`Programování` 1 pro matematiky

9. cvičení, 26-11-2024

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**
   - Vyskytlo se několik typických problémů, u kterých se zastavím podrobněji
   - Máme méně odevzdaných řešení

Kde se nacházíme

Dnes bude rychlé opakování a povíme si něco více o funkcích, příště začneme mluvit o třídách v Pythonu, čtení a zápisu do souborů a obsluze výjimek.

Co dělá tento kód?

```
functions = ["len", "min", "max", "sum"]

data = [1, 5, 3, 2, 4]
for fun_name in functions:
    fun = __builtins__.__dict__[fun_name]
    print(fun_name, fun(data))

---
len 5
min 1
max 5
sum 15
```

Jména všech objektů jsou v Pythonu uložena v tom nebo onom slovníku a můžeme tedy požadovaný objekt vybrat podle jména.

Domácí úkoly

Výběr vhodného kontejneru

Inverze slovníku

Toto bývá pociťováno jako těžší úloha, ale vám se vedlo dobře. ReCodEx u této úlohy často dává divné chyby nebo nedokáže rozeznat správné řešení, takže jsem několika z vás manuálně upravil hodnocení.

Základní řešení: Vytvořit seznam dvojic (nositel, vlastnost) a přeorganizovat jej.

Benfordův test

Toto byla aplikace na seznam nebo slovník. Seznam postačuje, protože máte jednoduchý a předem známý rozsah klíčů typu 1,2, ... n.

Vyhledávání v slovníku je rychlé, ale nic neporazí v rychlosti přímé indexování.

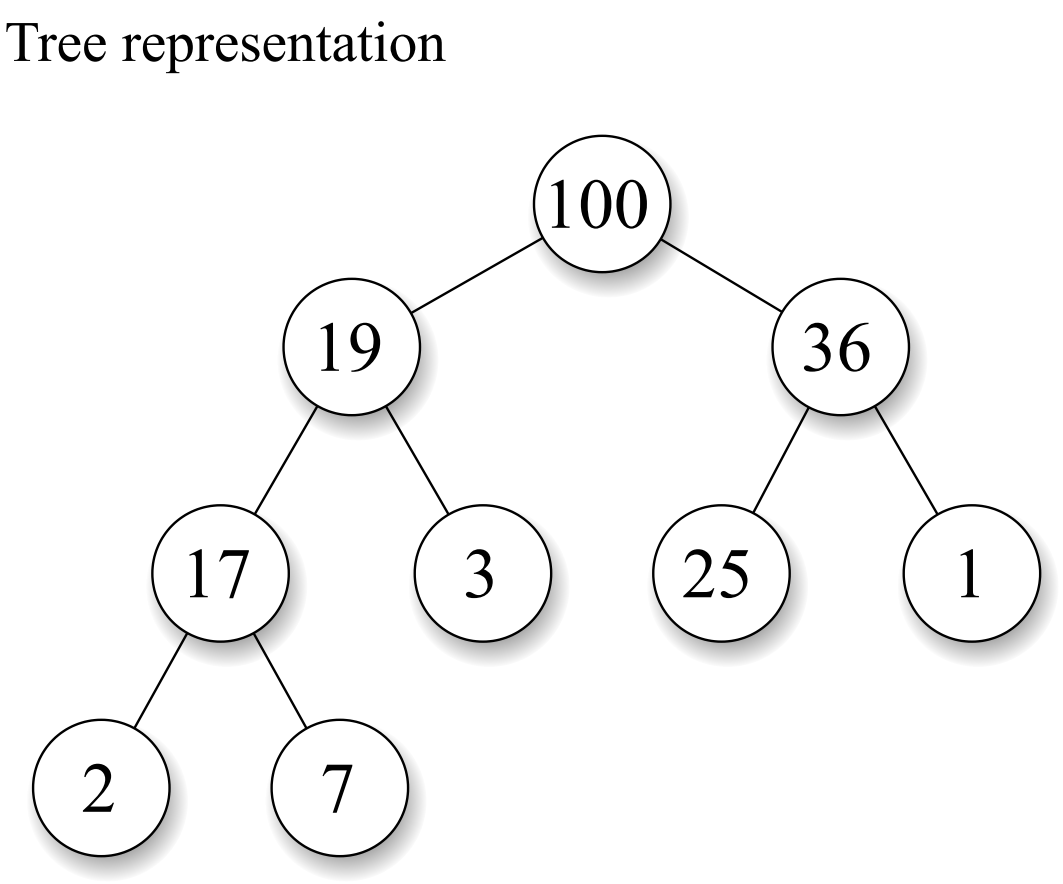
Pokud máte vytvořit přihrádky pro čísla 1..9 anebo 1..100, přidejte si nulový prvek. Seznam o jednu položku delší nikoho nezabije a používáte index totožné s hodnotou. U Benfordova testu se navíc přihrádka pro 0 docela hodí.

```
def benford_test():
    counts = [0] * 10  # Počítadlo pro číslice 0-9

    while True:
        # Načtení čísla ze vstupu
        line = input().strip()  # Nekonvertujeme na int
        if line == "-1":
            break

        # Zjištění první platné číslice
        first_digit = int(number[0])
        counts[first_digit] += 1  # Přímá aktualizace počítadla

    # Výstup výsledků: tiskneme seznam kromě položky 0.
    print(counts[1:])

# Spuštění programu
if __name__ == "__main__":
    benford_test()
```

Minimální/maximální součet

Základní řešení je setřídit seznam a vzít součet k nejmenších a k největších čísel. Složitost je $O(n \log n)$. Pro velice malé $k \leq \log n$ můžeme dostat lepší výsledek opakovaným výběrem minima a maxima. - ovšem za předpokladu, že nebudeme měnit tvar polí.

V příštím semestru se budeme učit o struktuře zvané *halda* (heap), pomocí které můžeme ušetřit nepotřebná srovnání a dosáhnout složitosti $O(n \log k)$.



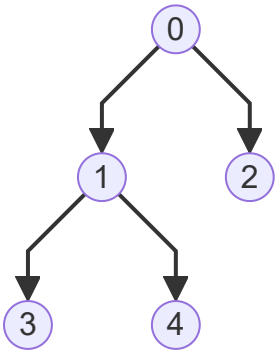
Halda - heap (By Kelott - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=99968794)

Různé

Vložka: Slovníky jako základ složitějších struktur

Slovníky můžou obsahovat jako hodnoty další slovníky, a tak můžeme vytvářet rozsáhlé hierarchické struktury, např. stromy:

```
strom = {"hodnota": 0, "deti" : [
    {"hodnota" : 1, "deti" : [
        {"hodnota": 3, "deti" : []},
        {"hodnota": 4, "deti" : []}
    ]},
    {"hodnota" : 2, "deti" : []}
    ]}
```



Takovýto zápis není zvlášť praktický pro lidského čtenáře, ale důležité je, že dokážeme složité struktury zapsat do textové formy a pracovat s nimi. Tento přístup je základem datoveho formátu JSON (JavaScript Object Notation).

Částečné setřídění pole

Úloha: Máme seznam celých čísel 0..n. Úloha je setřídit data *na místě* tak, aby byly všechny 0 na konci a zbylá data ve stejném pořadí na začátku seznamu. Tedy

```
[1,0,2,3,0,5,5,0,7] -> [1,2,3,5,5,7,0,0,0]
```

Řešení

Na místě znamená, že nebudeme vytvářet nový seznam a použijeme jenom fixní (nezávislou od velikosti vstupního seznamu) paměť pro potřebné proměnné.

Dva pointry:

- levý ukazuje po řadě na nuly v seznamu
- pravý ukazuje na následující nenulovou hodnotu

Projíždíme pole, vyměňujeme nulu u levého pointru za nenulovou hodnotu u pravého pointru. Končíme, když se pravý pointr zarazí o konec pole.

Neděláme nic fancy, žádné speciální kontejnery, žádný sort a pod., jenom jednoduchý přímočarý postup:

```
from random import shuffle # pro generování dat


def zeros_back(data: list[int]) -> list[int]:
    left = 0
    right = 0
    n = len(data)
    while left < n:
```

```
        if data[left] > 0: # hledáme nulu
            left += 1
            continue
        right = left
        while right < n and data[right] == 0: # hledáme následující nenulovou
hodnotu
            right += 1
        if right == n: # od left do konce seznamu jsou samé nuly, končíme.
            break
        data[left], data[right] = data[right], data[left] # Nalezli jsme,
vyměníme
        left += 1
    return data


def read_data() -> list[int]:
    data = []
    while (n := int(input())) != -1:
        data.append(n)
    return data


def generate_data() -> list[int]:
    data = [i for i in range(1, 10)] + [0] * 10
    shuffle(data)
    return data


def main() -> None:
    data = generate_data()
    print(data)
    print(zeros_back(data))


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

Pokračování: Funkce

Příklad

Napište funkci, která vrátí řešení rovnice $2^x + x = 11$.

Začneme tím, že zjevně $0 < x < 4$. a v tomto intervalu bude právě jedno řešení, protože funkce vlevo je rostoucí a vpravo konstantní. To, že řešení vidíte na první pohled, je dobré - máme kontrolu.

Snažíme se udělat obecnější řešení:

```
def fun(x):
    return 2**x + x - 11
```

```
def eqn_solve(f, l, p, eps = 1.0e-6):
    """We expect f(l) < 0 < f(p)"""
    if f(l) > 0 or f(p) < 0:
        print("l a p musí ohraničovat oblast, kde se nachází kořen. ")
        return None
    while abs(l-p) > eps:
        m = (l+p)/2
        if f(m)<0:
            l = m
        else:
            p = m
    return m

def main():
    print(eqn_solve(fun, 0, 4))

main()
```

Lambda-funkce

Kapesní funkce jsou bezejmenné funkce, které můžeme definovat na místě potřeby. Šetří práci například u funkcí jako `sort`, `min/max`, `map` a `filter`.

```
>>> seznam = [[0,10], [1,9], [2,8], [3,7], [4,6]]
>>> seznam.sort(key = lambda s: s[-1])
>>> seznam
[[4, 6], [3, 7], [2, 8], [1, 9], [0, 10]]
```

Pro přístup k prvkům kontejneru existuje v Pythonu speciální funkce:

```
>>> from operators import itemgetter

>>> seznam = [[0,10], [1,9], [2,8], [3,7], [4,6]]
>>> seznam.sort(key = itemgetter(1))
>>> seznam
[[4, 6], [3, 7], [2, 8], [1, 9], [0, 10]]
```

Lambda funkce s sebou nese určitou režii: Python zkonstruuje funktor, tedy třídu, která má vlastnosti funkce.

Rekurze: Permutace

Chceme vygenerovat všechny permutace množiny (rozlišitelných) prvků. Nejjednodušší je použít rekurzivní metodu.

```
def getPermutations(array):
    if len(array) == 1:   # Base case
        return [array]
    permutations = []
    for i in range(len(array)):
        # Aktuální prvek + všechny permutace pole bez něj
        perms = getPermutations(array[:i] + array[i+1:])
        for p in perms:
            permutations.append([array[i], *p])
    return permutations

print(getPermutations([1,2,3]))
```

Výhoda je, že dostáváme permutace setříděné podle původního pořadí.

Nevýhoda je, že dostáváme potenciálně obrovský seznam, který se nám musí vejít do paměti. Nešlo by to vyřešit tak, že bychom dopočítávali permutace po jedné podle potřeby?

Jiný příklad: Kombinace

Kombinace jsou něco jiné než permutace - permutace jsou pořadí, kombinace podmnožiny dané velikosti.

Začneme se standardní verzí, vracející seznam všech kombinací velikosti n. Všimněte si prosím odlišnosti oproti permutacím:

```
def combinations(a, n):
    result = []
    if n == 1:    # Base case: velikost 1 - vracíme seznam prvků
        for x in a:
            result.append([x])
        return result
    for i in range(len(a)):
        # Aktuální prvek + kombinace zbývajících o délce n-1
        for x in combinations(a[i+1:], n-1):
            result.append([a[i], *x])
    return result

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

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

Generátory a příkaz yield

Dáme-li list comprehension do kulatých závorek místo hranatých, dostaneme namísto seznamu generátor.

```
>>> r = (x for x in range(20) if x % 3 == 2)
>>> r
<generator object <genexpr> at 0x000001BC701E9BD0>
>>> for j in r:
...     print(j)
...
2
5
8
11
14
17
```

Následující ukázka demonstruje, jak Python interaguje s iterátorem:

```
>>> s = (x for x in range(3))
>>> next(s)
0
>>> next(s)
1
>>> next(s)
2
>>> next(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

`next(it)` vrací další hodnotu iterátoru, a pokud už další hodnota není, vyvolá iterátor výjimku `StopIteration`. To je standardní chování iterátoru. Co se skrývá pod kapotou? Toto:

Generátorem nazýváme funkci, která může fungovat jako iterátor - lze ji opakovaně volat, a ona pokaždé vrátí následující hodnotu z nějaké posloupnosti.

Příklad 1.

```
>>> def my_range(n):
        k = 0
        while k < n:
            yield k
            k += 1
        return

>>> list(my_range(5)

[0,1,2,3,4]
```

Příklad 2. Pokud vracíme hodnoty z posloupnosti, lze použít příkaz `yield from`:

```
>>> def my_range2(n):
        yield from range(n)

>>> list(my_range2(5)

[0,1,2,3,4]
```

Příklad 3 vám bude povědomý:

```
def read_list():
    while True:
        i = int(input())
        if i == -1:
            break
        yield i
    return

for i in read_list():
    print(f"Načetlo se číslo {i}.")

print("Konec cyklu 1")

for j in read_list():
    print(f"Teď se načetlo číslo {i}")

print("Konec cyklu 2")
```

Takto můžeme lehce oddělit cyklus zpracování dat od cyklu jejich načítání.

Generátory v Pythonu

Funkce pro počítání permutací a kombinací nám dávají potenciálně obrovské seznamy. Nešlo by je implementovat jako generátory?

```
def getPermutations(array):
    if len(array) == 1:   # Base case
        yield array
    for i in range(len(array)):
        # Aktuální prvek + všechny permutace pole bez něj
        for p in getPermutations(array[:i] + array[i+1:]):
            yield [array[i], *p]

for p in getPermutations([1,2,3]):
    print(p)
```

Podobně pro kombinace:

```python
def combinations(a, n):
    if n == 1:    # Base case: velikost 1 - vracíme seznam prvků
        for x in a:
            yield [x]
    for i in range(len(a)):
        # Aktuální prvek + kombinace zbývajících o délce n-1
        for x in combinations(a[i+1:], n-1):
            yield [a[i], *x]

for c in combinations([1,2,3,4,5],2):
    print(c)
```

Tyto funkce jsou implementovány v modulu `itertools`:

Combinatoric iterators:

Iterator	Arguments	Results
`product()`	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements

Examples	Results
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`	`AA AB AC AD BB BC BD CC CD DD`

Příklad: Erastothenovo síto

Jednoduchý generátor, generující čísla n, n+1, … můžeme implementovat například takto:

```python
def nums(n):
    yield n
    yield from nums(n+1)
```

Není to moc dobrá implementace, protože je rekurzivní, ale má základní vlastnosti, které potřebujeme:

- generuje potenciálně nekonečnou řadu čísel
- pamatuje si svůj vnitřní stav a na vyžádání nabídne další číslo.

Erastothenovo síto implementujeme tak, jak velí základní algoritmus: Začínáme s přirozenými čísly 2, 3, ..., a prosséváme je tak, že najdeme další číslo, které je prvočíslem, a pak odstraníme z posloupnosti všechny jeho násobky:

```
def nums(n):
    yield n
    yield from nums(n+1)


def sieve(s):
    n = next(s)
    yield n
    yield from sieve(i for i in s if i % n != 0)


s = sieve(nums(2))
for i in s:
    if i < 100:
        print(i)
    else: break
```

Toto není zajímavé pro reálné aplikace, kde nás zajímají obrovská prvočísla, protože to je hluboce rekurzivní implementace; ale je to dobrá ukázka toho, co lze udělat s generátory.

Domácí úkoly

- **Dobrá čísla**: máte vyhledat čísla s určitou vlastností v intervalu 1 .. n.
- **Najděte chybějící čísla**: V náhodné posloupnosti čísel z intervalu 1 .. 100 najít čísla, která v posloupnosti chybí.
- **k-ciferná čísla**: najít všechna k-ciferná čísla s daným ciferným součtem.