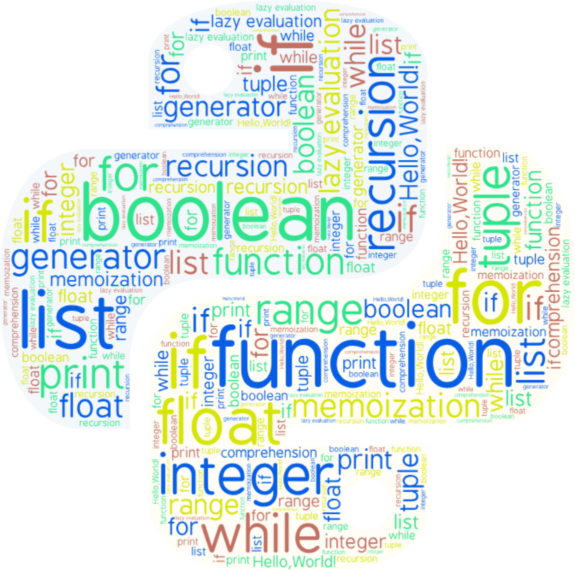
Python 101

Lec04
Useful collections and modules

thoum

August 18, 2019

What have we learned so far?



Turing Complete

In theory, we *can* do everything $_{\text{a computer can}}$ with what we have learned so far. That does not imply efficiency.

Python provides datatypes and *MANY* useful modules that will help us.
Today, we will take a peek at some of them.

Container Datatypes

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

So far, we have used lists and tuples to store values.
But there are different containers for different times.

Dictionary

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

[1] those that are hashable
[2] Hash Collision

How?

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 $= \Sigma value(c_i)\ mod\ 101$

```python
# maps word to number in between 0~100
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

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

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

Using Dictionaries

Dictionaries are iterables as well, so operations on iterables(*e.g.* len()) is OK.

When are dictionaries useful?

1. The obvious: Link A to B (d[A] = B)
2. Use the underlying hashing mechanism to find/search certain object fast.

set

Unordered collection of *unique* hashable objects.

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: 3.6s, set version: 0.05s

```
import time
import random

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

start_time = time.time()
for _ in range(50000):
    random.randint(-10000, 10000) in l
end_time = time.time()
print("search in list took: ", end_time - start_time)

start_time = time.time()
for _ in range(50000):
    random.randint(-10000, 10000) in s
end_time = time.time()
print("search in set took: ", end_time - start_time)
```

set examples

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

```
dir(set)
```

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))
```

Other operations are also found on the website.

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

```python
from collections import deque
import time

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__':
    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__':
    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

Pool

Parallel Computing is an advanced topic, as it requires careful coding to avoid errors like deadlock. (*e.g*. dining philosophers) But, Python's Pool allows us to do simple parallel computing.(use with care)

Pool

```
from multiprocessing import Pool


def f(n):
    return n*n

l = [1,2,3,4,5,6,7,8]

with Pool() as p:
    result = p.map(f, l)

print(result)
```

Pool Explained

Pool(4) means we will use 4 cpus. Pool() defaults to $os.cpu\_count$().

Pool Explained

with keyword handles initalization and cleanup of certain operations

```
# p is only effective within the 'with'
with Pool() as p:
    # do something

p = Pool()
# do something
p.close()
p.join()
```