionError` – dělení nulou
`ValueError` – chybný arguement
`IndexError` – přístup k indexu mimo rozsah
`KeyError` – dotaz na hodnotu neexistujícího klíče ve slovníku
`FileNotFoundError` – pokus o otevření neexistujícího souboru ke čtení
`MemoryError` – vyčerpání dostupné paměti
`KeyboardInterrupt` – běh programu byl přerušen stiskem `Ctrl-C`
`StopIteration` - žádost o novou hodnotu z vyčerpaného iterátoru

```
try:
    x, y = map(int, input().split())
    print(x/y)
except ZeroDivisionError:
    print("Nulou dělit neumím.")
except ValueError as ve:
    print("Chyba:", ve)
    print("Zadejte prosím dvě čísla.")
```

Obecně je syntaxe takováto:



Výjimky jsou objekty, jejich typy jsou třídy.

Výjimka se umí vypsat příkazem `print`

Atributy výjimky obsahují dodatečné informace o tom, co a kde se stalo.

Výjimky tvoří hierarchie, například `FileNotFoundError` je potomkem `IOError`. Můžeme zachytit obecnější typ a doptat se, o kterého potomka se jedná.

```
>>> raise RuntimeError("Jejda!")
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in <module>
    raise RuntimeError("Jejda!")
RuntimeError: Jejda!

>>> assert 1 == 2
Traceback (most recent call last):
  File "<pyshell#78>", line 1, in <module>
    assert 1 == 2
AssertionError

>>> assert 1 == 2, "Pravda už není, co bývala!"
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    assert 1 == 2, "Pravda už není, co bývala!"
AssertionError: Pravda už není, co bývala!
```