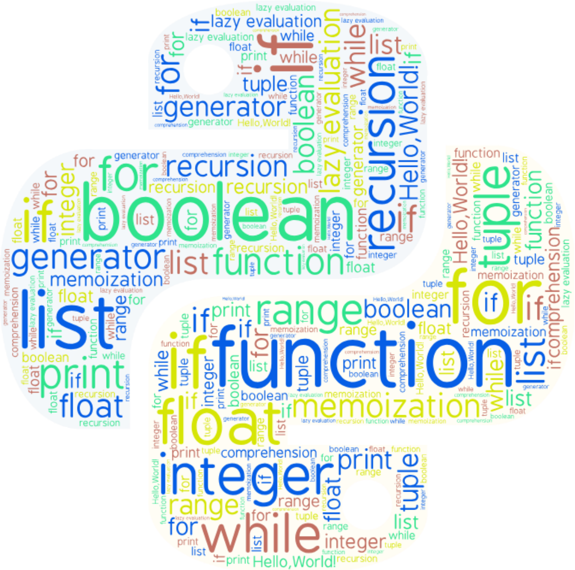
Python 101

Lec04
Useful data structures and modules

thoum

August 20, 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. This does not imply efficiency.

Python provides datatypes and *MANY* useful modules, which this slide is too narrow to contain.

Container Datatypes

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

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

Dictionary

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

[1] Hash Collision

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

```
# 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

```python
avengers = {'natasharomanof': 'blackwidow'} # use curly braces to create a dictionary

# the names are the KEYS, and the hero identity are VALUES
avengers['brucebanner'] = 'hulk'
avengers['tonystark'] = 'ironman'
avengers['peterparker'] = 'spiderman'
avengers['steverogers'] = 'captainamerica'

# asking for permission pattern
if 'tonystark' in avengers:
    print(avengers['tonystark'])

# overwrite key's value
avengers['steverogers'] = 'blueskull'
print(avengers['steverogers'])
```

Using Dictionaries - Cont'

```
# remove the key-value pair
del avengers['tonystark']

# asking for forgiveness pattern
try:
    print(avengers['tonystark'])
except KeyError as e:
    print("No {} in avengers!".format(e))
```

Using Dictionaries

```python
d = {'a':1, 'b':2, 'c':3}
# dictionaries are iterable
print(len(d))

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

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

# get the key:value pair
for k, v in d.items():
    print(k, v)
```

Detour: Error Handling

What could possibly go wrong with the following code?

```
n = int(input())
k = 3/n
```

Detour: Error Handling

Will this do?

```
n = int()
k = 3/n
```

Detour: Error Handling

Errors are inevitable; it is our job to handle them.

```
try: # try some code that can cause errors
    n = int(input())
    k = 3/n
# catch the errors and name them e
except:
    # only executed when there is error
    print("{} happened".format(e))
    k = 1
finally: # executes regardless of errors
    print(k)
```

Detour: Gotta Catch 'em all!

Avoid the Pokémon Gotta catch 'em all! pattern.

```
try:
    n = int(input())
    k = 3/n
except:
    print("some error happened")
```

Upgraded Dictionaries

1. *set*
2. *Counter*

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 methos like union(), intersection(), look here.
Or

```
help(set)
```

Counter

Counts occurrences

```
# load the "collections" module
import collections

words = ['red', 'blue', 'red', 'green', 'blue', 'blue']
# use the Counter class in "collections" module
cnt = collections.Counter(words)

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

Other operations are on the website.

deque: Double Ended Queue

Double Ended Queue

Use instead of *list* when inserting, popping occur at the front 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 right/left.

deque vs list

list:2.09s, deque: 0.009s

```python
# only load deque from collections
from collections import deque
import time

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

start = time.time()
for i in range(100000):
    #insert at front
    l.insert(0, i)
end = time.time()
print("took: ", end-start)

start = time.time()
q = deque() # an empty deque
for i in range(100000):
    #insert at front
    q.appendleft(i)
end = time.time()
print("took: ", end-start)
```

deque vs list

List outperforms deque in random access
list:0.1s, deque: 3.6s

```
from collections import deque
import random
import time

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

start = time.time()
for _ in range(100000):
    l[random.randint(0,999999)]
end = time.time()
print("took: ", end - start)

start = time.time()
for _ in range(100000):
    q[random.randint(0,999999)]
end = time.time()
print("took: ", end - start)
```

Detour: What to use?

Having a slight idea to complexity would help in

1. choosing what to use
2. understanding other's algorithms

Detour: Time Complexity

How long does an operation take?

Detour: Time Complexity

When a size of a problem is N,

1. Some takes constant time
   e.g. lst[i]
2. Some takes time proportional to N
   e.g. [x for x in range(N)]
3. Some takes time proportional to N^2
   e.g. Choosing 2 from N numbers
4. Some takes time proportional to $Nlog(N)$
   e.g. sorting

Detour: Big - O Notation

As N grows large, it will be the dominating factor of an algorithm. Other constants and lower terms can be ignored.
We denote this relation by O notation, describing how running time or space requirements of an algorithm grow as the input size grows.
This helps us analyze running time of algorithms, which would differ among computers with different hardwares.

1. Some takes constant time
   e.g. $O(1)$
2. Some takes time proportional to N
   e.g. $O(N)$
3. Some takes time proportional to N^2
   e.g. $O(N^2)$
4. Some takes time proportional to $Nlog(N)$
   e.g. $O(Nlog(N))$

Detour: Big - O Notation

Shakeela Bibi, Javed Iqbal, Adnan Iftekhar, Mir Hassan- **Analysis of Compression Techniques for DNA Sequence Data**

					technique is difficult to implement. 3.N-grams/2L algorithm off ers space efficiency only on frontend.
[3]	Huffman Coding	Lossless compression of long DNA chain.	B-globin nucleotides of 11 different classes 6-ND6 protein.	O(nlogn)	Only 4 bases of DNA (ACTG) can be map graphically not Protein and RNA.
[4]	1.LZW 2.Gzip 3.Bzip 4.RLE 5.Arithmatic	1.H3N2 has 45 .1% compression ratio. 2.E-coli has 73.1% compression ratio. 3.Gzip gives 4.3% compression ratio for H3N2. 4.Bzip gives 97.23%.	1.H3N2 2.E-coli 3.Becteria 4.Tomato 5.Rabbit	O(n)	1.Run length algorithm is not suitable for DNA data compression.

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

```python
from multiprocessing import Pool


def f(n):
    print(n)
    return n*n

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

with Pool() as p:
    # the order of execution is not guaranteed
    result = p.map(f, l)
print(result)
```

Pool

The order of execution is not guaranteed, although the end result would be ordered.

```
 ~/wor../pyt../lec05 <master> > python3 pool.py
1
2
3
4
5
7
8
6
[1, 4, 9, 16, 25, 36, 49, 64]
 ~/wor../pyt../lec05 <master> > python3 pool.py
1
3
5
4
6
7
2
8
[1, 4, 9, 16, 25, 36, 49, 64]
 ~/wor../pyt../lec05 <master> > python3 pool.py
1
2
3
5
6
7
8
4
[1, 4, 9, 16, 25, 36, 49, 64]
```

Pool Explained

1. Pool(4) means we will use 4 cpus.
2. Pool() defaults to $os.cpu\_count()$.
3. $map(f, l) \rightarrow [f(x) \; for \; x \; in \; l]$

with Explained

with keyword handles initalization and cleanup of certain operations

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

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

Pool Performance

single: 10.5 multi: 3.1

```python
from multiprocessing import Pool
import time

def is_prime(n):
    if n & 1 == 0:
        return True
    d = 3
    while d * d <= n:
        if n % d == 0:
            return True
        d = d + 2
    return False

l = list(range(2000000))

start_time = time.time()
result = [is_prime(n) for n in l]
end_time = time.time()
print("single took: ", end_time-start_time)

start_time = time.time()
with Pool() as p:
    # map(f, l) means apply f to all of l's element
    result = p.map(is_prime, l)
end_time = time.time()

print("multi took: ", end_time-start_time)
```