Laboratorium 10 — Ćwiczenia w programowaniu obiektowym. Tworzenie własnych kolekcji.

Emulowanie funkcji

Dzięki przeciążeniu metody `__call__` możemy sprawić, że obiekt klasy będzie zachowywał się jak funkcja z dodatkowymi możliwościami np. zapamiętywaniem stanów pośrednich.

```
class Iterate:
     def __init__(self, a = 0):
         self.a = a

     def __call__(self):
         self.a += 1
         return self.a

     # Jesli chcemy aby przyjmowala parametry
     # def __call__(self, *arguments, **keywords):
     # (1, 2) --- arguemnts
     # (1, 2, x = 3, y = 4) --- arguments + keywords


i = Iterate()
print(i())
print(i())
```

Zadanie 1 Napisz klasę emulująca funkcję potęgowania, przy czym potęga powinna startować od zadanej w konstruktorze i rosnąć z każdym wywołaniem o 1. Wykorzystaj powstałą klasę w pętli do wygenerowania kolejnych potęg liczby 2.

Iterator

Python pozwala na zbudowanie klasy iteratora poprzez rozszerzenie metod `__iter__` oraz `__next__`:

```
class Iterate:
     def __init__(self, a, b):
         self.a = a
         self.b = b

     def __iter__(self):
         return self

     def __next__(self):
         if self.a > self.b:
             raise StopIteration
         else:
             self.a += 1
         return self.a - 1

for i in Iterate(3, 8):
    print(i)
```

Zamiast klasy można też napisać odpowiednią funkcje:

```
def IterateFun(a, b):
    a = a
    while a <= b:
        yield a
        a += 1
for i in IterateFun(3, 8):
    print(i)
```

Zadanie 2 Zbuduj nową klasę generującą kolejne potęgi zadanej liczby bazującej na trzech wartościach: minimalnej potędze, maksymalnej potędze oraz zadanej liczbie. Wykorzystaj powstałą klasę w pętli `for`, w sposób w jaki używana jest funkcja `range()`.

Emulowanie kolekcji

Python zawiera szereg funkcji pozwalających na emulację kolekcji za pomocą klas. Wśród nich są:

- `__len__(self)` — powinna zwracać liczbę elementów w kolekcji;
- `__iter__(self)` — powinna zwracać iterator pozwalający przejść po całej kolekcji;
- `__reversed__(self)` — podobnie jak powyżej, ale powinna zacząć od końca kolejki i kierować się ku początkowi;
- `__getitem__(self, key)` — odpowiada ze operator `[key]` oraz `[min:max]`;
- `__setitem__(self, key, value)` — odpowiada za przypisanie wartości pod danym kluczem, lub dla danego zakresu;
- `__delitem__(self, key)` — usuwanie wartości z kolekcji (`del X[key]`, `del X[min:max]`);
- `__contains__(self, item)` — sprawdzenie czy elementy występuje w kolekcji;
- `__copy__(self)` — płytka kopia.

Przekazanie zakresu (początek:koniec:krok) do metod `__getitem__`, `__setitem__` lub `__delitem__` spowoduje utworzenie obiektu klasy `slice`. Np.:

```
class Test:
    def __getitem__(self, ind):
        print(ind)
        print(ind.start)
        print(ind.stop)
        print(ind.step)

test = Test()
test[1:5:2]
```

da nam:

```
slice(1, 5, 2)
1
5
2
```

Na potrzeby słowników istnieje także klasa abstrakcyjna `MutableMapping` ułatwiająca ich implementacje.

Zadanie 3 Twoim zadaniem będzie napisanie klasy emulującej sortującą się listę dwukierunkową. Klasa ta powinna zawierać implementację metod `__iter__(self)`, `__reversed__(self)` oraz `__contains__(self, item)`, metodę `insert` pozwalająca na wstawienie nowej wartości (tak, aby nie zaburzyć porządku w liście). W przypadku listy dwukierunkowej każdy element powinien zawierać referencję na obiekt następny oraz poprzedni lub `null`, jeśli takiego nie ma oraz wartość. Nowe elementy muszą być wstawiane w taki sposób aby zawartość listy była posortowana (pamiętaj, aby po wstawieniu nowego elementu zaktualizować adresy w aktualnym elemencie, poprzedniku oraz następniku). Implementacja powinna składać się z klasy `DList` oraz zagnieżdżonej klasy `Element`. Iterację po liście można wykonać przy użycia słowa kluczowego `yield` jak w kodzie poniżej (bez implementacji nowej klasy iteratora).

Kod do uzupełnienia:

```
class DList:
    class Element:
        def __init__(self, value, next = None, prev = None):
            self.__value = value
            self.next = None
            self.prev = None

        @property
        def value(self):
            return self.__value

    def __init__(self, args):
        self.__root = None
        self.__end = None
        for i in args:
            self.insert(i)

    def insert(self, value):
        # - przypadek, gdy lista jest pusta
        # - przypadek, gdy trzeba przeiterowac szukajac
        # wiekszego elementu i wstawic przed nim
        # - przypadek, gdy wstawiamy na koncu
        pass

    def __iter__(self):
        curr = self.__root
        while not (curr is None):
            yield curr.value - zwraca generator
            curr = curr.next

    def __reversed__(self):
        pass

    def __contains__(self, value):
        # przebiega po liscie az trafi
        # na rowny lub wiekszy elemnet
        # jesli wiekszy lub koniec listy
        # zwraca false
        pass

#----------------- TESTY -----------------
dlista = DList([2, 3, 4, 5])
for i in dlista:
    print(i)
print('\n--------\n')

dlista = DList([8, 10, 9, 13])
dlista.insert(7)
dlista.insert(12)
dlista.insert(15)
for i in reversed(dlista):
    print(i)

print('\n--------\n')
print('Contain 8:', 8 in dlista)
print('Contain 13:', 13 in dlista)
print('Contain 15:', 15 in dlista)
print('Contain 6:', 6 in dlista)
print('Contain 11:', 11 in dlista)
print('Contain 17:', 17 in dlista)
```