Programování 2

4. cvičení, 12-3-2024

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na https://github.com/PKvasnick/Programovani-2.
2. **Domácí úkoly**:
    - Tři domácí úkoly pro ttento týden měli různou obtížnost:
        - Maximum posloupnosti a jeho výskyty - velmi lehký
        - Počet hostů v restauraci: lehký
        - Frrekvence slov: středně těžký
    - Samozřejmě, že si o nich promluvíme, ale platí upozornění z minulého týdne: pokud máte pocit, že vám programování přerůstá přes hlavu, ozvěte se.

Dnešní program:

- Kvíz
- Domácí úkoly
- Třídění

Na zahřátí

> *"Code is like humor. When you have to explain it, it's bad." – Cory House*

Dobrý kód nepotřebuje mnoho komentářů (ale někdy potřebuje).

Co dělá tento kód

```
d = dict.fromkeys(range(10), [])
for k in d:
    d[k].append(k)
d
```

`dict.fromkeys` vytváří z kolekce klíče slovníku a případně jim přiřadí hodnotu. Je oto někdy dobrá náhrada `defaultdict` nebo `Counter`. Je to dobrá metoda, pokud potřebujete klíče ve specifické pořadí.

Tato konkrétní inicializace slovníku je ale velmi špatný nápad.

Počítání věcí v Pythonu

- `collections.defaultdict`
- `collections.Counter`

V principu se bez těchto speciálních udělátek lehce obejdeme, ale jejich použití zvyšuje robustnost kódu. Obě jste měli možnost si vyzkoušet u úkolu o frekvenci slov.

Pojmenované n-tice (ještě jednou)

Posledně jsme mluvili o `collections.namedtuple`

```
from collections import namedtuple

Point = namedtuple("Point", "x y")
point = Point(5, 6)
point.x
Out[5]: 5
point.y
Out[6]: 6
```

Čistější možnost:

```
from typing import NamedTuple


class Point(NamedTuple):
    x : float
    y : float


point = Point(1.2, 3.4)

print(f"{point.x=}, {point.y=}")
print(f"{point[0]=}, {point[1]=}")

point.x = 1.5
-----------
point.x=1.2, point.y=3.4
point[0]=1.2, point[1]=3.4
Traceback (most recent call last):
  File "C:\Users\kvasn\Documents\GitHub\Programovani-
2\code\Ex04\namedtuple_2.py", line 14, in <module>
    point.x = 1.5
AttributeError: can't set attribute
```

Výraznější a čistší definice, i když prakticky stejná funkcionalita.

Domácí úkoly

Několik poznámek:

- poučení o detekci koncového řetězce při načítání neplatilo jen pro první dvě cvičení, ale platí pořád.

- když se vám zdá, že váš kód drhne - že se vám špatně programuje - zkuste všechno zahodit a začít znovu.
- buďte úsporní a praktičtí: držte se faktů (zadání)
- udržujte si pořádek: koště nepatří do obýváku.

Cyklický kontejner

Jak si pamatovat počet hostů, kteří přišli před k hodinami nebo později?

1. Použijeme cyklický index: počet hostů, přicházejících v n-té hodině napíšeme do položky s indexem `n % k`. Tak se všechno správně přepisuje právě po k hodinách.
2. `collections.deque`

```
from collections import deque

data = list(range(30))
buffsize = 6
buffer = deque([0]*buffsize)

print(buffer)

for d in data:
    buffer.appendleft(d)
    buffer.pop()
    print(buffer)
```

O velkých a malých písmenech

```
uzivatele = [
    "johndoe@gmail.com",
    "JANICKA@seznam.cz",
    "PEPA@GMAIL.COM",
]

def is_new_email(address):
    for user in users:
        if ...:
            return False
    return True
```

Chceme doplnit `...` tak, aby kód fungoval - tedy např. aby fungovali následující testy:

```
assert is_new_email("john@gmail.com")
assert is_new_email("example@gmail.com")

assert not is_new_email("JOHNdoe@gmail.com")
assert not is_new_email("JOHNDOE@GMAIL.COM")
assert not is_new_email("Janicka@Seznam.cz")
assert not is_new_email("pepa@gmail.com")
```

`str.lower / str.upper` není dobré řešení

Zřejmé řešení:

```
def is_new_email(address):
    for user in users:
        if user.lower() == address.lower():
            return False
    return True
```

Nemáme problém pro 26 písmen anglické abecedy:

```
>>> import string
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

Ale: Německé ostré s, **ß**:

```
>>> "ß".lower()
'ß'
>>> "ß".upper()
'SS'
>>> "SS".lower()
???
```

Existuje ještě několik dalších problematických znaků:

```
for i in range(65535):
    c = chr(i)
    if c.lower() == c and c.upper().lower() != c: print(i, c)

"""
Dostaneme znaky jako:
181 µ
223 ß
305 ı
329 ŉ
383 ſ
496 ǰ
837 ͅ
# a další.
"""
```

Samozřejmě můžeme omezit množinu znaků povolených v e-mailových adresách, např. na malá a velká písmena ze `string.ascii_lowercase`, ale ne vždy jde řešit problémy omezením možností.

Ze stejných důvodů nejde použít ani `str.upper()`:

```
for i in range(65535):
    c = chr(i)
    if c.upper() == c and c.lower().upper() != c: print(i, c)

"""
Dostaneme znaky jako:
304 İ
1012 ϴ
7838 ẞ
8486 Ω
8490 K
8491 Å
"""
```

`str.casefold()`

dělá pro běžná písmena to, co `str.lower()`, ale chová se regulárně i pro problematická písmena:

```
>>> sentence = "THE QUICK brown FoX jumps OVER the LaZy dog."
>>> sentence.lower()
'the quick brown fox jumps over the lazy dog.'
>>> sentence.casefold()
'the quick brown fox jumps over the lazy dog.'
---
>>> word = "straße"
>>> word.lower()
'straße'
>>> word.casefold()
'strasse'
```

Takže `str.casefold()` je správný nástroj pro porovnání řetězců necitlivé k velikosti písmen:

```
def addresses_match(new, old):
    return new.casefold() == old.casefold()

address_in_database = "Imaginary Straße, 27"
new_address = "IMAGINARY STRASSE, 27"

print(addresses_match(new_address, address_in_database))  # True
```

Slovník nerozlišující velikost písmen

Chtěli bychom něco takovéhoto:

```
class CaseInsensitiveDict(dict):
    ...

d = CaseInsensitiveDict()
print(d)  # {}

d["Rodrigo"] = "Rodrigo"
```

```
print(d["RODRIGO"])  # Rodrigo
del d["rOdRiGo"]
print(d)  # {}

d["straße"] = "street"
d["STRASSE"] = "STREET"
print(d)  # {'strasse': "STREET"}
```

Slovníky používají pro přístup ke klíčům, pro nastavování hodnot pro nový klíč, a pro odstraňování klíčů ze slovníku dunder metody .

- `__setitem__(key, value)` se volá, když chceme nastavit hodnotu pro klíč, například příkazem `d[key] = value`;
- `__getitem__(key)` se volá, když chceme ze slovníku získat hodnotu pro daný klíč, např. příkazem `print(d[key])`;
- `__delitem__(key)` se volá, když chceme ze slovníku odstranit ze slovníku klíč a odpovídajícíc hodnotu, např. příkazem `del d[key]`.

Takže potřebujeme přepsat tyto metody tak, aby se před použitím klíčů tyto normalizovali pomocí `str.casefold`. Abychom si ulehčili práci, odvodíme naši třídu `CaseInsensitiveDict` od základního slovníku `dict`. Tak nám zůstane starost, aby se pro normalizaci klíčů použila funkce `str.casefold`, ale jinak chování slovníku převezmeme z mateřské třídy `dict`.

```
class CaseInsensitiveDict(dict):
    """Case-insensitive dictionary implementation."""

    def __getitem__(self, key):
        return dict.__getitem__(self, key.casefold())

    def __setitem__(self, key, value):
        dict.__setitem__(self, key.casefold(), value)

    def __delitem__(self, key):
        dict.__delitem__(self, key.casefold())
```

Zavedli jsme malé úpravy slovníku, ale složitou funkcionalitu přebíráme od `dict`. Naše implementace pracuje takto:

```
d = CaseInsensitiveDict()
print(d)  # {}

d["Rodrigo"] = "Rodrigo"
print(d["RODRIGO"])  # Rodrigo
del d["rOdRiGo"]
print(d)  # {}

d["straße"] = "street"
d["STRASSE"] = "STREET"
print(d)  # {'strasse': "STREET"}
```

Množina, nerozlišující velikost písmen

Aby množina nerozlišovala velikost písmen v prvcích, musíme použít `str.casefold` při všech manipulacích s prvky množiny, např.

- při pridání nového prvku, `set.add`;
- při odstraňování prvků z množiny, `set.remove` / `set.discard` (jak se liší?);
- při kontrole, zda se prvek nachází v množině, `value in set`.

Pro ilustraci si vytvoříme třídu `CaseInsensitiveSet`, do kterého můžeme přidávat prvky, odebírat je a zjišťovat, zda se prvek nachází v množině bez ohledu na velikost písmen.

Chceme, aby třída dělala přibližně toto:

```
class CaseInsensitiveSet(set):
    # ...

s = CaseInsensitiveSet()

s.add("Rodrigo")
s.add("mathspp")
s.add("RODRIGO")

print(s)  # CaseInsensitiveSet({'rodrigo', 'mathspp'})
print("RODRIGO" in s)  # True

s.discard("MaThSpP")  # Try to remove "mathspp"
print(s)  # CaseInsensitiveSet({'rodrigo'})

s.discard("mathspp")  # Try to remove "mathspp"
print(s)  # CaseInsensitiveSet({'rodrigo'})

s.add("mathspp")
s.remove("rodrigo")  # Remove "rodrigo" and error if not present
print(s)  # CaseInsensitiveSet({'mathspp'})
```

Použijeme stejný postup jako u slovníku. Pro zjištění, zda se hodnota nachází v množině, používáme *dunder* funkci `__contains__`:

```
class CaseInsensitiveSet(set):
    def add(self, value):
        set.add(self, value.casefold())

    def discard(self, value):
        set.discard(self, value.casefold())

    def remove(self, value):
        set.remove(self, value.casefold())

    def __contains__(self, value):
        return set.__contains__(self, value.casefold())
```

Takováto implementace není úplná - potřebovali bychom upravit ještě několik dalších dunder metod, např. `set.__update__()`.

Třídění

Abychom mohli věci třídit, musí tyto věci implementovat operátory porovnání:

Nejprve si prosvištíme pár jednoduchých metod:

Selection sort

Zleva nahrazujeme hodnoty minimem zbývající části posloupnosti:

```
def selection_sort(b):
    for i in range(len(b) -1):
        j = b.index(min(b[i:]))
        b[i], b[j] = b[j], b[i]
    return b
```

$O(n^2)$ operací, $O(1)$ paměť.

Bubble sort

```
def bubble_sort(b):
    for i in range(len(b)):
        n_swaps = 0
        for j in range(len(b)-i-1):
            if b[j] > b[j+1]:
                b[j], b[j+1] = b[j+1], b[j]
                n_swaps += 1
        if n_swaps == 0:
            break
    return b
```

$O(n^2)$ operací, $O(1)$ paměť.

Insertion sort

Začínáme třídit z kraje seznamu, následující číslo vždy zařadíme na správné místo do již utříděné části.

```
def insertion_sort(b):
    for i in range(1, len(b)):
        up = b[i]
        j = i - 1
        while j >= 0 and b[j] > up:
            b[j + 1] = b[j]
            j -= 1
        b[j + 1] = up
    return b
```

$O(n^2)$ operací, $O(1)$ paměť.

Bucket sort

- Nahrubo si setřídíme čísla do příhrádek
- Setřídíme obsah přihrádek
- Spojíme do výsledného seznamu

```
def bucketSort(x):
    arr = []
    slot_num = 10 # 10 means 10 slots, each
                  # slot's size is 0.1
    for i in range(slot_num):
        arr.append([])

    # Put array elements in different buckets
    for j in x:
        index_b = int(slot_num * j)
        arr[index_b].append(j)

    # Sort individual buckets
    for i in range(slot_num):
        arr[i] = insertionSort(arr[i])

    # concatenate the result
    k = 0
    for i in range(slot_num):
        for j in range(len(arr[i])):
            x[k] = arr[i][j]
            k += 1
    return x
```

To je docela ošklivý kód, uměli bychom ho vylepšit?

Částečné třídění a quicksort

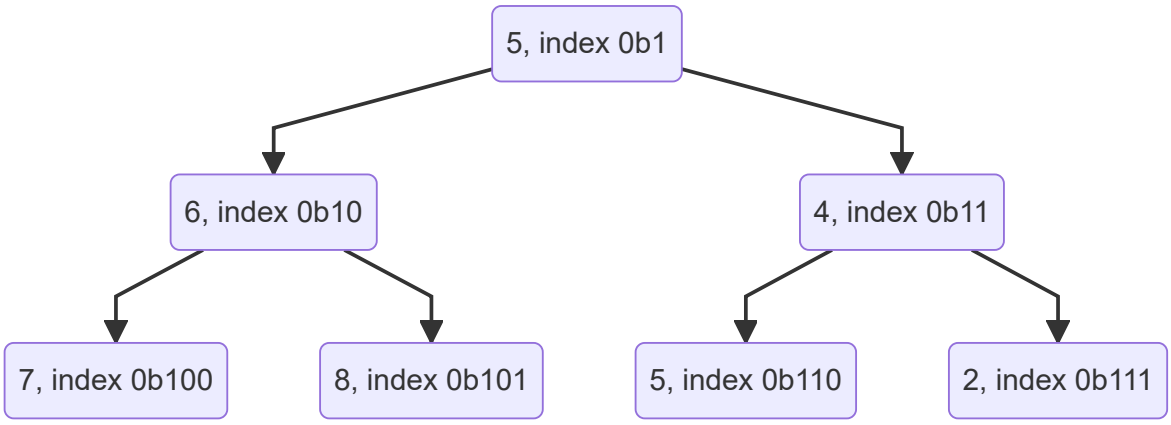
Ještě se na chvíli vrátíme k částečnému třídění pomocí pivotování:

```
from random import randint

data = [randint(1,10) for _ in range(10)]


def partial_sort(a: list[int], start: int = None, end: int = None) ->
tuple[list[int], int, int]:
    if start is None:
        start = 0
    if end is None:
        end = len(a)
    pivot = a[start]
    l_pivots = start
    r_pivots = start + 1
    p = r_pivots
    while p < end:
        if a[p] < pivot:
```

```
            hold = a[p]
            for i in range(p, l_pivots, -1):
                a[i] = a[i-1]
            a[l_pivots] = hold
            l_pivots += 1
            r_pivots += 1
        elif a[p] == pivot:
            hold = a[p]
            for i in range(p, r_pivots, -1):
                a[i] = a[i-1]
            a[r_pivots] = hold
            r_pivots += 1
        p += 1
    return a, l_pivots, r_pivots


print(data)
print(partial_sort(data))
```

Halda - heap

Binární strom, implementovaný v seznamu. Namísto struktury stromu používámee vztahy přes indexy:



- Potomci uzlu na indexu k jsou 2k a 2k+1
- Předek uzlu na indexu k je k // 2
- Uzel k je levý potomek svého předka, pokud k % 2 == 0, jinak je to pravý potomek.

Pravidlo min-haldy (min-heap): potomci uzlu jsou větší než hodnota v uzlu.

```
# heap implementation
from random import randint

def add(h:list[int], x:int) -> None:
    """Add x to the heap"""
    h.append(x)
    j = len(h)-1
```

```
    while j > 1 and h[j] < h[j//2]:
        h[j], h[j//2] = h[j//2], h[j]
        j //= 2


def pop_min(h: list[int]) -> int:
    """remove minimum element from the heap"""
    if len(h) == 1: # empty heap
        return None
    result = h[1]   # we have the value, but have to tidy up
    h[1] = h.pop()  # pop the last value and find a place for it
    j = 1
    while 2*j < len(h):
        n = 2 * j
        if n < len(h) - 1:
            if h[n + 1] < h[n]:
                n += 1
        if h[j] > h[n]:
            h[j], h[n] = h[n], h[j]
            j = n
        else:
            break
    return result


def main() -> None:
    heap = [None]  # no use for element 0
    for i in range(10):
        add(heap, randint(1, 100))
        print(heap)
    for i in range(len(heap)):
        print(pop_min(heap))
        print(heap)


if __name__ == '__main__':
    main()
```