

11. cvičení, 20-12-2023

Obsah:

- 0. Farní oznamy
- 1. Domácí úkoly
- 2. Opakování: Třídy
- 3. Soubory a výjimky

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

- Tímto týdnem končíme. Další domácí úkoly budou bonusové a nebudou se započítávat do celkového limitu.
- Limit počtu bodů pro tento semestr je $0.5 \times 270 = 135$ bodů. Kdo má tolik anebo víc, má zápočet.
- Několik z vás by mírným úsilím mohli dosáhnout na zápočet. Napište mi a něco vymyslíme.

Kde se nacházíme

Nové téma pro dnešek bude čtení a zápis do souborů a obsluha výjimek.

Domácí úkoly

Přišlo opravdu málo řešení. Úkoly byly jednoduché, stačilo použít standardní postupy.

Standardní postup pro počítání výskytů věcí v posloupnosti je *slovník*. Prototypová úloha byla úloha o nalezení modusu posloupnosti, tedy prvku s největším výskytem.

- Obyčejný slovník nebo *defaultdict*: modus naleznete jako `max(dic.items(), key = lambda x: x[2])`.
- *Counter* má speciální metody pro vylistování největších prvků.

```
>>> from collections import Counter
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

Účelem kurzu programování není pouze vás naučit, jak napsat cyklus `for`. Měli byste také znát některé základní postupy, například binární vyhledávání nebo počítání věcí.

Opakování: Třídy

Třídy nám umožňují seskupit data a funkce, které na nich operují a zpřístupňují je, a zároveň "schovat" detaily implementace. Třída je datový typ, od kterého si vytváříme instance.

```
# Třídy

class Zvire():

    def __init__(self, jmeno, zvuk):
        self.jmeno = jmeno
        self.zvuk = zvuk

    def slysi_na(self, jmeno):
        return self.jmeno == jmeno

    def ozvi_se(self):
        print(f"{self.jmeno} říká: {self.zvuk}")

...
>>> pes = Zvire("Puntča", "Hafff!")
>>> pes
<__main__.Zvire object at 0x000001A01A391B80>
>>> pes.slysi_na("Miau")
False
>>> pes.ozvi_se()
Puntča říká: Hafff!
>>> kocka = Zvire("Mourek", "Miau!")
>>> kocka.ozvi_se()
Mourek říká: Miau!
```

`self` nás odkazuje na instanci třídy.

`__init__()` je metoda, která vytváří instanci ze vstupních dat - *konstruktor*.

Metod s dvojími podtržítky existuje mnoho. Jsou to metody, které definují standardní aspekty objektů.

Vlastnosti a metody

```
>>> azor = Zvire("Azor", "Haf!")
>>> azor
<__main__.Zvire object at 0x00000214E4303D00>
>>> azor.jmeno
'Azor'
>>> azor.zvuk
'Haf!'
>>> azor.zvuk = "Haffff!"
>>> azor.slysi_na("azor")
False
>>> azor.ozvi_se()
Azor říká: Haffff!
```

Magické neboli "dunder" metody

Metody s názvy `__metoda__()` jsou metody zděděné od pythonského praobjektu `object` a definují základní chování každé třídy.

- `__new__()` a `__del__()` - vytvoření a smazání instance (`__new__()` potřebujeme pouze ve speciálních případech, kdy k vytvoření instance nestačí defaultní `__new__`, automaticky volané přes `object.__init__()`)
- `__str__()` je to, co používá funkce `print`
- `__repr__()` je to, co vypíše Pythonská konzole jako identifikaci objektu.
- Konverze na bool, str, int, float: `__bool__()`, atd., operace `__add__()`, `__subtract__` atd.
- Indexování `objekt[i]`: `__getitem__()`, `__len__()` atd.
- Volání jako funkce `objekt(x)`: `__call__()`
- Iterátor pro `for x in objekt:` `__iter__()`

```
class Zvire():

    def __init__(self, jmeno, zvuk):
        self.jmeno = jmeno
        self.zvuk = zvuk

    ...

    def __str__(self):
        return self.jmeno

    def __repr__(self):
        return f"Zvire({self.jmeno}, {self.zvuk})"

    def __eq__(self, other):
        return self.jmeno == other.jmeno and \
            self.zvuk == other.zvuk

    ...

>>> pes = Zvire("Punta", "haf!")
>>> pes
Zvire(Punta, haf!)
>>> print(pes)
Punta
>>> kocka = Zvire("Mourek", "Miau!")
>>> pes == kocka
False
```

Dokumentační řetězec

```

class Zvire():
    """vytvoří zvíře s danými vlastnostmi"""

    def __init__(self, jmeno, zvuk):
        self.jmeno = jmeno
        self.zvuk = zvuk

    ...

>>> help(Zvire)
>>> lenochod = Zvire("lenochod", "zzzz...")
>>> help(lenochod.slysi_na)

```

Dataclasses

```

from dataclasses import dataclass

@dataclass
class Osoba:
    jmeno: str
    prijmeni: str
    rok_narozeni: int

petr = Osoba('Petr', 'Novák', 1980)
marie = Osoba('Marie', 'Zahradníková', 1988)

print(petr)
print(marie)

```

`@dataclass` je dekorátor. Je to jenom syntaxe pro napsání, že naši definovanou třídu pošleme jako parametr funkci `dataclass`, která z něj vytvoří jiný objekt, který budeme používat pod jménem `Osoba`.

Dekorátor je funkce, a kromě vstupního objektu může obsahovat itaké další parametry:

```

@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False,
            frozen=False,
            match_args=True, kw_only=False, slots=False, weakref_slot=False)
class C:
    ...

```

Účel parametrů je víceméně zřejmý z názvů, zbytek si prosím najděte v dokumentaci. Tam najdete hromadu dalších věcí.

```

from dataclasses import dataclass

@dataclass(order=True)
class Osoba:
    jmeno: str
    prijmeni: str
    rok_narozeni: int

petr = Osoba('Petr', 'Novák', 1980)
marie = Osoba('Marie', 'Zahradníková', 1988)

print(petr)
print(marie)
print(petr < marie)

```

Dědičnost

```

class Kocka(Zvire):

    def __init__(self, jmeno, zvuk):
        Zvire.__init__(self, jmeno, zvuk)
        self._pocet_zivotu = 9 # interní

    def slysi_na(self, jmeno):
        # Copak kočka slyší na jméno?
        return False

...

>>> k = Kocka("Příšerka", "Mňauuu")
>>> k.slysi_na("Příšerka") (speciální kočičí verze)
False
>>> k.ozvi_se() (původní zvířecí metoda)
Příšerka říká: Mňauuu

```

Typy

```

>>> type(k) is Kocka
True
>>> type(k) is Zvire
False
>>> isinstance(k, Kocka)
True
>>> isinstance(k, Zvire)
True
>>> issubclass(Kocka, Zvire)
True

```

Prostory a rozsahy platnosti

Co dělá Python, když chce zjistit, kterou metodu třídy má volat?

Prostory jmen, **namespaces**:

- Zabudované funkce (print) `builtins`
- Globální jména - proměnné a funkce, definované mimo jakoukoli funkci nebo třídu `globals`
- Lokální jména definovaná při aktuálním volání uvnitř aktuální funkce `locals`
- Jména definovaná v aktuální třídě
- Jména definovaná v aktuálním objektu

Oblasti platnosti, **scopes**

Obyčejné jméno se hledá ve všech prostorech jmen, které jsou z daného kontextu vidět - lokální, globální, zabudované proměnné.

`objekt.jméno` se hledá

- mezi atributy objektu
- mezi atributy třídy
- mezi atributy nadřazených tříd

Soubory: čtení a zápis

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

Budeme se zabývat **textovými soubory**, v nichž bajty reprezentují 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 graciálně vypořádal, potřebujeme nástroje na *obsahu 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 **kontextového manažera** 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 výstup (sem píše `print()`)

`sys.stderr` – standardní chybový výstup

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

Chyby a výjimky

```
def divide(x, y):  
    return x/y  
  
divide(1, 0)  
  
Traceback (most recent call last):  
  File "<pyshe11#73>", line 1, in <module>  
    divide(1,0)  
  File "<pyshe11#72>", line 2, in divide  
    return x/y  
ZeroDivisionError: division by zero
```

Chyba vygeneruje výjimku, např.

`ZeroDivisionError` – dělení nulou

`ValueError` – chybný argument

`IndexError` – přístup k indexu mimo rozsah

`KeyError` – dotaz na hodnotu neexistujícího klíče ve slovníku

`FileNotFoundError` – pokus o otevření neexistujícího souboru ke čtení

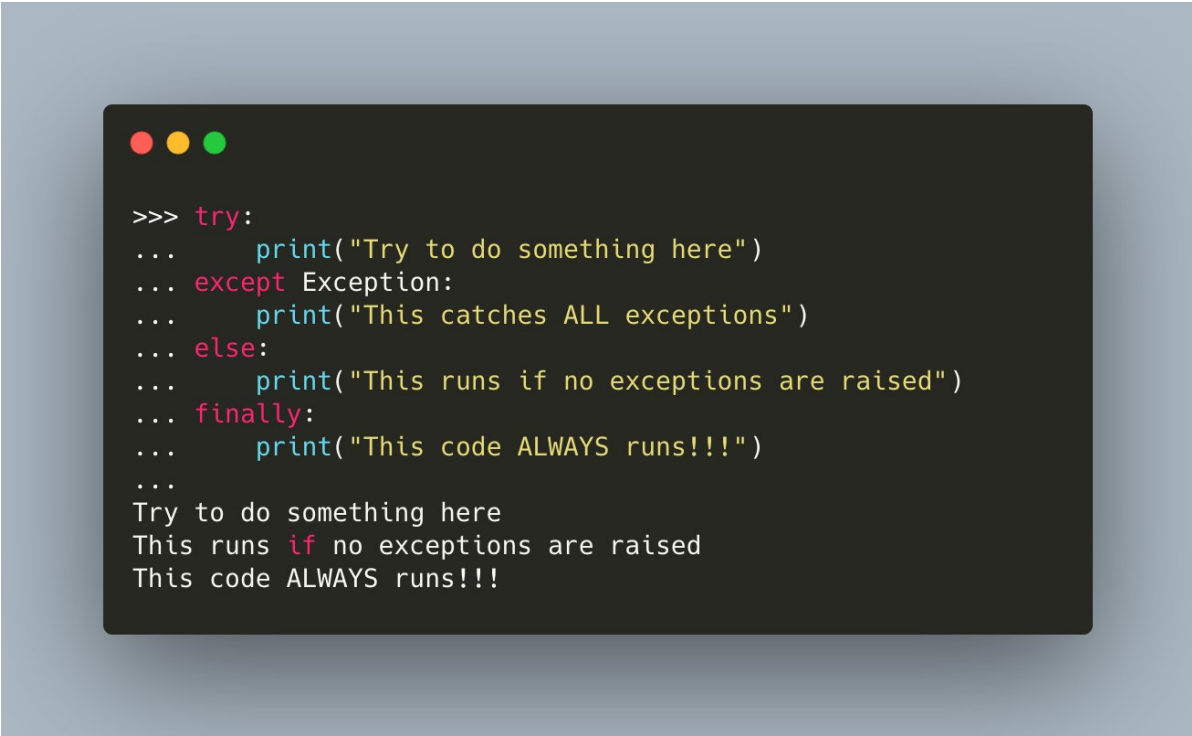
`MemoryError` – vyčerpání dostupné paměti

`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:



```
>>> try:
...     print("Try to do something here")
... except Exception:
...     print("This catches ALL exceptions")
... else:
...     print("This runs if no exceptions are raised")
... finally:
...     print("This code ALWAYS runs!!!")
...
Try to do something here
This runs if no exceptions are raised
This code ALWAYS runs!!!
```

Výjimky jsou objekty, jejich typy jsou třídy.

Výjimka se umí vypsát 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 "<pyshe11#75>", line 1, in <module>
    raise RuntimeError("Jejda!")
```


RuntimeError: Jejda!

```
>>> assert 1 == 2
```

Traceback (most recent call last):

File "<pyshe11#78>", line 1, in <module>

```
    assert 1 == 2
```

AssertionError

```
>>> assert 1 == 2, "Pravda už není, co bývala!"
```

Traceback (most recent call last):

File "<pyshe11#79>", line 1, in <module>

```
    assert 1 == 2, "Pravda už není, co bývala!"
```

AssertionError: Pravda už není, co bývala!