Lab 1: Wstęp teoretyczny do metod Pythona

W Pythonie wszystkie operatory mogą być przeciążane (tzn. implementowane) przez klasy Pythona oraz typy z rozszerzeń języka C, tak by działały na tworzonych obiektach. Python sam automatycznie przeciąża niektóre operatory, tak by wykonywały one różne zadania w zależności od typu przetwarzanego obiektu wbudowanego.

1.1 MODUŁ RANDOM

Biblioteka podstawowa Pythona zawiera moduł random, który jest przeznaczony do generowania liczb losowych dla różnych rozkładów prawdopodobieństwa (random.py). De facto są to liczby pseudolosowe, ponieważ sekwencja wygenerowanych w ten sposób liczb zależy od ziarna (ang. seed). Rozkład liczb pseudolosowych ma pewne ukryte regularności, które nie pojawiają się przy generacji liczb prawdziwie losowych. Generator inicjowany ziarnem, które może przyjąć *k* różnych wartości, jest w stanie wyprodukować co najwyżej *k* różnych ciągów liczb.

Dla niezmiennej wartości ziarna, generator random będzie generował niezmienną sekwencję wartości. A zatem jeśli ziarnu przypiszemy wartość 2, zawsze otrzymamy taką samą sekwencję:

```
import random
random.seed(2)
print(random.random())
print(random.random())
print(random.random())
```

Dane wyjściowe zawsze będą zgodne z następującą sekwencją:

0.9560342718892494
0.9478274870593494
0.05655136772680869

Prawie wszystkie funkcje modułu random korzystają z podstawowej funkcji random(), która generuje zmiennoprzecinkowe liczby losowe z lewostronnie domkniętego przedziału wartości [0.0, 1.0). Moduł random korzysta z opracowanego w 1997 roku przez Matsumoto i Nishimura algorytmu Mersenne Twister [1]. Nazwa algorytmu pochodzi od faktu, że na długość okresu wybierana jest liczba pierwsza Mersenne'a. Algorytm zapisany jest w pliku mt19937ar.c, który przechowuje funkcję genrand_res53(), przy pomocy której generator jest w stanie wygenerować 53-bitowe liczy losowe o rozkładzie równomiernym (wykorzystując do tego dwie 32-bitowe liczby całkowite). Okres generatora wynosi $2^{19937} - 1$. Istnieje jeszcze inna implementacja wspomnianego algorytmu – mt19937-64, która używa 64-bitowej długości słowa do generowania liczb pseudolosowych.

Funkcje dostarczane przez moduł random stanowią metody dowiązane ukrytego egzemplarza klasy random.Random. random.Random() tworzy generator liczb pseudolosowych, czyli obiekt, który generuje ciąg liczb, które wydają się losowe (są pseudolosowe). Wspomniany obiekt (ziarno) może być ciągiem znakowym (string), liczbą całkowitą (integer) lub zmiennoprzecinkową (float). Obiekt ten jest następnie wykorzystywany do stworzenia obiektu, który generuje określoną sekwencję liczb pseudolosowych. Przykładowo można wywołać:

- random.Random(57),
- random.Random(888.6),
- random.Random("Hello World"), lub
- random.Random(99898989),

aby uzyskać generator określonego ciągu liczb pseudolosowych. Określenie ziarna jest jednak konieczne tylko wtedy, gdy pożądane jest uzyskanie powtarzalnej „losowości” w tworzonym programie.

Następnie można użyć tak powstałego generatora do wyodrębnienia pseudolosowych liczb z tej sekwencji:

```
#Stwórz generator z pominięciem ziarna tak,
#aby pseudolosowe sekwencje wygenerowane za każdym razem
#gdy program jest uruchamiany się między sobą różniły

#Zaimportuj moduł random
import random
randomGen = random.Random()
# Wygeneruj liczbę w przedziale [0, 1)
number = randomGen.random()
print(number)
# Wygeneruj liczbę całkowitą w przedziale [0, 5]
number = randomGen.randint(0, 5)
print(number)
```

Należy zauważyć, że w powyższym przykładzie następuje przypisanie generatora random.Random do zmiennej o nazwie randomGen – samo stworzenie generatora nie jest do niczego przydatne.

Można tworzyć własne egzemplarze klasy Random w celu uzyskania generatorów, które nie współdzielą stanu. Jest to szczególnie przydatne w przypadku programów wielowątkowych, tworzących oddzielną instancję Random dla każdego wątku i używając metody jumpahead(), aby upewnić się, że wygenerowane sekwencje widziane przez każdy wątek nie nakładają się.

Klasa Random może być również klasą bazową dla innych klas, które będą korzystać z innego generatora liczb. W takim przypadku należy zastąpić metodę random() oraz funkcje rejestrujące stan generatora, tj. seed(), getstate(), setstate() oraz jumpahead().

Jako przykład tego typu dziedziczenia moduł random udostępnia klasę WichmannHill, która implementuje alternatywny generator liczb przygotowany w czystym Pythonie [2]. Należy jednak zaznaczyć, że generator ten nie spełnia współczesnych standardów, ze względu na zbyt krótki okres oraz niewystarczającą losowość generowanych sekwencji.

Moduł random udostępnia również klasę SystemRandom, która wykorzystuje funkcję systemową os.urandom() do generowania liczb losowych ze źródeł dostarczanych przez system operacyjny Unix (/dev/urandom) oraz CryptGenRandom dla systemu Windows. SystemRandom nie opera się na stanie oprogramowania, a sekwencje nie są odtwarzalne. W związku z tym metody seed() i jumpahead() nie działają i są ignorowane. Metody getstate() i setstate() wywołują NoImplementedError, jeśli są wywołane.

UWAGA #1: Ze względu na fakt, że Mersenne Twister jest generatorem deterministycznym, nie nadaje się on do zastosowań kryptograficznych. Do tego wykorzystywany jest moduł secrets z biblioteki standardowej Pythona.

UWAGA #2: Pod adresem [3] znaleźć można implementację algorytmu Mersenne Twister przygotowaną w pełni w języku Python. Autorem implementacji jest Guido van Rosum.

1.2 METODY SPECJALNE (DUNDER)

Specjalne metody są oznaczone podwójnym podkreśleniem po obu stronach ich nazwy (np. __init__). Alternatywnie metody te są określane mianem dunder (od Double-underscores) lub magicznych.

Python używa specjalnych metod w celu zwiększenia funkcjonalności klas. Większość z nich działa w tle i jest wywoływana automatycznie, gdy program tego potrzebuje. Nie można ich wprost wywołać. Dla przykładu, gdy tworzony jest nowy obiekt, Python automatycznie wywołuje metodę __new__, która z kolei wywołuje metodę __init__.

Metoda __init__

Metoda specjalna __init__ to konstruktor Pythona. Konstruktor to specjalna metoda, którą program wywołuje przy tworzeniu obiektu. Konstruktor jest używany w klasie do inicjowania składowych danych do obiektu. Np. mając daną klasę **myśliwiec**, można użyć konstruktora do przypisania cech myśliwca do każdego obiektu MiG-29.

Metoda __init__ jest w Pythonie odpowiednikiem konstruktora C++ w podejściu obiektowym. Funkcja __init__ jest wywoływana za każdym razem, gdy obiekt jest tworzony z klasy. Metoda __init__ pozwala klasie zainicjować atrybuty obiektu i nie służy żadnemu innemu celowi. Jest używany tylko w klasach.

Przykład #1:

```
class Mysliwiec:
    def __init__(self,typMysliwca):
        self.typ = typMysliwca
DuchKijowa = Mysliwiec("MiG-29")
```

Najpierw deklarujemy klasę Mysliwiec za pomocą słowa kluczowego class. Używamy słowa kluczowego def, aby zdefiniować funkcję lub metodę, taką jak metoda __init__. W powyższym przykładzie metoda __init__ inicjuje atrybut typ. Słowo kluczowe self, wiąże atrybuty nowoutworzonego obiektu DuchKijowa z otrzymanym argumentem („MiG-29”).

Metoda __len__

Metoda __len__ Pythona zwraca dodatnią liczbę całkowitą, która reprezentuje długość obiektu, na którym jest wywoływana. Implementuje wbudowaną funkcję len(). Wynik len(x) i x.__len__() jest taki sam: obie zwracają liczbę elementów w obiekcie, czyli jego długość. Funkcja len() wywołuje

wewnętrznie x.__len__() w celu określenia długości obiektu. Różnica między len() a x.__len()__ jest czysto syntaktyczna nie semantyczna.

Przykład #2:

```
>>> print(len([1, 2, 3]))
3
>>> print([1, 2, 3].__len__())
3
```

Metoda __getitem__

To metoda niejawnego przeciążania operatorów (przeciąża indeksowanie, wycinanie, iterację), która pozwala na wydobywaniu elementów po indeksie. Dla klas implementujących metodę __getitem__, która wywoływana jest przy operacji indeksowania, otrzyma ona liczbę całkowitą.

Przykład zastosowania metody __getitem__ na potrzeby iteracji:

```
class Indexer:
    def __getitem__(self, index):
        return index ** 2 #podniesienie indeksu do kwadratu


X = Indexer()
X[2] # X[i] wywołuje X.__getitem__(i)


for i in range(5):
    print(X[i], end='  ') # Przy każdej iteracji wywołuje
__getitem__(X, i)
```

Przykład zastosowania metody__getitem__ w wyrażeniach wycinania (wycinanie z użyciem składni wycinków):

>>> L = [5, 6, 7, 8, 9]

>>> L[2:4] # Wycinanie z użyciem składni wycinków

[7, 8]

>>> L[1:]

[6, 7, 8, 9]

>>> L[:-1]

[5, 6, 7, 8]

>>> L[::2]

[5, 7, 9]

Parametry wycinania są pakowane w specjalny *obiekt wycinka*, który jest przekazywany do instancji listy. Obsługa indeksowania oraz wycinania przy pomocy jednej metody jest zaprojektowana poniżej:

```
class Indexer:
    data = [5, 6, 7, 8, 9]
    def __getitem__(self, index): # Wywoływany przy indeksowaniu i wycinaniu
        print('getitem:', index)
        return self.data[index] # Realizuje dostęp po indeksie lub wycinanie
X = Indexer()
```

Indeksowanie przy pomocy klasy Indexer wygląda następująco:

X[0]

getitem: 0

Out[7]: 5

X[-1]

getitem: -1

Out[8]: 9

X[1]

getitem: 1

Out[9]: 6

X[2:4]

getitem: slice(2, 4, None)

Out[10]: [7, 8]

Wycinanie przy pomocy klasy Indexer wygląda następująco:

X[:]

getitem: slice(None, None, None)

Out[11]: [5, 6, 7, 8, 9]

X[1:]

getitem: slice(1, None, None)

Out[12]: [6, 7, 8, 9]

X[:-1]

getitem: slice(None, -1, None)

Out[14]: [5, 6, 7, 8]

Instrukcja *for* przy każdej iteracji odczytuje jeden element, wykorzystu jąc kolejny indeks, licząc od zera, aż zostanie wywołany wyjątek końca zakresu. Z tego powodu metoda __getitem__ może być jednym ze sposobów implementacji iteratora w Pythonie: jeśli pętla for wykryje tę metodę, będzie ją wywoływać z kolejnymi indeksami. Każdy obiekt obsługujący indeksowanie potrafi również obsłużyć iterację:

```
class stepper:
    def __getitem__(self, i):
        return self.data[i]
X = stepper()#X jest instancją klasy stepper
```

Metoda __setitem__

Metoda __setitem__ implementuje analogiczny mechanizm przypisywania wartości zarówno dla indeksów, jak i dla wycinków — w tym ostatnim przypadku również otrzymuje obiekt wycinka, który można przekazać dalej tak samo jak w przypadku metody __getitem__.

```
def __setitem__(self, index, value): # Przechwytuje przypisania do indeksu lub wycinka
    self.data[index] = value # Przypisanie do indeksu lub wycinka
```

W tab. 1 znaleźć można podsumowanie zaprezentowanych w niniejszym rozdziale metod dunder. Pełną listę metod magicznych (jest ich 127) wraz z objaśnieniami znaleźć można pod https://blog.finxter.com/python-dunder-methods-cheat-sheet/.

Tab. 1: Nazwy metod specjalnych.

Nazwa metody	Przeciąża	Wywoływana dla
`__init__`	Konstruktor (tworzenie instancji)	`X = Klasa(args)`
`__len__`	Podaje długość struktury danych	`len(X)`
`__getitem__`	Indeksowanie, wycinanie, iteracje	`X[klucz]`, `X[i:j]`, pętle for oraz inne iteracje, jeśli nie ma __iter__
`__setitem__`	Przypisanie indeksu i wycinka	`X[klucz] = wartość`, `X[i:j] = sekwencja`

Przykład #1: Talia jako sekwencja kart.

```
import collections


Card = collections.namedtuple('Card', ['rank', 'suit']) #rank - figura; suit - kolor

class PolishDeck:

    ranks = [str(n) for n in range(2, 11)] + list('WDKA') #W - Walet; D - Dama; K - król; A - as

    suits = 'spades diamonds clubs hearts'.split() #spades - piki; diamonds - karo; clubs - trefl; hearts - kier

    def __init__(self):

        self._cards = [Card(rank, suit) for suit in self.suits

                       for rank in self.ranks]

    def __len__(self):

        return len(self._cards)

    def __getitem__(self, position):

        return self._cards[position]
```

Pierwszą rzeczą, na którą należy w powyższym przykładzie zwrócić uwagę, jest użycie collections.namedtuples do skonstruowania klasy, do reprezentowania poszczególnych kart w talii. Poczynając od wersji Pythona 2.6, namedtuple może być używane do budowania klas obiektów, które są pakietami atrybutów bez niestandardowych metod, jak np. rekordy bazodanowe:

>>> exemplar_card = Card('7', 'diamonds')

>>> print(exemplar_card)

Card(rank='7', suit='diamonds')

Podobnie jak dowolna inna pythonowa kolekcja, talia (deck) odpowiada na len() poprzez zwrócenie liczby kart w talii:

```
>>> deck = PolishDeck()
>>> print(len(deck))
52
```

Odczytanie określonych kart z talii, np. pierwszej i ostatniej, jest trywialne poprzez wykorzystanie deck[0] i deck[-1] co zapewnia metoda __getitem__:

```
>>> print(deck[0])
Card(rank='4', suit='hearts')
>>> print(deck[-1])
Card(rank='A', suit='clubs')
```

Celem losowego doboru karty z zaprojektowanej talii (sekwencji elementów) można wykorzystać funkcję choice z pakietu random (random.choice):

```
>>> print(choice(deck))
Card(rank='3', suit='hearts')
>>> print(choice(deck))
Card(rank='9', suit='hearts')
>>> print(choice(deck))
Card(rank='D', suit='diamonds')
```

Ponieważ __getitem__ odnosi się do operatora [] struktury self._cards, zaprojektowaną uprzednio talię można pokawałkować.

Załóżmy, że chcemy wybrać trzy karty z wierzchu nowoutworzonej talii jart:

```
>>> print(deck[:3])
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'), Card(rank='4', suit='spades')]
```

Załóżmy, że chcemy wybrać z nowoutworzonej talii kart tylko asy, poczynając od karty z indeksem 12 i przeskakując po talii co 13 kart:

```
>>> print(deck[12::13])
```

[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'), Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]

Implementacja metody specjalnej __getitem__ pozwala uczynić talię kart iterowalną:

>>> for card in deck: # doctest: +ELLIPSIS

… print(card)

Card(rank='2', suit='spades')

Card(rank='3', suit='spades')

Card(rank='4', suit='spades')

…

Talię można również iterować w odwrotnej kolejności:

>>> for card in reversed(deck): # doctest: +ELLIPSIS

… print(card)

Card(rank='5', suit='diamonds')

Card(rank='4', suit='clubs')

Card(rank='10', suit='hearts')

…

Iteracja jest często niejawna. Jeśli kolekcja nie ma metody __contains__, operator in wykonuje skanowanie sekwencyjne. In działa z PolishDeck dlatego, że jest iterowalny:

>>> Card('D', 'hearts') in deck

True

>>> Card('7', 'beasts') in deck

False

Typowo szereguje się karty według figury (asy to najwyższa karta), a następnie kolorami w kolejności pik (najwyższy), następnie kier, karo i trefl. Oto funkcja, która porządkuje karty wg. tej zasady, zwracając 0 dla 2-jki trefl i 51 dla asa pik:

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)
def spades_high(card):
    rank_value = PolishDeck.ranks.index(card.rank)
```

```
    return rank_value * len(suit_values) + suit_values[card.suit]
```

Mając określoną spades_high, można teraz poszeregować talie rosnąco figurami:

>>> for card in sorted(deck, key=spades_high):

… print(card)

Card(rank='2', suit='clubs')

Card(rank='2', suit='diamonds')

Card(rank='2', suit='hearts')

Card(rank='2', suit='spades')

…

Card(rank='A', suit='clubs')

Card(rank='A', suit='diamonds')

Card(rank='A', suit='hearts')

Card(rank='A', suit='spades')

Chociaż PolishDeck niejawnie dziedziczy z obiektu, jest funkcjonalność nie jest dziedziczona, ale pochodzi z wykorzystania modelu danych i kompozycji. Dzięki implementacji specjalnych metod __len__ i __getitem__, PolishDeck zachowuje się jak standardowa sekwencja Pythona, pozwalając mu korzystać z podstawowych funkcji języka (np. iteracji i krojenia (slicing)) oraz z biblioteki standardowej, jak to zaprezentowano na przykładach z wykorzystaniem random.choice, reversed i sorted. Dzięki kompozycji, implementacje __len__ i __getitem__ mogą przekazać całą pracę obiektowi listy self.__cards.

Przykład #2: Poruszanie figurami na szachownicy do gry w warcaby z wykorzystaniem listy

OPIS ZADANIA: Załóżmy, że chcemy wygenerować sześć przypadkowych ruchów pionka na szachownicy warcabów o rozmiarach 8x8. Wartość -1 oznacza ruch w lewo, +1 oznacza ruch w prawo. Startujemy z pola o współrzędnych (2,2).

6 przypadkowych ruchów pionka na szachownicy warcabów

```
import random
kierunek=[ random.choice([-1,1]) for i in range(6) ]
wspolrzedne=[]
x=2
for y in range(6):
    if (x + kierunek[y]) == 0:
```

```
        x = 2
    elif (x + kierunek[y] > 8):
        x = 7
    else:
        x += kierunek[y]
        wspolrzedne.append([x,y+2])
```

Przykład #3: Przypadkowe rozrzucanie pionków od warcabów na planszy.

```
import random
# plansza = [[0 for _ in range(8)] for _ in range(8)]
plansza = [8 * [0] for _ in range(8)]
ilosc = 0
while (ilosc < 12):
    x = random.choice(range(8))
    y = random.choice(range(8))
# if (((x+y) % 2) == 1) and (plansza[y][x] == 0):
    czarne = ((x+y) % 2) == 1
    if (czarne) and (plansza[y][x] == 0):
        plansza[y][x] = 1
        ilosc += 1
for y in range(8):
    print(plansza[y])

#Pierwsza część warunku (zob. wiersz komentarza) sprawdza, czy pole
#jest czarne. Użycie zmiennej nie tylko upraszcza warunek, ale
#pozwala zachować informację o polu i stanowi dodatkowe objaśnienie
#(nazwa zmiennej).
```

Przykład #4: Poruszanie pionkiem w wersji obiektowej.

```
import random
PRAWO=1
LEWO=-1
```

```
class Pionek:
    x=y=1
    def __init__(self,x,y):
        self.x=x
        self.y=y

    def losowe_ruchy(self):
        while self.y<8:
            self.y +=1
            dx=random.choice([PRAWO, LEWO])
            if (self.x+dx==0) or (self.x+dx>8):
                self.x -= dx
            else:
                self.x += dx
            yield (self.x,self.y)

p = Pionek(1,1)
wspolrzedne = [ _ for _ in p.losowe_ruchy ()]
```

BIBLIOGRAFIA:

[1] M. Matsumoto and T. Nishimura, „Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom numer generatore”, ACM Transactions on Modeling and COmputer Simulation, vol. 8, no. 1, January pp. 3-30, 1998.

[2] Wichmann, B.A. and Hill, I.D., „Algorithm AS 183: An efficient and portable pseudo-random numer generator”, Applied Statistics 31 (1982), pp. 188-190.

[3] https://github.com/python/cpython/blob/main/Lib/random.py#L71