KOLEKCJE W PYTHONIE

Sebastian Buczyński @ Python-Łódź 10.2016

HELLO THERE

Technical Lead w Focus Telecom Polska

Z Pythonem od 2012. roku

INNE WYSTĄPIENIA

Pisanie testów z perspektywy programisty
PHPers Łódź #3

Kiedy Python nie wystarcza
Python Łódź 03.2016

Distributed locks with Python and Redis
EuroPython2015

Pisanie testów dla początkujących
PyConPL 2015

PLAN

Najpopularniejsze kolekcje z biblioteki standardowej

Przyjrzenie się ich implementacji
mocne i słabe strony

OŚWIADCZENIE

Szczegóły implementacyjne dotyczą CPythona 3.5

LISTA

Sekwencja obiektów, którą można indeksować liczbami całkowitymi z zakresu <0; len(l) - 1>

```
some_list = [1, 2, 3]

some_list[0]  # 1
some_list[2]  # 3
some_list[4]  # IndexError
```

LISTA - TYPOWE OPERACJE 1

Dopisywanie/usuwanie elementów z końca (stos)

```
some_list = [1, 2, 3]

some_list.append(4)
print(some_list) # [1, 2, 3, 4]

some_list.pop()
print(some_list) # [1, 2, 3]
```

LISTA - TYPOWE OPERACJE 2

Kontrola obecności elementów

```
some_list = [1, 2, 3]

2 in some_list # True

some_list.index(2) # 1

some_list.count(3) # 1
```

LISTA - TYPOWE OPERACJE 3

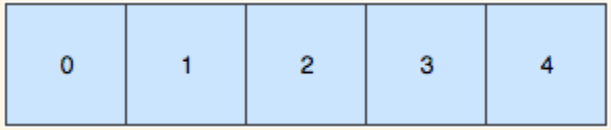
Wstawianie elementów w środek i ich usuwanie

```
some_list = [1, 2, 3]

some_list.insert(1, 100)
print(some_list)  # [1, 100, 2, 3]

some_list.remove(100)
```

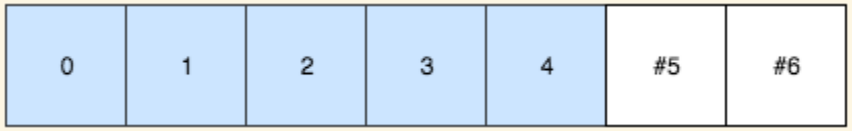
Operacjami *insert*, pop(x) gdzie x!= 0 i *remove* można sobie (najłatwiej) zrobić krzywdę

LISTA - IMPLEMENTACJA 1

Tablica - ciągły obszar pamięci



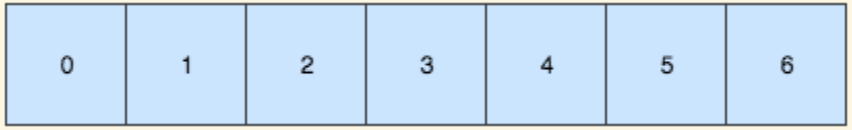
Często większa, niż lista, którą reprezentuje



Zmiana długości listy jest na tyle częstą operacją, że aby ją optymalizować Python zostawia dodatkowe miejsce

LISTA - IMPLEMENTACJA 2

Listy dynamicznie zmieniają rozmiar



Alokacja jest robiona oszczędnie, zoptymalizowana pod dołączanie kolejnych elementów

LISTA - IMPLEMENTACJA 3

Alokujemy trochę więcej niż potrzeba


```
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);
```

Dla list o rozmiarze < 9

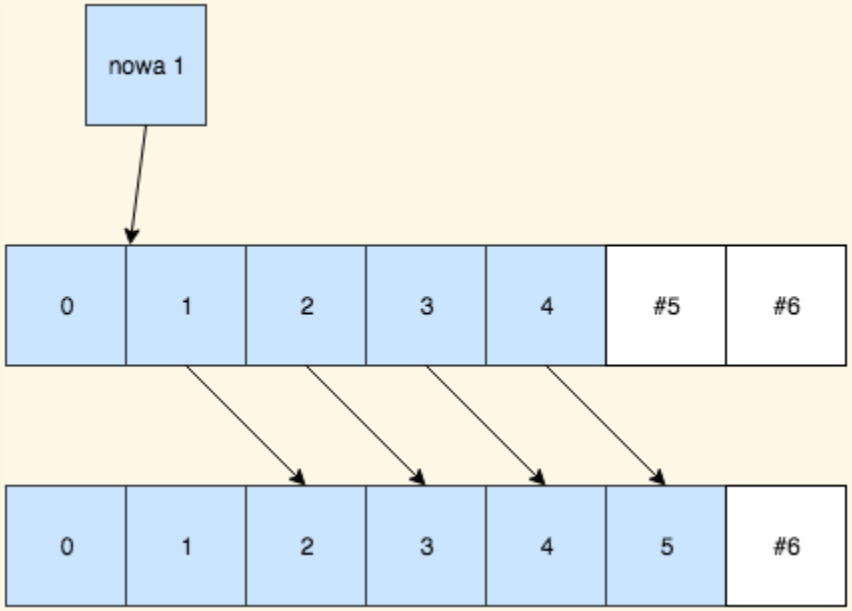
```
new_allocated = newsize / 8 + 3;
```

Dla list o rozmiarze >= 9

```
new_allocated = newsize / 8 + 6;
```

LISTA - IMPLEMENTACJA VS RZECZYWISTOŚĆ 1

Wstawianie elementów na początku/w środek jest kosztowne, gdyż wymaga przesunięcie wszystkich pozostałych - O(n).



LISTA - IMPLEMENTACJA VS RZECZYWISTOŚĆ 2

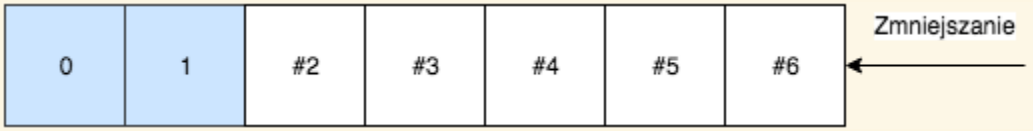
Łączenie list

```
some_list = [1, 2, 3]
some_list.extend(list(range(100000)))  # OK

some_list += list(range(100000))  # OK

for el in range(100000):
    some_list.append(el)  # NOT OK
```

Łącząc listy w dwóch pierwszych przypadkach alokacja pamięci odbędzie się raz - w trzecim wielokrotnie.

LISTA - POZOSTAŁE UWAGI

Lista zostanie zmniejszona, jeżeli będzie zajęte poniżej 50% zarezerwowanej pamięci



DEQUE (TALIA)

interfejs podobny do listy, jednak z wydajnym dodawaniem/usuwaniem elementów na obu końcach

```
from collections import deque
deque([1, 2, 3])
```

DEQUE - TYPOWE OPERACJE 1

Dopisywanie/usuwanie elementów na początku i końcu

```
some_deque = deque([1, 2, 3, 4, 5])
some_deque.popleft()  # deque([2, 3, 4, 5])

some_deque.pop()  # deque([2, 3, 4])

some_deque.appendleft(0)  # deque([0, 2, 3, 4])
some_deque.append(6)  # deque([0, 2, 3, 4, 6])
```

DEQUE - TYPOWE OPERACJE 2

Tablica/lista cykliczna

```
some_deque = deque([1, 2, 3])
some_deque[0]  # 1

some_deque.rotate(-1)
some_deque[0]  # 2

some_deque.rotate()
some_deque[0]  # 1
```

d.rotate() jest równoznaczne z d.appendleft(d.pop())

DEQUE - TYPOWE OPERACJE 3

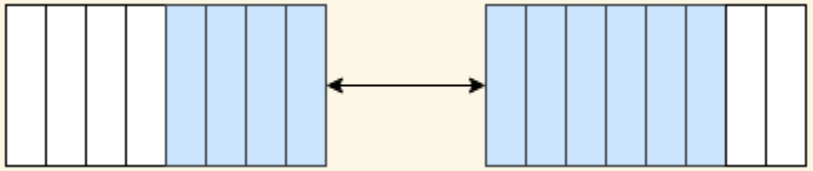
Bufor, n ostatnich obiektów

```
some_deque = deque(maxlen=3)
for number in range(4):
        some_deque.append(number)

print(some_deque)  # deque([1, 2, 3], maxlen=3)
```

Nowe elementy dołączane z przodu lub z tyłu 'wypychają' z kolekcji elementy z drugiego końca deque

DEQUE - IMPLEMENTACJA 1

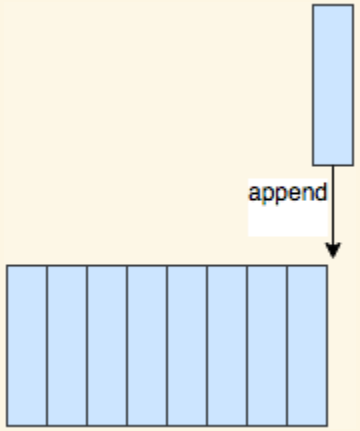
Wiele list połączonych ze sobą, częściowo lub w całości wypełnione



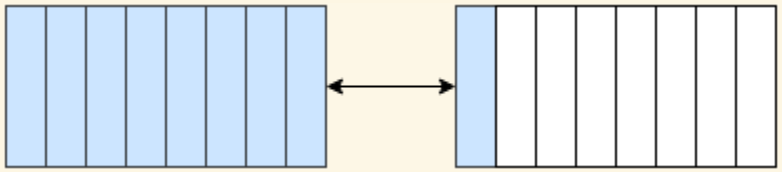
Bloki stałej wielkości - 64 (CPython 3.5!)

DEQUE - IMPLEMENTACJA 2

Wydajne doczepianie elementów z obu stron



Dane lądują w nowozaalokowanym bloku pamięci



DEQUE VS LISTA

Lista szybsza niż deque podczas append'ów?

To realokacje pamięci są najbardziej czasochłonnymi operacjami

Skoro deque alokuje zawsze bloki po 64 elementy, a lista zależnie od swojej wielkości, to...

```
new_allocated = newsize / 8 + 6 # dla wiekszych list
```

```
64 < newsize / 8 + 6
```

```
58 < newsize / 8
```

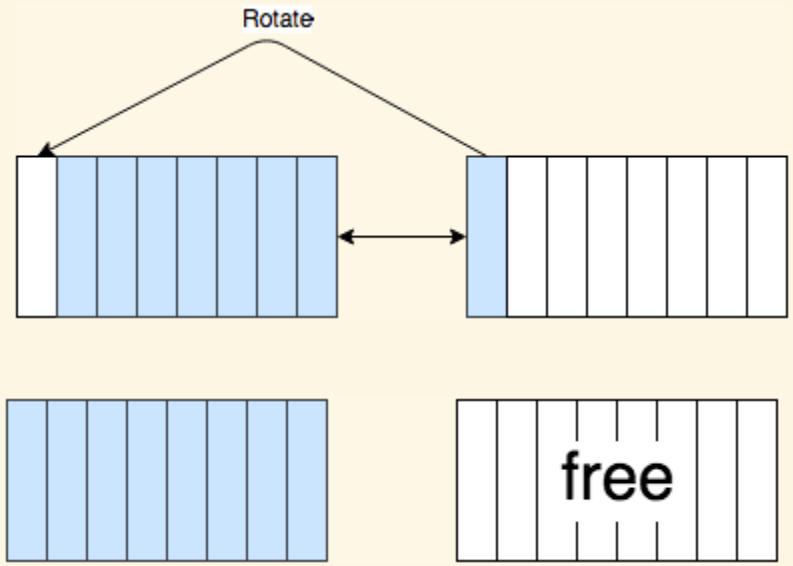
```
58 * 8 < newsize
```

```
464 < newsize
```

To nie do konca prawda, gdyż lista musi zajmować ciągły obszar pamięci...

DEQUE - POZOSTAŁE UWAGI

Puste bloki będą zwalniane, na przykład po rotacji



Deque nigdy nie będzie trzymać więcej niż 126 pustych komórek

SŁOWNIK (DICT)

Szerokie zastosowanie, różne charakterystyki

```
some_dict = dict()  # some_dict = {}
some_dict['key'] = 'value'

'key' in some_dict  # True

some_dict['key']  # 'value'
```

Najszybszą operacją ma być indeksowanie - wstawianie elementu po kluczu i późniejszy dostęp

DICT W SŁUŻNIE INTERPRETERA 1

keyword arguments - kilka elementów, jeden zapis, jeden odczyt

```
def fun(name=user_name, email=email):
    pass
```

DICT W SŁUŻNIE INTERPRETERA 2

Class method lookup - kilka - kilkanaście elementów, zwykle jeden zapis, wiele odczytów

```
class A:
  @classmethod
  def lookup_for_me(cls):
    pass
```

Słownik klasy, a dziedziczenie

```
class B(A):
  pass

B.lookup_for_me()  # wymaga siegniecia do slownika klasy bazowej
```

DICT W SŁUŻNIE INTERPRETERA 3

instance attributes - kilka - kilkanaście elementów, wiele zapisów i odczytów

```
class A:
  def __init__(self, name, last_name):
    self.name = name
    self.last_name = last_name

a = A('Sebastian', 'Buczynski')
a.__dict__  #  {'last_name': 'Buczynski', 'name': 'Sebastian'}
```

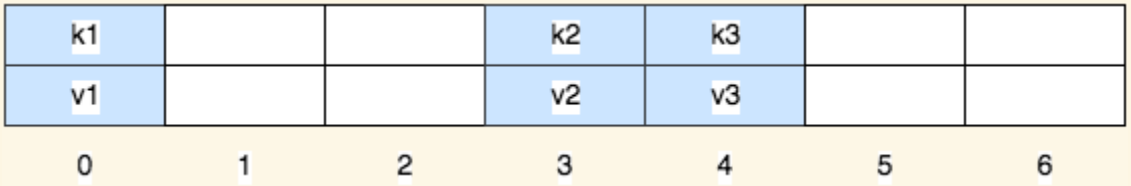
KLASY NA DIECIE - DYGRESJA

Słownik jest stosunkowo pamięciożerny. Można swoje klasy odchudzić - używając __slots__

```
class A:
  __slots__ = ['name', 'last_name']
  def __init__(self, name, last_name):
    self.name = name
    self.last_name = last_name

a.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute '__dict__'</module></stdin>
```

DICT - IMPLEMENTACJA 1

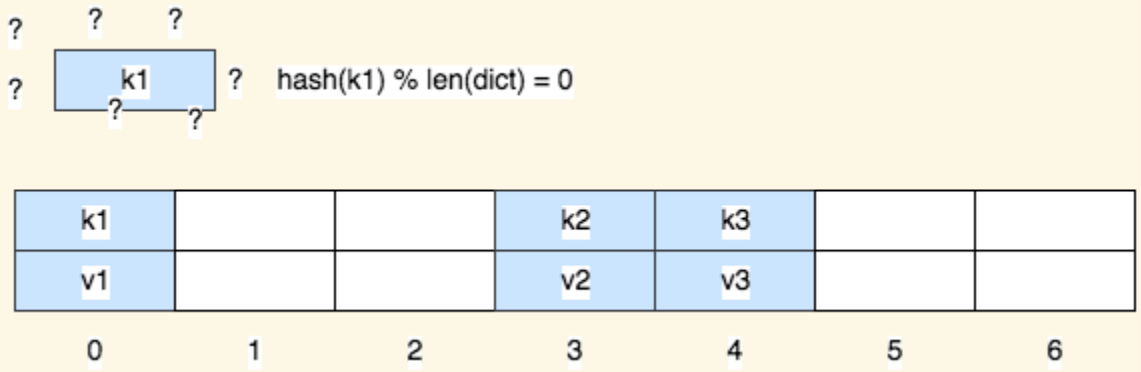
Słownik jest zaimplementowany jako tablica haszująca - optymalne dla szybkiego znajdywania elementów



Pod maską jest to "rzadka" tablica - co najmniej 1/3 przestrzeni jest pozostaje nieużywana

DICT - IMPLEMENTACJA 2

Słownik "tłumaczy" klucz na pozycje w znajdującej się pod spodem tablicy - haszowanie



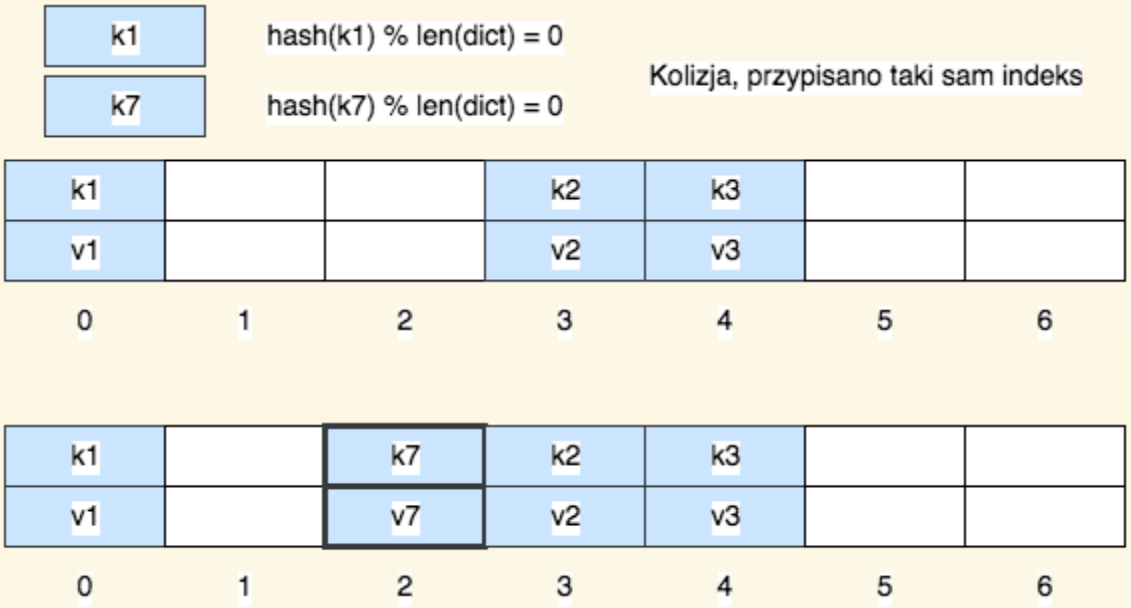
k1 musi posiadać metodę __hash__

Pythonowe haszowanie wyróżnia to, że podobne wartości otrzymują bliskie sobie indeksy - hash(1) = 1, hash(2) = 2

DICT - IMPLEMENTACJA 3

Po co tyle wolnego miejsca?

KOLIZJE



Utrzymanie 1/3 wolnej przestrzeni jest wymagane ze względu na rozwiązywanie kolizji

DICT - IMPLEMENTACJA 4

Rozszerzanie słownika tanie nie jest

Jeżeli zajętość słownika przekroczy granicę 2/3, to zmieniamy jego rozmiar zgodnie z wzorem:

```
new_size = used * 2 + capacity / 2
```

jeżeli tylko dodajemy nowe wartości nie usuwając poprzednich, to słownik będzie zwiększał się 2-krotnie

DICT - PRZYDATNE KLASY POCHODNE W BIBLIOTECE STANDARDOWEJ

DEFAULTDICT

Przy konstrukcji podajemy jeden argument - callable, który utworzy nieistniejący element w przypadku próby odwołania

```
from collections import defaultdict

d = defaultdict(lambda : [])
d['key1'].append(3)  # KeyError w przypadku zwykłego słownika
print(d)  # defaultdict(<function <lambda="">> at 0x102183a60>, {'key1':
```

COUNTER

Proste zliczanie elementów sekwencji

```
from collections import Counter

c = Counter(['a', 'b', 'a', 'c', 'a', 'c'])
print(c)  # Counter({'a': 3, 'c': 2, 'b': 1})
```

ORDEREDDICT

Słownik zapamiętujący kolejność wstawiania elementów

```
from collections import OrderedDict
od = OrderedDict()
od['key'] = 'some_val'
od['other_key'] = 'other_val'

print(od)  # OrderedDict([('key', 'some_val'), ('other_key', 'other_val'
```

Materiał przygotowany na podstawie:

- źródeł CPythona w wersji 3.5
- *Piękny kod* Andy'ego Orama oraz Grega Wilsona (niestety mocno przestarzały)

DZIĘKI ZA WYSŁUCHANIE

PYTANIA?