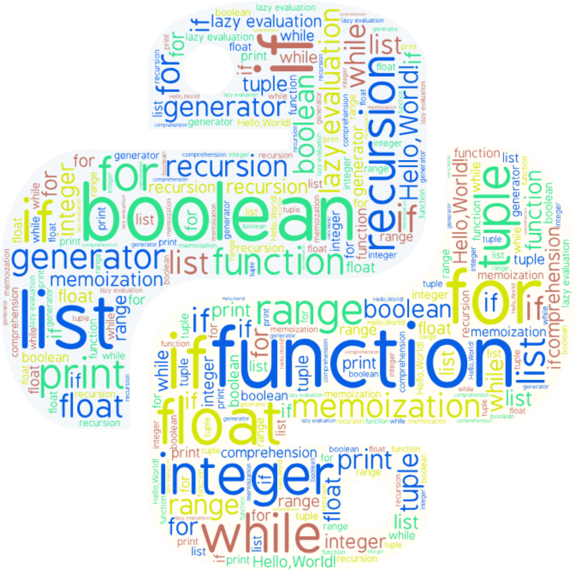
Python 101

Lec05
Collections

thoum

May 8, 2019

What have we learned so far?



Turing Complete

We can do everything$_{\text{a computer can}}$ with what we have learned so far.
But efficiency / simplicity is called for.

Container Datatypes

Things that holds other things (*e.g.* tuples, lists, etc.).

So far, we have used lists and tuples to store values (instead of declaring 100,000 variables). But different times calls for different containers.

Dictionary

Similar to a list, but instead of using *int* as indices, we can use arbitrary objects[1] as indices(or we call them keys)
Instead of comparing the key with every key it has, *dictionary* does it efficiently. [2]

[1] those that are hashable
[2] well, most of the time

How?

Imagine how we use an actual Dictionary. We don't start from 'aardvark' to look up 'floccinaucinihilipilification'. We start from the F section.
We have mapped 'floccinaucinihilipilification' to 'f', so that we don't have to compare every word; we just compare with those in the F section. We call this mapping *hashing*.

Hashing

Mapping data of arbitrary size onto data of a fixed size.
Mapping words to its first letters, mapping students to student IDs, mapping your socks to the colors of white, grey and black to pair them, they are all hashing.

Hashing Example

$a = 1, b = 2, \ldots z = 26$

score of a word $= \Sigma value(c_i)$ *mod* 101

```
def score(word):
    s = sum((ord(c)-ord('a')+1 for c in word))
    return s % 101

print(score('hardwork'))
print(score('luck'))
print(score('attitude'))

# kind of how dictionary works(ignore collision for now)
lst = [0] * 101
lst[score('newton')] = 'gravity'
lst[score('einstein')] = 'e=mc2'

print(lst[score('einstein')])
print(lst[score('newton')])
```

Hashing

Hashing by itself is an important topic with wide range of usage(Encryption, Bitcoins...), but will not go into further details.

Using Dictionaries

```
d = {} # use curly braces to create a dictionary

# the name becomes the key, and the hero identity becomes value
d['brucebanner'] = 'hulk'
d['tonystark'] = 'ironman'
d['peterparker'] = 'spiderman'
d['steverogers'] = 'captainamerica'

print(d['tonystark'])
# overwrite key's value
d['steverogers'] = 'blueskull'
# no key: error
# print(d['thanos'])
```

Using Dictionaries

```
#iterate over keys
for k in d:
    print(k)

#iterate over values
for v in d.values():
    print(v)
```

Using Dictionaries

Operations on iterables(*e.g.* len()) is possible on dictionaries as well.

List vs Dictionary

Input: 151 pokemon's name and number.
Output: Print the pokemon's number when given name.

List Ver.

```
N = 151
pokedex = [0] * 151
for i in range(N):
    pokedex[i] = input().split()


pokemon = input()
for p in pokedex:
    if p[0] == pokemon:
        print(p[1])
        break
```

Dict Ver.

```
N = 151
pokedex = {}
for i in range(N):
    name, number = input().split()
    pokedex[name] = number


pokemon = input()
print(pokedex[pokemon])
```

Performance?

set

Unordered collection of distinct hashable objects.
Used for membership, removing duplicates and so on.

set examples

removing duplicates

```
lst = [1,1,1,1,1,5,1,2,3,10,10]
a = set(lst)
print(a)
```

set examples

testing membership
list version: 2.978s, set version: 0.044s

```
import timeit
import random

l = list(range(10000))
s = set(l)

def f():
    return random.randint(-10000, 10000) in l
def g():
    return random.randint(-10000, 10000) in s

if __name__ == '__main__':
    # repeat f and g 40000 times each
    print(timeit.timeit("f()", setup="from __main__ import
        f",number=40000))
    print(timeit.timeit("g()", setup="from __main__ import
        g",number=40000))
```

set examples

For other operations like union(), intersection(), difference(), look here.

Counter

Counts occurrences

```
from collections import Counter
words = ['red', 'blue', 'red', 'green', 'blue', 'blue']
cnt = Counter(words)

print(cnt['green'])
print(cnt['chicken'])
print(cnt.most_common(2))
```

deque

Double Ended Queue

Use instead of *list* when inserting, popping happens at the beginning of the list. *e.g.* keeping track of values for moving average?

Insert, pop at the beginning of the list creates an overhead of shifting every other element to the left.

deque vs list

list:3.920s, deque: 0.015s

```
from collections import deque
import timeit

l = []
q = deque() # an empty deque

def f():
    l = []
    for i in range(100000):
        #insert at beginning
        l.insert(0, i)
def g():
    q = deque() # an empty deque
    for i in range(100000):
        #insert at beginning
        q.appendleft(i)

if __name__ == '__main__':
    # repeat f and g 40000 times each
    print(timeit.timeit("f()", setup="from __main__ import
        f",number=2))
    print(timeit.timeit("g()", setup="from __main__ import
        g",number=2))
```

deque vs list

List outperforms deque in random access
list:0.111s, deque: 3.597s

```
from collections import deque
import random
import timeit

l = list(range(1000000))
q = deque(range(1000000))

def f():
    return l[random.randint(0,999999)]
def g():
    return q[random.randint(0,999999)]

if __name__ == '__main__':
    # repeat f and g 40000 times each
    print(timeit.timeit("f()", setup="from __main__ import
        f",number=100000))
    print(timeit.timeit("g()", setup="from __main__ import
        g",number=100000))
```

What to use?

In choosing the right *thing* to use, having a slight idea of complexity would help.

Time/Memory Complexity

How long does an operation take? How much memory does it use?

Time Complexity

When input size is N Some operation might take constant time.
e.g. lst[i]
Some operation might take time proportional to N.
e.g. iterating over a list
Some operation might take time proportional to N^2.
e.g. matrix multiplication
Some operation might take time proportional to $Nlog(N)$.
e.g. sorting

Time-Memory Tradeoff

Things can be sped up by using memory. We memorized 99dan, so it takes us constant time to answer 9*9.
We didn't memorize(at least I didn't) every integer multiplication, so it takes us time to answer 938212834 * 41237.

Time-Memory Tradeoff

In C, strlen() counts the number of letters, so it takes time proportional to the length of the string.
In Python, len() returns an internally stored length which is updated everytime the list changes.
There is a tradeoff between memory, and speed.

Time-Memory Tradeoff

Python dicts uses extra memory, and hash computing time, to make access fast enough.
In fact, we can think that Python itself uses more running time and memory to make coding fast enough.