`Programování` 1 pro matematiky

11. cvičení, 15/16-12-2021

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. **Domácí úkoly**
    - O víkendu jste dostali novou trojici domácích úkolů.
    - Další trojice bude následovat
    - Dáme nějakou nápovědu
3. **Opakování** velice rychle třídy a datové objekty

Kde se nacházíme

Dnešní hlavní téma bude čtení a zápis do souborů a obsluha výjimek.

Domácí úkoly

1. Mroží operátor :=

Při načítání data v domácích úkolech používáte typicky cyklus while s neuspokojivým uspořádáním logiky:

```
lines = []
while True:         # Nevyužíváme podmínku cyklu a testujeme v těle cyklu
    line = input()
    if "---" in line:
        break
    lines.append(line)
text = "\n".join(lines)
```

anebo

```
lines = []
line = input()
while "---" not in line:
    lines.append(line)
    line = input()  # Využíváme podmínku cyklu, ale máme opačně tok kódu.
text = "\n".join(lines)
```

Důvod, proč si musíme vypomáhat úpravou logiky je, že nemůžeme v logickém výrazu, testujícím přítomnost koncového řetězce, zároveň provést přiřazení.

```
while "---" not in input(): # Nefunguje, ztratili bychom výstup input()
    ...
```

Mroží operátor := (walrus operator) je řešení právě na tuto situaci. Nedověděli jste se o něm dříve proto, že jde o prostředek, který není dobré nadužívat. Náš kód s jeho pomocí bude vypadat takto:

```
lines = []
while "---" not in (line := input():
    lines.append(line)
text = "\n".join(lines)
```

Mroží operátor nám tedy umožňuje "ukrást" si část výrazu do proměnné. Na použití mrožího operátoru se vztahují určitá omezení, například jej nelze použít v lambda funkci, alespoň ne jednoduše. Používat taková ukradená přiřazení není obecně dobrá praxe.

2. Počítání věcí v Pythonu: Třída `collections.Counter`

U slovníků jsme viděli, že jejich důležitou aplikací je počítání věcí. Naučili jsme se používat `collections.defaultdict`, ale modul `collections` obsahuje také specializovanou třídu `Counter`, více zde.

```
from collections import Counter

s = "abrakadabra"
ctr = Counter(s)
print(ctr)

Counter({'a': 5, 'b': 2, 'r': 2, 'k': 1, 'd': 1})
```

3. Generátor podřetězců

Úloha o nejdelším společném podřetězci má docela velký význam, a pro případ *souvislých* podřetězců není obtížná. Opravdu důležitá a opravdu těžká úloha je najít nejdelší společnou podposloupnost znaků v řetězci.

Souvislé podřetězce daného řetězce vygeneroujeme snadno:

```
def get_substrings(source):
    for i in range(len(source)):
        for j in range(i, len(source)+1):
            yield source[i:j]

print(list(get_substrings("abrakadabra")))
```

```
['', 'a', 'ab', 'abr', 'abra', 'abrak', 'abraka', 'abrakad', 'abrakada',
'abrakadab', 'abrakadabr', 'abrakadabra', '', 'b', 'br', 'bra', 'brak',
'braka', 'brakad', 'brakada', 'brakadab', 'brakadabr', 'brakadabra', '', 'r',
'ra', 'rak', 'raka', 'rakad', 'rakada', 'rakadab', 'rakadabr', 'rakadabra',
'', 'a', 'ak', 'aka', 'akad', 'akada', 'akadab', 'akadabr', 'akadabra', '',
'k', 'ka', 'kad', 'kada', 'kadab', 'kadabr', 'kadabra', '', 'a', 'ad', 'ada',
'adab', 'adabr', 'adabra', '', 'd', 'da', 'dab', 'dabr', 'dabra', '', 'a',
'ab', 'abr', 'abra', '', 'b', 'br', 'bra', '', 'r', 'ra', '', 'a']
```

Při této úloze musíme uvažovat v termínech velice dlouhých řetězců. Proto používáme generátor, a proto neřešíme, že naše metoda bude produkovat duplicitné podřetězce.

Nebudeme generovat podřetězce obou řetězců, protože pak bychom si museli oba sety podřetězců pamatovat. Namísto toho můžeme pro každý podřetězec jednoho řetězce zjistit, zda se nachází v druhém řetězci.

Takovéto zjišťování pak lze výrazně zefektivnit, pokud bychom dokázali postupně generovat podřetězce s danou velikostí. To lze také udělat snadno, a přináší to dvě podstatné výhody:

- Pokud pro danou délku nenajdeme společný podřetězec, nemusíme hledat pro větší délky.
- Pokud společný podřetězec pro danou dělku najdeme, nemusíme hledat dále a můžeme přejít k větší délce.

```
def get_substrings(s:str, k:int):
    if k == 0:
        yield ""
    elif k > 0:
        for i in range(len(s) - k + 1):
            yield s[i:i+k]


def find_longest_substring(s1, s2):
    max_substring = ''
    for size in range(1, min(len(s1), len(s2))):
        subs_found = 0
        for sub in get_substrings(s1, size):
            if sub in s2:
                subs_found += 1
                max_substring = sub
                break
        if subs_found == 0:
            break
    return max_substring

```

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("Punťa", "Hafff!")
>>> pes
<__main__.Zvire object at 0x000001A01A391B80>
>>> pes.slysi_na("Miau")
False
>>> pes.ozvi_se()
Punťa 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 dvojitými 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!
```

Identita objektu

```
>>> jezevcik = Zvire("˘Spagetka", "haf")
>>> bernardyn = Zvire("Bernard", "HAF!!!")
>>> maxipes = bernardyn
```

```
>>> maxipes.jmeno = "F´ık"
>>> bernardyn.jmeno
'F´ık'
>>> type(jezevcik)
<class 'Zvire'>
>>> id(jezevcik), id(bernardyn), id(maxipes)
(737339253592, 737339253704, 737339253704)
>>> bernardyn is maxipes
True
>>> bernardyn is jezevcik
False
```

Znaková reprezentace objektu

`__str__()` je to, co používá funkce `print`

`__repr__()` je to, co vypíše Pythonská konzole jako identifikaci objektu.

```
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})"

    ...
>>> pes = Zvire("Punta", "haf!")
>>> pes
Zvire(Punta, haf!)
>>> print(pes)
Punta
```

Protokoly pro operátory

```
class Zvire():

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

...

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

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

```
False
```

Podobně lze předefinovat řadu dalších operátorů:

- Konverze na bool, str, int, float
- Indexování `objekt[i], len(i)`, čtení, zápis, mazání.
- Přístup k atributům `objekt.klíč`
- Volání jako funkce `objekt(x)`
- Iterátor pro `for x in objekt:`

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)

```

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 ko˘cka sly˘s´ı na jm´eno?
        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

(konec opakování)

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* (”anglick´a abeceda“ o 95 znac´ıch)
- *iso-8859-2* (nav´ıc znaky v´ychodoevropsk´ych jazyk° u)
- *cp1250* (n ˇeco podobn´eho specifick´eho pro Windows)
- *UTF-8* (v´ıcebajtov´e znaky, pokr´yv´a vˇ etˇsinu jazyk°u svˇ eta

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.

```
f = open("soubor.txt", "w")
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
```

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 x:
    print("Chyba:", x)
    print("Zadejte prosím dvě čísla.")
```

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!
```