

ELOQUENT PYTHON PROGRAMMING

STRUKTUR DATA

Biasanya kita akan bekerja dengan sejumlah data yang lumayan banyak, dan daripada menamai nilainya satu per satu kita menyusunnya dalam suatu struktur dalam memori yang sesuai, cocok untuk pemrosesannya. Struktur data bawaan Python terdiri atas beberapa macam, antara lain fixed structured, array, doubly linked list, hashmap, yang disertai spesifikasi interfacenya atau class seperti:

1. tuple, daftar beragam objek dengan urutan indeks, anggotanya tidak dapat diedit (immutable)
2. list, daftar beragam objek dengan urutan indeks, anggota dapat diubah, ditambah, dikurangi
3. dict, kumpulan pasangan kunci&nilai, kunci tidak dapat diedit, nilai dapat diedit, anggota dapat dibuang, ditambah
4. set, kumpulan objek tanpa duplikat, tanpa urutan
5. str, deretan karakter, dengan urutan indeks, tidak dapat diedit

dan beberapa tambahan dari modul pustaka sertaan:

1. array
2. double end queue deque
3. namedtuple

In [3]:

```
from array import array
from collections import deque, namedtuple

for cls in (tuple, list, dict, set, str, array, deque):
    metoda = [k for k in dir(cls) if '_' not in k] # List comprehension
    print(cls, ':')
    print(metoda)
    print('~'*20)

<class 'tuple'> :
['count', 'index']
~~~~~
<class 'list'> :
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
~~~~~
<class 'dict'> :
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
~~~~~
<class 'set'> :
['add', 'clear', 'copy', 'difference', 'discard', 'intersection', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'union', 'update']
~~~~~
<class 'str'> :
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitli
```

```

nes', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
~~~~~
<class 'array.array'> :
['append', 'byteswap', 'count', 'extend', 'frombytes', 'fromfile', 'fromlist', 'fromunic
ode', 'index', 'insert', 'itemsize', 'pop', 'remove', 'reverse', 'tobytes', 'tofile', 't
olist', 'tounicode', 'typecode']
~~~~~
<class 'collections.deque'> :
['append', 'appendleft', 'clear', 'copy', 'count', 'extend', 'extendleft', 'index', 'ins
ert', ' maxlen', 'pop', 'popleft', 'remove', 'reverse', 'rotate']
~~~~~

```

Source code dari class namedtuple yang dapat dilihat dengan fungsi inspect.getsource(), sbb:

```
In [ ]: from inspect import getsource
print(getsource(namedtuple))
```

TUPLE

Tentang tuple sudah banyak publikasi penjelasan dari sudut interface atau spesifikasinya, dan kali ini kita ingin melihat juga dari sisi implementasi strukturnya.

Struktur tuple dapat dibayangkan mirip dengan struktur records pada Pascal atau structs pada C, merupakan sekumpulan kecil data dari berbagai tipe yang dioperasikan sebagai suatu grup. Tuple terdiri atas deretan objek pointer (64-bit pada sistem 64 bit) yang menunjuk ke berbagai objek di luar tuple itu sendiri, seperti terlihat dari alamat tuple dan alamat elemen-elemennya pada kode berikut.

Pada sistem saya 64-bit, ukuran bytes tuple selalu bertambah dengan kelipatan 8 bytes jika elemennya ditambah dengan objek, dari sembarang tipe, yang ukurannya jauh lebih besar dari 8 bytes.

```
In [22]: t1=(65535,)
t2=(65535,'hello')
t3=(65535,'hello',22/7)
t5 = (28,29,30,65536,22/7,'hello','tuple')
#alamat tuple dibandingkan dengan alamat anggotanya
for o in *t5, t5:
    print(hex(id(o)),o)
#ukuran alokasi memori tuple dibanding ukuran anggotanya
for o in t1,t2,t3,t5, *t5:
    print(o.__sizeof__(), o)
```

```

0x231573c0450 28
0x231573c0470 29
0x231573c0490 30
0x2315bc6ec10 65536
0x2315bc6e810 3.142857142857143
0x2315b8f1270 hello
0x2315745ce30 tuple
0x2315bbfcbe0 (28, 29, 30, 65536, 3.142857142857143, 'hello', 'tuple')
32 (65535,)
40 (65535, 'hello')
48 (65535, 'hello', 3.142857142857143)
80 (28, 29, 30, 65536, 3.142857142857143, 'hello', 'tuple')
28 28
28 29

```

```
28 30
28 65536
24 3.142857142857143
54 hello
54 tuple
```

Tuple adalah immutable dalam artian anggotanya tidak dapat diedit, ditambah, tetapi tuple dapat dibuat lagi dengan anggota yang baru.

```
In [2]: t1 = ((2021, 'lulus sekolah'),)
t1 = t1 + ((2022, 'mulai kerja'),)
print(t1)
```

```
((2021, 'lulus sekolah'), (2022, 'mulai kerja'))
```

Pada contoh di atas tuple t1 bukan di edit tetapi nama t1 dipakai lagi, dikaitkan dengan objek yang baru yaitu gabungan objek t1 yang sebelumnya dengan suatu ekspresi baru. Kita dapat memastikan dengan melihat id objeknya, yaitu alamat objek yang berbeda. (implementasi CPython id = alamat)

```
In [3]: t1 = ((2021, 'lulus sekolah'),)
old_id = id(t1)
t1 = t1 + ((2022, 'mulai kerja'),)
new_id = id(t1)
print(old_id, new_id)
```

```
1479522468672 1479521798720
```

```
In [4]: t1 = ((2021, 'lulus sekolah'),)
obj0 = id(t1[0])
t1 = t1 + ((2022, 'mulai kerja'),)
obj1 = id(t1[0])
print(obj0, obj1)
```

```
1479521717120 1479521717120
```

Kita perlu ingat bahwa operasi assignment dalam Python hanya melibatkan *pointer* sedangkan objek aslinya sendiri tidak berpindah, tidak dicopy. Pada contoh di atas objek elemen pertama tetap digunakan pada tuple yang baru, dilihat dari alamat objeknya yang sama.

```
In [5]: t2 = t1[:] #tuple aliasing
print(t1 is t2)
```

```
True
```

```
In [6]: Lst1 = list([1,2,3,4])
Lst2 = Lst1[:] # list cloning
print(Lst1 is Lst2)
```

```
False
```

Perlu diperhatikan juga bahwa **tuple tidak mendukung cloning**, karena sifatnya yang *immutable* maka tidak ada alasan untuk mendukung cloning, membuat objek baru dengan isi yang sama. Jadi walaupun kita menggunakan sintaks yang biasa untuk membuat cloning pada list, efeknya pada

tuple adalah aliasing. (cloning adalah membuat objek berbeda berisikan elemen yang sama, sedangkan aliasing adalah memberi nama yang berbeda untuk objek yang sama)

In [2]:

```
...
tuple, kumpulan objek yang dapat diiterasi
generator expressions, mirip list comprehension tapi menghasilkan objek generator
unpacking operator asterisk * memiliki efek mengiterasi generator
...
for klas in (tuple, list,):    # for'each'-in Loop dari iterable tuple
    metodas = (m for m in dir(klas) if '_' not in m)    # membentuk suatu generator
    print(metodas)
    print(*metodas, '\n')    # asterisk * unpacking generator, mengeluarkan elemen-2 nya.
```

<generator object <genexpr> at 0x0000018FDC2C5D90>
count index

<generator object <genexpr> at 0x0000018FDC2C5E00>
append clear copy count extend insert pop remove reverse sort

Iterable ialah suatu objek yang mampu mengembalikan elemen anggotanya satu per satu. Contoh iterable antara lain semua tipe sekuen (seperti list, tuple, str), dan beberapa tipe non-sekuens seperti dict, objek file, dan juga objek dari klas yang memiliki metoda `__iter__()` atau `__getitem__()`, yang mengimplementasikan semantik sekuen.

Iterable dapat digunakan bersama for loop atau bersama fungsi lain yang memerlukan sekuen seperti `map()`, `zip()`.

Generator ialah suatu fungsi yang mengeluarkan elemen di dalamnya satu per satu, dan cenderung lebih hemat memori daripada list yang dibentang isi elemennya. Generator dapat digunakan bersama for loop, atau fungsi `next()` yang hanya mengeluarkan satu elemen setiap kali dipanggil. (ekspresi bintang, asterisk *) juga dapat digunakan dalam kasus tertentu)

LIST

List adalah suatu array yang ukurannya dapat berubah ubah. Implementasinya menempati satu lokasi memori yang utuh tidak terpecah, berisi pointer pointer objek yang diletakkan berdampingan, dan pada kepala list tercantum pointer ke lokasi ini, serta jumlah elemen atau panjang list.

Pada akhir dari lokasi array sudah dicadangkan tempat untuk pointer baru jika anggota array bertambah, dan jika cadangan ini habis maka akan dialokasikan tempat (bisa jadi dengan alamat baru) dengan cadangan yang lebih besar lagi, karena setiap alokasi tempat baru akan memakan ongkos waktu untuk menyalin semua pointer yang ada dari lokasi lama ke dalamnya.

Proses mencari anggota dalam list (pointer ke objek) dapat dilakukan langsung tanpa tergantung dari jumlah anggota list, karena alamatnya dapat diperoleh dengan perkalian indeks dan ukuran pointer ditambah offset awal dari lokasi, sama seperti mengakses RAM. Notasi kompleksitasnya adalah konstan $O(1)$. Penambahan anggota di akhir list, dapat dianggap kecil juga karena ongkos tambahan penyalinan di-amortisasi, ditanggung bersama anggota-anggota yang ditambahkan langsung selama cadangan belum habis terpakai, lalu ukuran cadangan yang makin membesar setiap kali dibuat baru.

In [4]:

```

list1 = list()
list2 = [28]
list3 = [28,42]
list5 = [28,42,255,'list array of pointers']

for obj in list5, *list5:
    print(hex(id(obj)))
for obj in list1,list2,list3:
    print(obj.__sizeof__())

```

```

0x2315bcf9f00
0x231573c0450
0x231573c0610
0x231573c20b0
0x2315bccca470
40
48
56

```

ARRAY

In [20]:

```

from array import array
arr1 = array('b',[1])
arr2 = array('b',[1,2])
arr3 = array('b',[1,2,3])
arr4 = array('b',[1,2,3,4])
arr5 = array('b',[1,2,3,4,5])
for o in arr5, arr5[0], arr5[1], arr5[2]:
    print(hex(id(obj)))
print(f'{arr5.itemsize = } byte(s)')
for o in arr1, arr2, arr3, arr4, arr5:
    print(o.__sizeof__(), o)
for i in range(6,12):
    arr5.append(i)
    print(arr5.__sizeof__(), arr5)

```

```

0x2315bcfa000
0x2315bcfa000
0x2315bcfa000
0x2315bcfa000
arr5.itemsize = 1 byte(s)
65 array('b', [1])
66 array('b', [1, 2])
67 array('b', [1, 2, 3])
68 array('b', [1, 2, 3, 4])
69 array('b', [1, 2, 3, 4, 5])
73 array('b', [1, 2, 3, 4, 5, 6])
73 array('b', [1, 2, 3, 4, 5, 6, 7])
73 array('b', [1, 2, 3, 4, 5, 6, 7, 8])
73 array('b', [1, 2, 3, 4, 5, 6, 7, 8, 9])
81 array('b', [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
81 array('b', [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

```

Implementasi `array.array` ini berupa deretan objek homogen statik, langsung, bukan pointer, terlihat dari alamat dan penambahan ukuran array seperti pada kode di atas. Alamat anggota array adalah sama dengan alamat kepala array ('serumah'). Deklarasi tipe atau ukuran anggota, dalam contoh ini 'b' adalah untuk signed integer 1 byte minimum. Bahkan kita dapat melihat penambahan cadangan

di akhir array, tambah 4, tambah 8, makin membesar. Kompleksitas waktu sama dengan list, kompleksitas ruang sama, tetapi pemakaian absolut memori lebih hemat, dapat diatur secara statik.

DEQUE

Menurut Raymond Hettinger, proses insert elemen di posisi awal pada suatu list menggunakan waktu yang tidak sedikit, $O(n)$, sehingga jika suatu program besar banyak melakukan hal itu maka kinerja keseluruhan akan menurun. Itulah sebabnya dia menambahkan suatu struktur data doubly-linked list, double ended queue atau disingkat deque dengan metoda andalannya *appendleft*, *extendleft*, *popleft*, dalam modul pustaka collections.

Operasi append, pop pada posisi awal/akhir deque memiliki kompleksitas waktu konstan $O(1)$.

Tetapi struktur doubly-linked list melambat dalam akses elemen di tengah.

In [2]:

```
from time import time
from collections import deque

deck = deque()
start = time()
for n in range(1_000_000):
    deck.appendleft(n)
stop = time()
print(f'deque appendleft {1+deck[0]:,} elemen {stop-start:.2f}')

Lst = list()
nmax = 250_000
start = time()
for n in range(nmax):
    Lst.insert(0,n)
stop = time()
print(f'list insert0 {1+Lst[0]:,} elemen {stop-start:.2f}'')
```

deque appendleft 1,000,000 elemen 0.12

list insert0 250,000 elemen 14.30

DICT dan SET

Struktur dari dict/set adalah hashmap atau sebutan lainnya hashtable. Dict adalah kumpulan objek dengan tatanan berupa pasangan kunci dan nilai, Set adalah kumpulan objek tanpa duplikasi.

Kunci pada dictionary haruslah objek yang hashable karena strukturnya tidak memiliki indeks tetapi menggunakan fungsi hash pada key/kunci sebagai 'indeks' elemennya. Objek yang hashable antara lain integer, float, string, tuple yang beranggotakan hashable object. List termasuk objek yang unhashable, tidak dapat digunakan sebagai kunci.

Dalam hashmap/dict kompleksitas pencarian elemen adalah $O(1)$ dan insertion/deletion juga $O(1)$, artinya konstan tidak tergantung pada banyaknya elemen anggota yang terdaftar.

Jika terjadi *collision* atau fungsi hash atas dua atau beberapa key menghasilkan 'indeks hash' yang sama maka perlu suatu penanganan lagi, misalnya perbaikan fungsi hash, linear probing, quadratic probing, double hash layering, atau menggunakan chaining link list, dan ini akan mempengaruhi performa kompleksitas waktunya.

In [1]:

```
nomor_telp = dict()
nomor_telp[('sam', 'edw')] = '0888999733x'
nomor_telp[('budi', 'di amrik')] = '(818) 785 090x'
```

```
print(nomor_telp[('sam', 'edw')])
print(nomor_telp)
```

```
0888999733x
{('sam', 'edw'): '0888999733x', ('budi', 'di amrik'): '(818) 785 090x'}
```

Salah satu penggunaan dict ialah untuk menghitung jumlah pemunculan huruf-huruf. Berikut contoh kode tentang anagram, permainan kata, yaitu huruf-huruf dalam suatu kata-kata diacak menjadi kata-kata yang berbeda, misalnya beras|sebar|besar, "eleven plus two" | "twelve plus one", "drum piano asbak"|"opa bandar musik". Kode kita hanya akan menghitung huruf dan menyimpulkan apakah dua ungkapan kata merupakan anagram atau tidak.

In [2]:

```
def anagram(word, words):
    def cc(word):
        d={}
        for c in word.lower().replace(' ', ''):
            if c not in d:
                d[c] = 0
                d[c] = d[c] + 1
        return d
    ref = cc(word)
    return [wrds for wrd in words if cc(wrd) == ref]

print(anagram("setec astronomy", ("too many secrets", "my Socrates note")))
print(anagram('beras', ('besar', 'sebar', 'sabar', 'resah')))
```

```
['too many secrets', 'my Socrates note']
['besar', 'sebar']
```

Baris ke-7 pada contoh di atas menghitung kemunculan setiap huruf dalam dictionary dengan kunci berupa hurufnya dan nilainya diakumulasi. Baris ke-5,6,7 dapat diganti menjadi satu baris menggunakan metoda dict.get() sbb:

In [3]:

```
def anagram(word, words):
    def cc(word):
        d={}
        for c in word.lower().replace(' ', ''):
            d[c] = d.get(c, 0) + 1
        return d
    ref = cc(word)
    return [wrds for wrd in words if cc(wrd) == ref]

print(anagram("setec astronomy", ("too many secrets", "my Socrates note")))
print(anagram('beras', ('besar', 'sebar', 'sabar', 'resah')))
```

```
['too many secrets', 'my Socrates note']
['besar', 'sebar']
```

Proses counting dengan dictionary sepertinya banyak dipakai sehingga 'seseorang' (bisa jadi Guido sendiri) membuat class khusus bernama Counter yang juga merupakan implementasi dari bentuk matematis **multiset**, dan disematkan dalam pustaka collections. Akhirnya, kode kita dapat dipadatkan menjadi:

In [4]:

```
from collections import Counter
def anagram(word, words):
```

```

cc = lambda w: Counter(w.lower().replace(' ', ''))

ref = cc(word)
return [wrd for wrd in words if cc(wrd) == ref]

print(anagram("setec astronomy", ("too many secrets", "my Socrates note")))
print(anagram('beras', ('besar', 'sebar', 'sabar', 'resah')))

['too many secrets', 'my Socrates note']
['besar', 'sebar']

```

Kita dapat melihat gaya penulisan fungsional ala Python seperti contoh berikut yang melakukan pengecekan apakah dua ungkapan merupakan anagram atau bukan.

```

In [5]: from collections import Counter
cc = lambda w: Counter(w.lower().replace(' ', ''))
is_anagram = lambda w1,w2: cc(w1) == cc(w2)

print(is_anagram('eleven plus two','twelve plus one'))

```

True

```

In [ ]: import inspect
print(inspect.getsource(Counter))

```