Skriptovací jazyky - Cvičení 3

Adam Albert

Co jsou iterátory?

- Iterátor v Pythonu je objekt, který umožňuje programátorům projít (iterovat) přes všechny prvky kolekce, jako jsou seznamy, n-tice, slovníky, množiny atd.
- V Pythonu iterátor implementuje metody `__iter__()` a `__next__()` (jsou to tzv. magiceké metody).
- Iterátory v Pythonu jsou široce používány, protože umožňují čistý a efektivní způsob, jak projít prvky kolekce, aniž by bylo nutné vědět předem, kolik prvků kolekce obsahuje.
- Python také poskytuje vestavěnou funkci iter(), která bere iterovatelný objekt (např. seznam) a vrací iterátor pro tento objekt. Podobně funkce next() je používána k získání dalšího prvku iterátoru.

Protokol Iterátoru

- Metoda `__iter__()` vrací iterátor objektu.
- Metoda `__next__()` vrací další prvek v sekvenci.
- Když nejsou další prvky, vyvolá se výjimka `StopIteration`.

Iterace přes seznam

```
# Definujeme seznam
my_list = [1, 2, 3, 4]

# Získáme iterátor pomocí funkce iter()
my_iter = iter(my_list)

# Projdeme prvky pomocí funkce next()
print(next(my_iter))  # Výstup: 1
print(next(my_iter))  # Výstup: 2
# a tak dále, dokud nevyvolá StopIteration po projití všech prvků
```

Generátory

- Generátory v Pythonu jsou jednoduchý způsob pro implementaci iterátorů.
- Jsou psány jako běžné funkce, ale používají výraz `yield` kdykoli chtějí vrátit data.
- Během toho, když generátor produkuje data, je možné udržet stav funkce, což umožňuje efektivní a jednoduché zpracování sekvencí dat, aniž by bylo nutné uchovávat celou sekvenci v paměti.

Generátory - výhody

- **Efektivita paměti**: Generátory ukládají pouze svůj aktuální stav (a ne celý seznam), což znamená, že spotřeba paměti zůstává nízká, i když pracují s velkými datovými sadami.
- **Lazy vyhodnocování**: Generátory vyprodukují ”další” hodnotu pouze na vyžádání, což znamená, že výpočet další hodnoty se neuskuteční, dokud není skutečně potřeba. To může zlepšit výkon programu, zvláště když nejsou všechny vygenerované hodnoty skutečně použity.
- **Jednoduchost použití:** Generátory mohou zjednodušit kód potřebný pro iteraci, protože se o správu iterátorů stará Python.

Jak Generátory Fungují

- Generátor vytvoříte definováním funkce, která místo *return* používá *yield*.
- Kdykoli Python narazí na výraz yield, vrátí hodnotu ”vedle” *yield* a pozastaví vykonávání funkce.
- Funkce může být později pokračována od místa, kde byla pozastavena.

```
def simple_generator():
    yield 1
    yield 2
    yield 3

# Vytvoření generátoru
gen = simple_generator()

# Iterace přes generátor
for value in gen:
    print(value)
```

Rozumnější příklad

```
students_db = [
    {"name": "Alice", "age": 22, "grade_avg": 1.8},
    {"name": "Bob", "age": 23, "grade_avg": 2.4},
    {"name": "Charlie", "age": 20, "grade_avg": 1.5},
    {"name": "Diana", "age": 21, "grade_avg": 2.1}
]

def student_data_generator(student_db):
    for student in student_db:
        yield student
```

Pomocí generátoru `student_data_generator` můžeme jednoduše iterovat přes všechny záznamy v naší ”databázi” bez nutnosti načítat všechny záznamy do paměti najednou:

```
for student in student_data_generator(students_db):
    print(f"Student: {student['name']}, Age: {student['age']},
    Average Grade: {student['grade_avg']}")
```

Rozumnější příklad

- Rozšířený příklad s přidaným filtrováním studentů dle jejich průměrné známky.

```
def student_data_generator_filtered(student_db, min_grade_avg):
    for student in student_db:
        if student["grade_avg"] <= min_grade_avg:
            yield student

for student in student_data_generator_filtered(students_db, 2.0):
    print(f"Student: {student['name']} has an average grade below
    2.0")
```

Generátor načítání ze souboru

- Generátory v Pythonu jsou velmi užitečné pro efektivní práci se soubory, zvláště pokud se jedná o velké soubory, které by bylo neefektivní nebo dokonce nemožné načítat celé do paměti najednou.
- Následující příklad ukazuje, jak vytvořit generátor pro postupné čtení řádků ze souboru, což minimalizuje spotřebu paměti:

```
def cti_radky(soubor):
    """Generátorová funkce pro postupné čtení řádků ze souboru."""
    with open(soubor, 'r', encoding='utf-8') as s:
        for radek in s:
            yield radek.strip()  # Odstraní bílé znaky

# Použití generátoru pro čtení ze souboru
cesta_k_souboru = 'cesta/k/vasemu/souboru.txt'
for radek in cti_radky(cesta_k_souboru):
    print(radek)
```

OOP v Pythonu

- Třída v Pythonu je vytvořena pomocí klíčového slova *class* a slouží jako šablona pro vytváření objektů. Třída definuje stav a chování objektů, které jsou jejími instancemi.
- Definice třídy:

```
class MyClass:
    def __init__(self, value):  # Konstruktor
        self.attribute = value

    def method(self):  # Metoda třídy
        return self.attribute * 2
```

- Vytváření instancí

```
obj = MyClass(10)
```

Instanční vs. třídní proměnné

- Nejběžnější místo, kde se setkáte s atributy, je v metodě `__init__`, což je konstruktor třídy. Atributy zde definované jsou obvykle instanční atributy, což znamená, že každá instance třídy má vlastní kopii těchto atributů.

```
class Auto:
    def __init__(self, znacka, model):
        self.znacka = znacka  # Instanční atribut
        self.model = model    # Instanční atribut
```

- Třídní atributy jsou definovány přímo v těle třídy a sdíleny mezi všemi instancemi této třídy.

```
class Auto:
    pocet_aut = 0  # Třídní atribut

    def __init__(self, znacka, model):
        self.znacka = znacka
        self.model = model
        Auto.pocet_aut += 1
```

Dědičnost

- Dědičnost umožňuje nové třídě převzít atributy a metody jiné třídy. Třída, ze které se dědí, se nazývá nadřazená třída (nebo rodičovská třída), zatímco třída, která dědí, se nazývá podřazená třída (nebo dítě).
- Příklad dědění:

```
class Parent:
    def __init__(self):
        self.value = "Parent"

    def show(self):
        print(self.value)

class Child(Parent):  # Dědění z Parent
    def __init__(self):
        super().__init__()  # Volání konstruktoru nadřazené třídy
        self.value = "Child"  # Přepsání atributu
```

Dědičnost - rozšířený příklad

- Mějme definici základní třídy *Zaměstnanec*, která bude sloužit jako nadřazená třída pro všechny specifické role.

```
class Zaměstnanec:
    def __init__(self, jmeno, plat):
        self.jmeno = jmeno
        self.plat = plat

    def vypis_info(self):
        return f"Zaměstnanec: {self.jmeno}, Plat: {self.plat}"

    def pracuj(self):
        return "Zaměstnanec pracuje."
```

Dědičnost - rozšířený příklad

- Dvě specializované role, *Manažer* a *Programátor*, které dědí z Zaměstnanec. Každá z těchto rolí bude mít specifické atributy a metody.
- Funkce *super()* v Pythonu se používá v kontextu dědičnosti a slouží k volání metod nadřazené (rodičovské) třídy z potomkové (dětské) třídy. Tímto způsobem umožňuje přístup k metodám nadřazené třídy, které byly přepsány v dětské třídě, nebo umožňuje inicializovat atributy nadřazené třídy z konstruktoru potomka.
- Definice Manažera:

```
class Manažer(Zaměstnanec):
    def __init__(self, jmeno, plat, oddělení):
        super().__init__(jmeno, plat)
        self.oddělení = oddělení

    def vypis_info(self):
        return f"{super().vypis_info()}, Oddělení: {self.oddělení}"

    def pracuj(self):
        return "Manažer řídí své zaměstnance."
```

Dědičnost - rozšířený příklad

- Definice Programátora:

```
class Programátor(Zaměstnanec):
    def __init__(self, jmeno, plat, programovací_jazyk):
        super().__init__(jmeno, plat)
        self.programovací_jazyk = programovací_jazyk

    def vypis_info(self):
        return f"{super().vypis_info()}, Programovací jazyk:
        {self.programovací_jazyk}"

    def pracuj(self):
        return f"Programátor píše kód v {self.programovací_jazyk}."
```

Dědičnost - rozšířený příklad

- Předpokládejme, že chceme vytvořit speciální typ programátora, SeniorProgramátor, který má navíc metodu pro mentoring junior programátorů.

```
class SeniorProgramátor(Programátor):
    def __init__(self, jmeno, plat, programovací_jazyk, juniorů):
        super().__init__(jmeno, plat, programovací_jazyk)
        self.juniors = juniors

    def mentoruj(self):
        return f"Senior programátor mentoruje {self.juniors} juniorů."

    def pracuj(self):
        # Přetížení metody pracuj z Programátor
        return f"{super().pracuj()} a mentoruje junior programátory."
```

Dědičnost - rozšířený příklad

- Použití:

```
zaměstnanec = Zaměstnanec("Jan Novák", 30000)
manažer = Manažer("Petr Sýkora", 50000, "IT")
programátor = Programátor("Lucie Králová", 40000, "Python")
senior = SeniorProgramátor("Martin Zelený", 60000, "Python", 3)

print(manažer.vypis_info())
print(programátor.pracuj())
print(senior.mentoruj())
```

Třídy a properties

- Properties tříd (vlastnosti) se vytváří pomocí tzv. dekrátorů, což umožňuje přistupovat k atributům třídy a manipulovat s nimi jako s jednoduchými atributy, ale za použití metod, které definují logiku za jejich získáváním (getter) a nastavováním (setter).
- Tento přístup pomáhá zachovat zapouzdření dat tím, že skrývá implementační detaily a poskytuje veřejné rozhraní pro práci s daty třídy.
- Když použijete *@property* na metodu třídy, tato metoda se stává ”getterem” pro název vlastnosti, který odpovídá názvu metody. To umožňuje číst hodnotu atributu bez přímé manipulace s interními daty třídy.

```
class Osoba:
    def __init__(self, jmeno):
        self._jmeno = jmeno # '_' před 'jmeno' je konvence v Pythonu
                            # označující chránený atribut

    @property
    def jmeno(self):
        return self._jmeno
```

Třídy a properties

- Chcete-li umožnit nastavování hodnoty property, použijte *@property_name.setter* dekorátor na další metodu s tím samým názvem. Tato metoda pak slouží k nastavování hodnoty atributu.

```
class Osoba:
    def __init__(self, jmeno):
        self._jmeno = jmeno

    @property
    def jmeno(self):
        return self._jmeno

    @jmeno.setter
    def jmeno(self, hodnota):
        if isinstance(hodnota, str) and len(hodnota) > 0:
            self._jmeno = hodnota
        else:
            raise ValueError("Jméno musí být řetězec s
            délkou větší než 0.")
```

Zadání pro procvičení

- Vytvořte třídu *`Kniha`*, která bude reprezentovat knihu v knihovně. Každá instance třídy Kniha by měla obsahovat následující atributy:
  - Název knihy (řetězec)
  - Autor knihy (řetězec)
  - Rok vydání (celé číslo)
  - Počet stran (celé číslo)
  - ISBN kód (řetězec)
- Dále vytvořte následující metody uvnitř třídy *`Kniha`*:
  - Konstruktor *`__init__`*, který inicializuje všechny výše uvedené atributy.
  - Metodu *`__str__`*, která vrátí řetězec s krátkým popisem knihy (např. ”Název knihy: XYZ, Autor: ABC”).
  - Metodu *`vek_knihy`*, která vrací věk knihy v letech od jejího vydání do současného roku.

Zadání pro procvičení

- Následně vytvořte třídu Knihovna, která bude spravovat kolekci knih. Tato třída by měla obsahovat:
  - Atribut *`kniha`*, který bude seznamem instancí třídy Kniha.
  - Metodu *`pridej_knihu`*, která přijme instanci Kniha a přidá ji do seznamu knih.
  - Metodu *`vypis_knihy`*, která vypíše informace o všech knihách v knihovně.
  - Přidejte metodu *`najdi_knihu`*, která bude hledat knihy podle názvu nebo autora a vracet seznam knih odpovídajících dotazu.
  - Rozšiřte třídu Knihovna o metodu *`odstran_knihu`*, která umožní odstranit knihu ze seznamu na základě ISBN kódu.
- Testujte vaše třídy vytvořením několika instancí třídy *`Kniha`* a přidáním těchto instancí do instance třídy *`Knihovna`*. Zkuste vypsat seznam knih v knihovně, najít knihu podle názvu nebo autora a odstranit knihu ze seznamu.

Další zdroje pro samostudium

- Důrazně doporučuji si prostudovat skripta zde: http://mrl.cs.vsb.cz/people/gaura/skj/skripta.pdf
- Projděte si také kurz od Pyladies zde: https://naucse.python.cz/course/pyladies/beginners/class/