Python Data Structures

Zbigniew Jędrzejewski-Szmek

Institute of Experimental Physics
University of Warsaw

Python Summer School Kiel, September 4, 2012

Version `AP_version_2011-361-ge89b7ef`

This work is licensed under the *Creative Commons CC BY-SA 3.0 License.*

Outline

1. Introduction
   - Motivation
   - How does Python manage memory?
2. Comprehensions, dictionaries, and sets
   - Comprehensions
   - Dictionaries
   - How to create a dictionary?
   - Specialized dictionary subclasses
   - Sets
3. Iterators
   - Iterator classes
   - Generator expressions
   - Generator functions
4. Closures
5. The end

Introduction

Why?

You can write `sort()`, but it'll take one day.
DRY. Use proper data structures.

The venerable `list`

A sequence of arbitrary values

Question from initial survey:
Is `list.pop(0)` faster than `list.pop()`?

Reminder: computational complexity

$$f(x) = O(g(x))$$

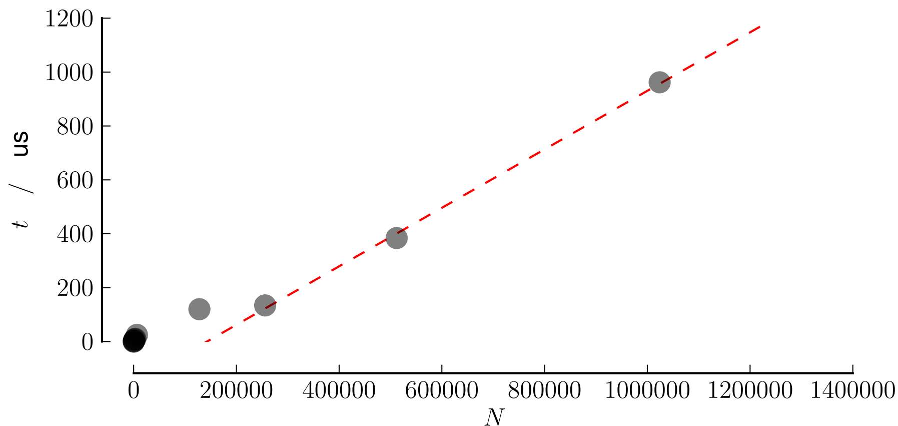
There's a constant C, such that
$f(x) \leqslant C \cdot g(x)$ for big enough x.

For "computational complexity" x is N

Test, test, test

```
N = 1000
L0 = range(N)
L = L0[:]
for i in xrange(N):
    L.pop(0)
```

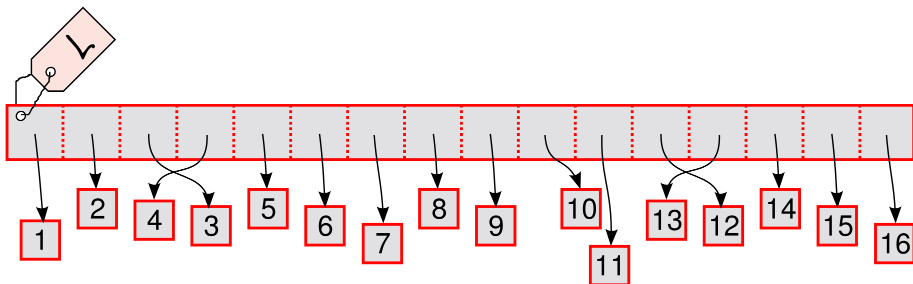
How does time required depend on problem size?



What does this mean?

Computational complexity depends on the data structure and operation — good to understand what you're doing

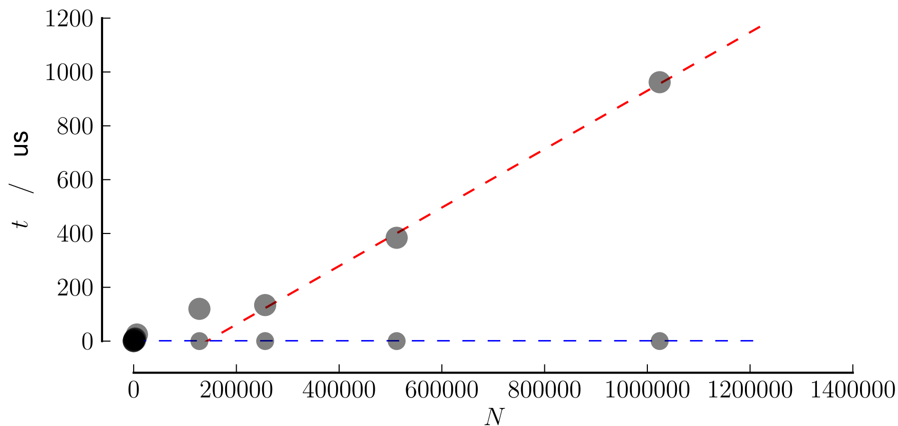
How is `list` implemented?




```
list.pop(0)
```

```
list.pop(-1)
```

`pop(0)` vs `pop(-1)`



Solution

for the `.pop(0)` problem

`collections.deque` pops from both ends cheaply:

- .pop()
- .popleft()

. . . but does not allow other removals in the middle

Trimming from both sides

```
>>> d = collections.deque([1,2,3])
>>> d.pop()
3
>>> d.popleft()
1
>>> d.pop()
2
```

Testing is not enough here

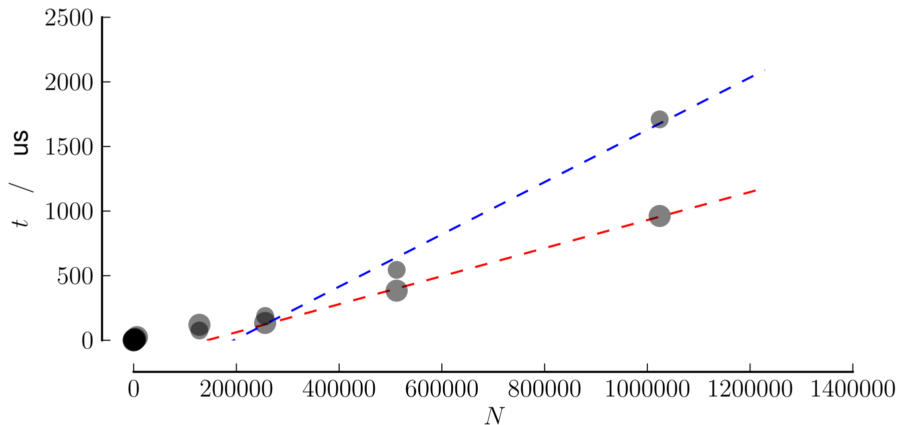
Time complexity issues
do not show up during development,
because $O(N^2)$ is small for small N.

The complexity constant

$$f(x) \leqslant \mathbf{C} \cdot g(x)$$

Complexity constant example

CPython vs. Jython



Other languages have "variables"

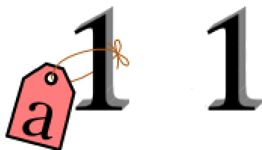
```
int a = 1;
```

```
a = 2;
```

```
int b = a;
```

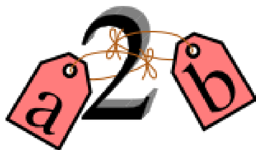
Python has “names”

```
a = 1
```




```
a = 2
```

```
b = a
```



Example

```
>>> L = ['bird'] * 3
>>> L[1] += ' free!'
```

Example

```
>>> L = [['bird']] * 3
>>> L
>>> L[0][0] = 'free'
>>> L
```

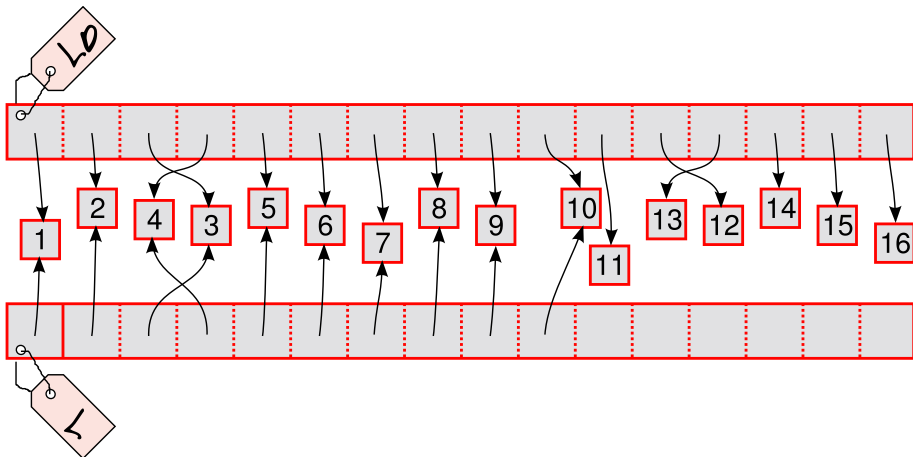
Removing an element from a list

```
N = 1000
L0 = range(N)
L = L0[:]
...
```

Why?

Objects are kept around if they are referenced

`ints` are referenced from both `L` and `L0`



Demo

```
>>> x = 100000
>>> y = 100000
>>> print(id(x), id(y), sep='\n')
```

```
>>> x = 1
>>> y = 1
>>> print(id(x), id(y), sep='\n')
```

Identity and equality

`is` vs `==`

```
if x is None:
    ....
```

```
if x == 11:
    ....
```

Why it's good to use provided data structures

> This describes an adaptive, stable, natural mergesort,
> modestly called timsort (hey, I earned it <wink>).
>
> *Tim Peters*, in `listsort.txt`

Comprehensions, dictionaries, and sets

List comprehensions

```
[day for day in week]

[day for day in week if day[0] != 'M' ]

[(day, hour) for day in week
             for hour in ['12:00', '14:00', '16:00'] ]
```

Creating dictionaries

A mapping of keys to arbitrary values

```
D = {
  'key1': "value1",
  'key2': "value2",
}
```

From an iterator

```
text = """\
lundi     Monday
mardi     Tuesday
mercredi  Wednesday
jeudi     Thursday
vendredi  Friday
samedi    Saturday
dimanche  Sunday
"""
source = (line.split() for line in text.splitlines())
weekdays = dict(source)
```

From a comprehension

```
{x:x**2 for x in xrange(10)}
```

How much does it cost to enter an item in the dictionary?

```
>>> D = {}
>>> L = range(10000)
>>> for i in L:
...     D[i] = i
```

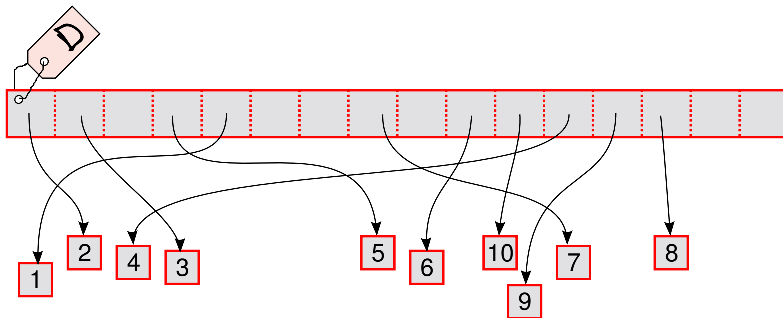
$$O^*(1)$$

How much does it cost to retrieve an item from the dictionary?

```
>>> for i in L:
...     assert D[i] == i
```

$$O(1)$$

How are dictionaries organized?



Weekdays in French and in English

The order of keys is random!

A list of the weekdays?

```
>>> weekdays.keys()
['mardi', 'samedi', 'vendredi', 'jeudi',
 'lundi', 'dimanche', 'mercredi']
```

collections.OrderedDict

```
>>> weekdays = collections.OrderedDict(source)
>>> weekdays.keys()
['lundi', 'mardi', 'mercredi', 'jeudi',
 'vendredi', 'samedi', 'dimanche']
```

Classifying objects

Sorting objects into groups
grades.log:

```
20120101 John 5
20120102 John 4
20120107 Mary 3
20120109 Jane 2
...
```

Sorting objects into groups

version 0

```
grades = {}
for line in open('grades.log'):
    date, person, grade = line.split()
    grades[person].append(grade)
```

Sorting objects into groups

version 0.5

```
grades = {}
for line in open('grades.log'):
    date, person, grade = line.split()
    if person not in grades:
        grades[person] = []
    grades[person].apend(grade)
```

Sorting objects into groups

version 1.0

```
grades = collections.defaultdict(list)
for line in open('grades.log'):
    date, person, grade = line.split()
    grades[person].append(grade)
```

Counting objects in groups

Let's make a histogram

```
grades = collections.Counter()
for line in open('grades.log'):
    date, person, grade = line.split()
    grades[grade] += 1
```

```
>>> grades
Counter({'4': 2, '3': 1, '2': 1, '5': 1})
```

```
>>> print('\n'.join(x + ' ' + 'x'*y
...         for (x, y) in sorted(grades.items())))
1 x
2 xxx
3 x
4 xxx
5 xx
```

Set

A non-ordered collection of unique elements

```
visitors = set()

for connection in connection_log():
    ip = connection.ip
    if ip not in visitors:
        print('new visitor')
        visitors.add(ip)
```

Using sets

Poetry:
find words used by the poet, sorted by **length**

```
>>> poem = '''\
... She walks in beauty, like the night
... Of cloudless climes and starry skies;
... And all that's best of dark and bright
... Meet ...
... '''
```

```
>>> words = set(poem.lower().translate(None, ',;').split())
>>> words
{'she', 'like', 'cloudless', ...}
```

```
>>> sorted(words, key=len, reverse=True)
['cloudless', 'starry', 'beauty', ...]
```

Iterators

Collections and their iterators

- sequence.__iter__() ⟶ iterator
- iterator.next() ⟶ item
- iterator.next() ⟶ item
- iterator.next() ⟶ item

Iterators can be non-destructive or destructive

- `list`
- `file`

How can we create an iterator?

1. write a class with `.__iter__` and `.next`
2. use a generator expression
3. write a generator function

1. Iterator class

```
import random

class randomaccess(object):
    def __init__(self, list):
        self.indices = range(len(list))
        random.shuffle(self.indices)
        self.list = list

    def next(self):
        return self.list[self.indices.pop()]

    def __iter__(self):
        return self
```

2. Generator expressions

```
# chomped lines
[line.rstrip() for line in open(some_file)]
```

```
# remove comments
[line for line in open(some_file)
      if not line.startswith('#')]
```

```
>>> type([i for i in 'abc'])
<type 'list'>
>>> type( i for i in 'abc' )
<type 'generator'>
```

3. Generator functions

```
>>> def gen():
...     print '--start'
...     yield 1
...     print '--middle'
...     yield 2
...     print '--stop'
```

```
>>> g = gen()
>>> g.next()
--start
1
>>> g.next()
--middle
2
>>> g.next()
--stop
Traceback (most recent call last):
  ...
StopIteration
```

Generator objects

```
>>> g = gen()
>>> g
<generator object gen at 0x7f24a5e1c690>
>>> g.<TAB>
g.__iter__(
g.next(
g.send(
g.throw(
g.gi_running
```

Generator objects

__iter__ and next

```
>>> g.__iter__()
<generator object gen at 0x7f24a5e1c690>
>>> g
<generator object gen at 0x7f24a5e1c690>

>>> g.next()
--start
1
```

Generator objects

send, throw, and gi_running

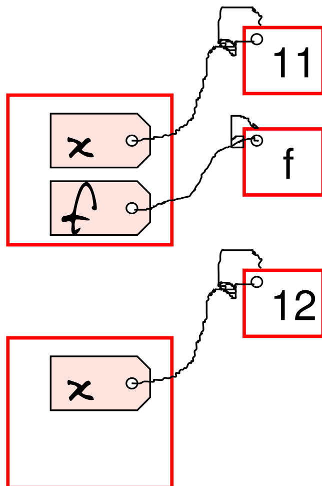
```
>>> help(g.send)
    send(arg) -> send 'arg' into generator,
    return next yielded value or raise StopIteration.
>>> help(g.throw)
    throw(typ[,val[,tb]]) -> raise exception in generator,
    return next yielded value or raise StopIteration.
>>> g.gi_running
0
```

collections.namedtuple

```
>>> import collections
>>> Person = collections.namedtuple('Person',
...                                  'first last SSN')
>>> Person
<class '__main__.Person'>
>>> Person.__doc__
'Person(first, last, SSN)'
>>> p = Person('John', 'Doe', 123456789)
>>> p[0], p[1], p[2]
('John', 'Doe', 123456789)
>>> p.first
'John'
>>> p.last
'Doe'
>>> p.SSN
123456789
```

What namespaces are involved in running a function

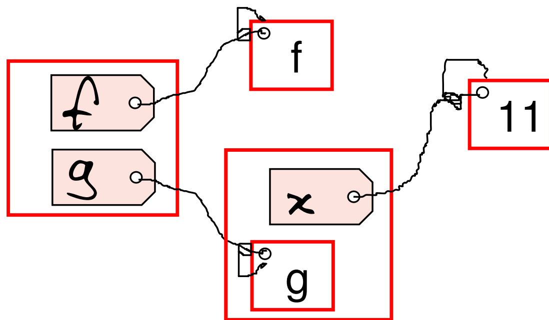
```
>>> x = 11
>>> def f():
...     x = 12
>>> f()
```



Closures

```
>>> def f(x):
...   def g():
...     return x**2
...   return g
>>> g = f(11)
>>> g
<function g at 0x1bea1b8>
>>> g()
121
>>> g.func_closure
(<cell at 0x1b6be50: int object at 0x18da6c8>,)
>>> g.func_closure[0].cell_contents
11
```

Closure visualization



That's all!