

Metoda `__new__()`:

Metoda `__new__()` je metoda, která je volána před `__init__()` při vytváření instance třídy.

Na rozdíl od `__init__()` metoda `__new__()` je statická metoda a je zodpovědná za vytvoření instance třídy. Používá se obvykle v případech, kdy je potřeba přizpůsobit proces vytváření instance, například když třída má imutabilní objekty a je potřeba je inicializovat specifickým způsobem nebo když je třeba vytvořit instanci jiného typu než je instance aktuální třídy. `__new__()` je také často používána v třídách, které jsou potomkem neměnných typů, jako je například `int`, `str`, atd. a umožňuje modifikovat chování vytváření instance.

Hešování v Pythonu:

V Pythonu je hashování důležitou součástí interního mechanismu ukládání a vyhledávání dat ve slovnících a mnoha jiných datových strukturách. Objekty, které mohou být použity jako klíče ve slovnících, musí být hashovatelné, což znamená, že musí implementovat metodu `__hash__()`. Metoda `__hash__()` je používána k vytvoření hashe (čísla) z daného objektu, který slouží jako index do hashovací tabulky. Tato metoda by měla vrátit stejný hash pro objekty, které jsou "rovnocenné" podle metody `__eq__()`. To znamená, že objekty, které jsou rovny podle definice `__eq__()`, by měly mít stejný hash.

1. Pomocí dopředného vyhledávání (lookahead) vytvořte takový regulární

výraz, který bude odpovídat (match) heslu s těmito omezeními:

- bude obsahovat alespoň jedno malé písmeno,
- bude obsahovat alespoň jednu číslici,
- bude obsahovat alespoň 3 velká písmena,
- nebude obsahovat slova Jan nebo Vik,
- bude dlouhé 4-16 znaků.

Regulární výraz:

```
^(?=.*[a-z])(?=.*\d)(?=.*[A-Z]{3})(?!.*(?:Jan|Vik)).{4,16}$
```

Vysvětlení:

`^ a $` - zajišťuje, že celý vstup musí odpovídat regulárnímu výrazu (začátek a konec řetězce).

`(?=.*[a-z])` - dopředné vyhledávání (lookahead), které zajišťuje, že v řetězci musí být alespoň jedno malé písmeno.

`(?=.*\d)` - dopředné vyhledávání, které zajišťuje, že v řetězci musí být alespoň jedna číslice.

`(?=.*[A-Z]{3,})` - dopředné vyhledávání, které zajišťuje, že v řetězci musí být alespoň tři velká písmena.

`(?!.*(?:Jan|Vlk))` - negativní dopředné vyhledávání, které zajišťuje, že v řetězci nesmí být slova "Jan" nebo "Vlk".

`{4,16}` - zajišťuje, že řetězec musí být dlouhý 4 až 16 znaků, kde `.` znamená libovolný znak.

2. Co nejčastěji/nepravděpodobněji vypíše následující kód v Pythonu a proč?

Kód:

```
import random

class LoMe(object):

    def hash (self):

        return random. randint (1, 10000)

a = LoMe()

dict = {a: 42}

lst = list(dict)

print(a in dict, a in lst)
```

Výstup:

Tento kód by měl pravděpodobně vypsát `True False`.

Důvodem je to, že ve vytvořeném slovníku `dict` je použita instance třídy `LoMe`

jako klíč. Python implicitně používá hashovací funkci pro klíče ve slovníku.

Nicméně, když si uživatel explicitně definuje metodu `__hash__()` pro vlastní třídu,

jako je toto, Python použije tuto metodu namísto výchozího chování. Ve vašem kódu

je definována metoda `hash()`, která vrací náhodné číslo mezi 1 a 10000, ale

nedefinuje metodu `__hash__()`.

Tím pádem je implicitní hashovací funkce pro tuto instanci třídy v podstatě vrací None, což způsobuje, že Python použije `id(instance)` jako hash.

Když se vytvoří seznam `lst` pomocí klíčů slovníku `dict`, Python užívá identifikátory objektů jako klíče pro tento seznam. Takže instance `a` bude ve slovníku, ale ne v seznamu, což způsobí výsledek `True False`.

3. Definujte funkci `in_both`, která bude mít dva parametry, první z nich bude brát řetězec s hodnotami oddělenými středníkem a druhý bude brát řetězec s hodnotami oddělenými zavináčkem. Návrátovou hodnotou funkce bude seznam s těmi hodnotami, které se vyskytují v obou řetězcích.

Příklad:

Kód:

```
strA = "A;A;B;C;D;D;E"
strB = "BoC@C@C@E"
res = in_both(strA, strB) //v res bude ['B', 'C', 'E']
```

Riesenie:

```
def in_both(strA, strB):
    setA = set(strA.split(';'))
    setB = set(strB.split('@'))
    return list(setA.intersection(setB))
```

Tato funkce rozdělí oba vstupní řetězce podle jejich oddělovačů (; a @), vytvoří množiny z jednotlivých řetězců a poté vrátí seznam prvků, které jsou v obou množinách (tj. průnik množin).

1.1. Napsat regex, který nahradí název souboru basename a koncovku za extension, ale

název souboru nesměl končit .tar nebo .gz

test.soubor.pdf -> basename.extension

test.tar -> test.tar

Regulární výraz:

```
^(?!.*(?:\.tar|\.gz)$)(.*)\..*$
```

Vysvětlení:

^ a \$ zajišťují, že začátek a konec řetězce musí odpovídat celému výrazu.

(?!.*(?:\.tar|\.gz)\$) je negativní výrazy dopředného vyhledávání (negative lookahead), který zajišťuje, že řetězec nesmí končit .tar nebo .gz.

(.*) zachytí všechny znaky před poslední tečkou, která odděluje jméno souboru od přípony.

\. odpovídá tečce, která odděluje basename od přípony.

.* odpovídá jakémukoli počtu znaků přípony.

Tento regulární výraz bude platný pouze pro soubory, které mají název bez přípony .tar nebo .gz. Pokud tato podmínka není splněna, celý název souboru s příponou zůstane beze změny.

2.2. Napsat v pythonu nebo v ruby kód, který vygeneruje seznam kombinací 3 čísel z

množiny 0 až n-1. Žádný číslo se nesmí rovnat jinému (x != y and x != z and y != z)

```
def generate_combinations(n):
```

```
    combinations = []
```

```
    for i in range(n):
```

```
        for j in range(i + 1, n):
```

```
            for k in range(j + 1, n):
```

```
                combinations.append((i, j, k))
```

```
    return combinations
```