{ PYCON KOREA 2014 : 'awesome'}

구종만 jongman@gmail.com

자기소개

- 10년차 개발자 (파이썬은 대략 6년)
- 퀀트 개발자 (≈ 풀타임 파이썬)
- 알고리즘 문제 해결 전략 ('11)
- algospot.com 운영진 ('07~)

ALGOSPOT

오늘의 주제

1. dict 기초 사용
2. dict 동작 원리
3. 이디엄들과 응용 클래스들

이 톡 왜 하나요?

- 이론적인 흥미
  - 이런 유용한 자료 구조를 어떻게 만들었을까?
  - `dict`를 사용하며 발생하는 문제들은 왜 생길까?
- 실용적인 흥미
  - 코드를 쉽게 만들어 줄 수 있는 이디엄과 응용 클래스들

Part I
dict의 기초 사용

dict = 연관 매핑

```
>>> d = {'jan': 1, 'feb': 2, 'mar': 3}
>>> print d['jan'], d['feb'], d['mar']
1 2 3
```

- "사전" 자료 구조
- 키(key)와 값(value) 사이의 관계를 저장한다

만들기

```
# dict 리터럴
>>> {'jan': 1, 'feb': 2, 'mar': 3, ...}
{'apr': 4, 'aug': 8, 'dec': 12, 'feb': 2, ..}
# dict()의 키워드 인자
>>> dict(jan=1, feb=2, mar=3, apr=4, ..)
{'apr': 4, 'aug': 8, 'dec': 12, 'feb': 2, ..}
# (key, value) 쌍의 목록으로 만들기
>>> month_names = [('jan', 1), ('feb', 2), ('mar', 3),
                   ('apr', 4), ('may', 5), ('jun', 6), ..]
>>> dict(month_names)
{'apr': 4, 'aug': 8, 'dec': 12, 'feb': 2, ..}
```

Dictionary Comprehension

```
# 키나 값 변경하기
>>> {name.title(): num for name, num in month_names}
{'Mar': 3, 'Feb': 2, 'Aug': 8, 'Sep': 9, ..}
# 짝수 달만 고르기
>>> {name: num for name, num in month_names if num % 2 == 0}
{'feb': 2, 'aug': 8, 'apr': 4, 'jun': 6, ..}
# dict 뒤집기
>>> months = dict(month_names)
>>> {v: k for k, v in months.items()}
{1: 'jan', 2: 'feb', 3: 'mar', 4: 'apr', ..}
```

탐색

```
>>> months = {'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4, ..}
>>> months['jan']
1
>>> months['Jan']
KeyError: Jan
>>> 'Jan' in months
False
```

get() / setdefault()

```
>>> months.get('Jan')
None
>>> months.get('Jan', 1)
1
>>> months['Jan']
KeyError: Jan
>>> months.setdefault('Jan', 1)
1
>>> months['Jan']
1
```

예제: 길이 분포 구하기

```
>>> words = open('/usr/share/dict/words').read().splitlines()
>>> words
['A', 'a', 'aa', 'aal', 'aalii', 'aam', 'Aani', 'aardvark', ..
```

- 목표: 단어 길이의 분포를 구해 보자
  - 길이 1인 단어: 52개
  - 길이 2인 단어: 160개
  - 길이 3인 단어: 1420개
  - ..

예제: 길이 분포 구하기

```
>>> length_count = {}
>>> for l in map(len, words):
      if l in length_count:
        length_count[l] += 1
      else:
        length_count[l] = 1
```

예제: 길이 분포 구하기

```
>>> length_count = {}
>>> for l in map(len, words):
      length_count[l] = length_count.get(l, 0) + 1
```

예제: 길이별 목록 구하기

- 아예 목록을 구해 보자!

```
>>> by_len = {}
>>> for w in words:
      by_len.get(len(w), []).append(w)
>>> by_len
{}
# ... 어?
```

예제: 길이별 목록 구하기

- setdefault()로 해결!

```
>>> by_len = {}
>>> for w in words:
      by_len.setdefault(len(w), []).append(w)
>>> by_len
{1: ['A', 'a', 'B', 'b', 'C', ...],
 2: ['aa', 'Ab', 'ad', 'ae', 'Ah', ...],
 3: ['aal', 'aam', 'aba', 'abb', 'Abe', ..],
 ..}
```

삭제

```
>>> del months['mar']
>>> 'mar' in months
False
```

순회

```
>>> months.keys()
['mar', 'feb', 'aug', 'sep', ..]
>>> [month for month in months]
['mar', 'feb', 'aug', 'sep', ..]
>>> months.values()
[3, 2, 8, 9, ..]
>>> months.items()
[('mar', 3), ('feb', 2), ('aug', 8), ('sep', 9), ..]
```

- 순서는 엉망진창!
- `iterkeys()`, `itervalues()`, `iteritems()`

Part II
dict 구현하기

요구 조건

- 추가/검색/삭제/순회 연산을 지원할 것
- 가능한한:
  - 빠른 속도
  - 적은 메모리 사용량

속도란 무엇일까?

- dict를 구현한 4개의 서로 다른 클래스가 있다고 하자
- 각 클래스 인스턴스를 생성한 뒤
  - 0~19까지의 키를 넣고
  - 이 중 임의의 한 키를 찾는 연산을 10만번 반복
- 각 클래스마다 평균 소요 시간을 반환한다.

결과

dict 구현	평균시간
A	2ns
B	5ns
C	3ns
D	0.5ns

- D가 가장 좋은 구현처럼 보인다!
- 그러나 진실은 저 멀리에..

방법론의 문제

(엄청나게 여러 가지 문제가 있지만)

- 가장 큰 문제: dict의 크기에 따라 속도는 변화한다!

속도의 2가지 그림자

- `dict`의 크기가 작을 때 얼마나 빠르게 동작하는가?
  - → 프로그램 최적화
  - 중복 연산 제거, 캐시 친화적 자료 구조, ..
- `dict`의 크기가 커질 때 속도가 어떻게 변화하는가?
  - → 자료를 어떠한 형태로 저장하는가?
  - 오늘의 주제

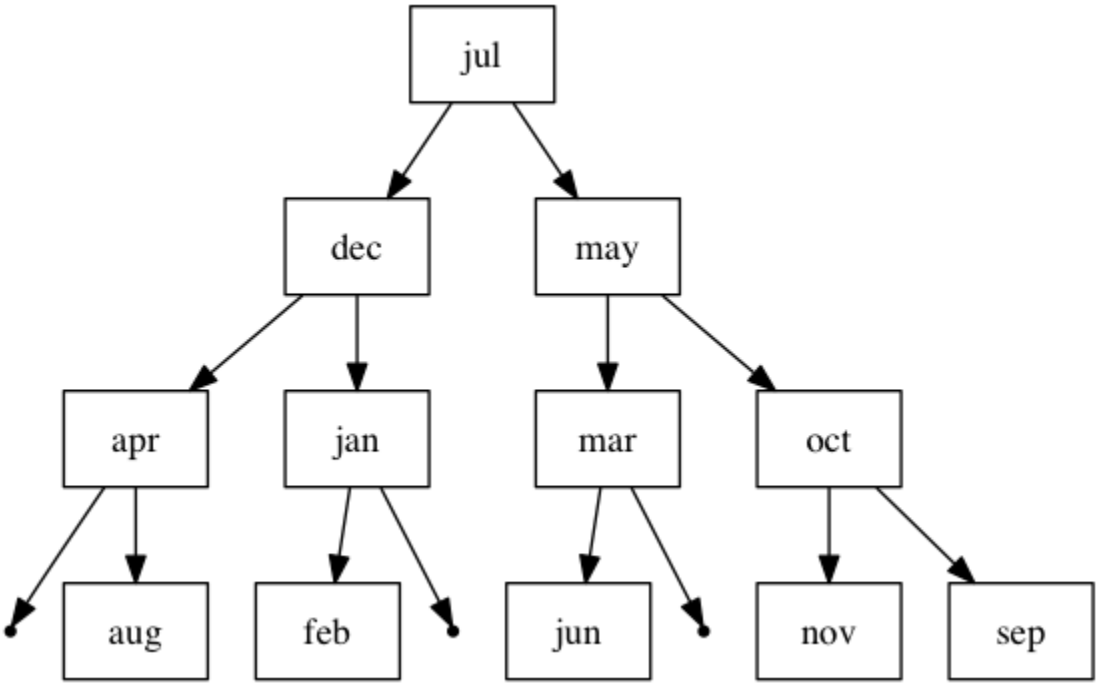
어떻게 자료를 저장하나?

인덱스	키	값
0	'jan'	1
1	'feb'	2
2	'mar'	3
3	'apr'	4
4	'may'	5
5	'jul'	6
6	'jun'	7
7	'aug'	8
..	..	..

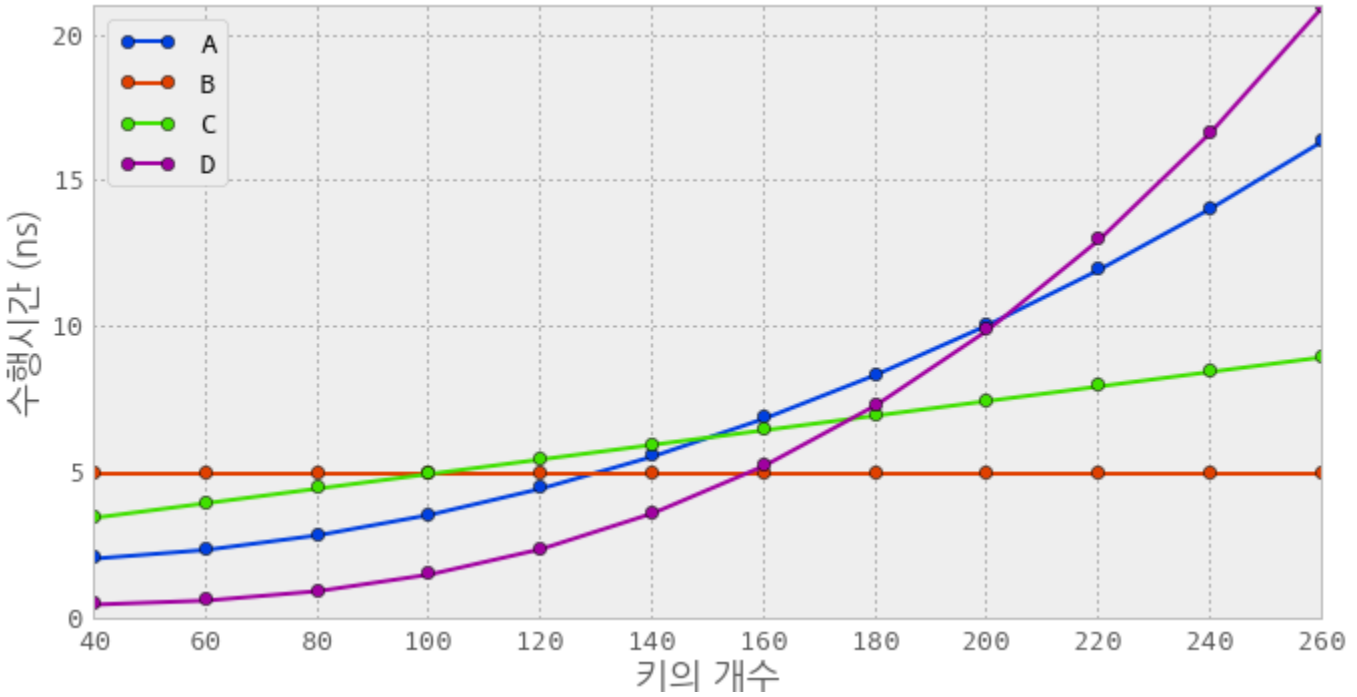
어떻게 자료를 저장하나?

인덱스	키	값
0	'apr'	4
1	'aug'	8
2	'dec'	12
3	'feb'	2
4	'jan'	1
5	'jul'	6
6	'jun'	7
7	'mar'	3
..	..	..

어떻게 자료를 저장하나?



서로 다른 속도 변화


A
B
C
D
수행시간 (ns)
키의 개수
0
5
10
15
20
40
60
80
100
120
140
160
180
200
220
240
260

파이썬의 선택

- 해시테이블(hash table)을 사용
- dict의 크기와 상관없이 일정한 속도를 유지!
- 흔한 사용처에 빠르게 동작하도록 수많은 최적화
  - (CPython 소스코드 참조)

핵심 아이디어:

진짜 핵심 아이디어:

값의 저장 위치가 그 값에 의해 정의된다

해시 테이블

- 일부는 키가 들어 있고, 일부는 **비어 있는** 배열

인덱스	키	값
0	'jan'	1
1	(EMPTY)	(EMPTY)
2	(EMPTY)	(EMPTY)
3	'mar'	3
4	'feb'	2
5	(EMPTY)	(EMPTY)
6	(EMPTY)	(EMPTY)
7	(EMPTY)	(EMPTY)

해시 함수

- 키를 갈아 넣으면 숫자가 나와요

```
>>> hash('jan')
-8681419883224622032
>>> hash('feb')
-4177197833201190620
>>> hash(float)
-9223372036574961420
>>> hash(__builtins__)
-9223372036574926416
```

해시값 → 테이블 내의 위치

```
>>> hash('jan') % 8
0
>>> hash('feb') % 8
4
>>> hash('mar') % 8
3
```

탐색이 간단하다

- 해시값 계산 후 바로 찾음
- 해당 자리가 비어 있으면 해당 키가 없다
- 키 개수와 상관 없이 일정한 속도!

영향: 순서가 엉망진창

- 순회는 키와 상관없이 배열의 0부터 n-1번까지
- 이 중 비어 있지 않은 곳을 반환!

모두 평화로운줄 알았으나

커다란 문제가 있었으니..


충돌!

충돌: 문제

```
>>> hash('jan') % 8
0
>>> hash('apr') % 8
0
```

'apr'은 어디에 넣지?

충돌 해결: 체이닝

- 테이블의 각 자리에 연결 리스트를 넣어둔다

인덱스	(키, 값)
0	[('jan', 1), ('apr', 4)]
1	[]
2	[]
3	[('mar', 3)]
4	[('feb', 2)]
5	[]
6	[]
7	[]

충돌 해결: 체이닝

- 테이블의 각 자리에 연결 리스트를 넣어둔다

인덱스	(키, 값)
0	[('jan', 1), ('apr', 4), ('may', 5)]
1	[]
2	[('nov', 11)]
3	[('mar', 3), ('dec', 12)]
4	[('feb', 2), ('jun', 6)]
5	[('oct', 10)]
6	[('jul', 7), ('aug', 8)]
7	[('sep', 9)]

체이닝: 장단점

- (장점) 간단하다
- (단점) 한 리스트가 커지면 느려진다
  - 대부분 자리에 한두 개의 키만 포함되도록 조절
- (단점) 느리다: 연결 리스트를 유지해야 함
  - 메모리 할당 비용
  - 캐시 로컬리티

충돌 해결: 개방 주소 (CPython)

- 그 다음$^{(*)}$ 자리에 넣는다!

인덱스	키	값
0	'jan'	1
1	'apr'	4
2	(EMPTY)	(EMPTY)
3	'mar'	3
4	'feb'	2
5	(EMPTY)	(EMPTY)
6	(EMPTY)	(EMPTY)
7	(EMPTY)	(EMPTY)

영향: 탐색도 까다로워요

- 다른 키가 들어 있으면 그 다음 자리도 봐야 함
- 원하는 키를 찾거나, 빈 자리가 있어야 결론
- 빈 자리가 적으면 한바퀴를 다 돌아서야 확인 가능
  - 빈 자리의 비율(load factor)이 중요해진다

영향: 삭제가 까다로워요

```
del months['jan']
```

인덱스	키	값
0	(EMPTY)	(EMPTY)
1	'apr'	4
2	(EMPTY)	(EMPTY)
3	'mar'	3
4	'feb'	2
5	(EMPTY)	(EMPTY)
6	(EMPTY)	(EMPTY)
7	(EMPTY)	(EMPTY)

영향: 삭제가 까다로워요

- "변수가 여기 있었다"

인덱스	키	값
0	*dummy*	*dummy*
1	'apr'	4
2	(EMPTY)	(EMPTY)
3	'mar'	3
4	'feb'	2
5	(EMPTY)	(EMPTY)
6	(EMPTY)	(EMPTY)
7	(EMPTY)	(EMPTY)

영향: 순서가 달라요

- 같은 사전도 순서가 달라요!

```
>>> a = {'jan': 1, 'apr': 4}
>>> b = {'apr': 4, 'jan': 1}
>>> print a, b
{'jan': 1, 'apr': 4} {'apr': 4, 'jan': 1}
>>> print a == b
True
>>> print a.items() == b.items()
False
```

충돌 줄이기

- 적절한 리사이즈
- 널뛰는 해시 함수

리사이즈

- 중요성
  - 키가 많아질 수록 충돌이 잦아진다
  - 배열이 꽉 차면? 영원히 빙빙 돌 수도 있다!
- 배열 크기의 2/3보다 키가 많아지면 크기를 늘린다
  - 기존 배열 크기의 2배 혹은 4배

영향: 엄청 빠르다

- 배열의 1/3 이상은 항상 비어 있다
- 만약 우리가 원하는 키가 없다면:
  - 0번 헛수고할 확률: 1/3 = 33%
  - 1번 헛수고할 확률: 2/3 * 1/3 = 22%
  - 2번 헛수고할 확률: 2/3 * 2/3 * 1/3 = 14%
  - ..
  - 10번 헛수고할 확률: 0.5%

영향: 순서가 바뀜

```
a = {8: 'hello', 1: 'world'}
b = {8: 'hello', 1: 'world'}

# 키 몇 개를 주르르 추가: 리사이징이 일어난다!
for i in xrange(5): b[i+100] = 1
# 지워도 이미 늘어난 배열은 아직 줄어들지 않았다
for i in xrange(5): del b[i+100]

print a == b # True
print a.items() == b.items() # False!
```

영향: 순서가 바뀜

A

인덱스	키	값
0	8	'hello'
1	1	'world'
2	(EMPTY)	(EMPTY)
3	(EMPTY)	(EMPTY)
4	(EMPTY)	(EMPTY)
5	(EMPTY)	(EMPTY)
6	(EMPTY)	(EMPTY)
7	(EMPTY)	(EMPTY)

B

인덱스	키	값
0	(EMPTY)	(EMPTY)
1	1	'world'
2	(EMPTY)	(EMPTY)
3	(EMPTY)	(EMPTY)
4	(EMPTY)	(EMPTY)
5	(EMPTY)	(EMPTY)
6	(EMPTY)	(EMPTY)
7	(EMPTY)	(EMPTY)
8	8	'hello'
..	..	..

영향: 순서가 바뀜

- `keys()`와 `values()`의 순서가 달라질 수 있는 이유
- `values()`를 호출하면 새 리스트를 만든다
- 이때 GC 호출이 되면 사전의 크기가 바뀔 수 있음!

영향: 순회 중 업데이트

```
months = {'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4, ..}
for k, v in months.iteritems():
    if v >= 6:
        del months[k]
print months
```

```
RuntimeError: dictionary changed size during iteration
```

영향: 순회 중 업데이트

```
months = {'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4, ..}
for k, v in months.items():
    if v >= 6:
        del months[k]
print months
```

```
{'mar': 3, 'may': 5, 'feb': 2, 'jan': 1, 'apr': 4}
```

영향: 사용자 클래스

```
class User(object):
    def __init__(self, name, email):
        self.name = name
        self.email = email

>>> jongman = User('jongman', 'jongman@gmail.com')
>>> rating = {jongman: 'good'}
>>> print rating[jongman]
good
```

오.. 잘 된다!

영향: 사용자 클래스

과연 그럴까?

```
>>> jongman2 = User('jongman', 'jongman@gmail.com')
>>> print rating[jongman2]
KeyError: <__main__.User object at 0x105e1c910>
```

- 기본적으로는 모든 인스턴스를 다른 키로 인식!
- 해시에 사용할 수 있도록 추가 구현이 필요

영향: 사용자 클래스

- 두 가지 메소드 구현:
  - `__hash__(self)`: 해시값을 반환
  - `__eq__(self, other)`: 두 값이 같은지 반환 (충돌 해결에 사용)
- 어떻게? 멤버 변수들이 모두 같으면(*) 같은 값!

영향: 사용자 클래스

```
class User(object):

    ..

    def members(self):

        return (self.name, self.email)


    def __eq__(self, other):

        return self.members() == other.members()


    def __hash__(self):

        return hash(self.members())
```

결론적으로

- 배열의 크기와 상관 없이 (거의) 상수 시간에 동작하는 자료 구조!

Part 3
응용 클래스

dict 스러운 클래스들

- `collections.OrderedDict`
- `collections.defaultdict`
- `collections.Counter`
- `shelve.Shelf`

OrderedDict

- 삽입한 순서대로 순회가 이루어진다!

```
>>> from collections import OrderedDict
>>> d = OrderedDict()
>>> d['girls'] = 1
>>> d['generation'] = 2
>>> d['gggg'] = 3
>>> d['babybaby'] = 4
>>> d.items()
[('girls', 1), ('generation', 2), ('gggg', 3), ('babybaby', 4)
```

OrderedDict

- 이렇게 하면 실패

```
# 이렇게 하면 한줄에 할 수 있겠지? 히히
>>> d = OrderedDict(girls=1, generation=2, gggg=3,
                    babybaby=4)
# 이런..
>>> d.items()
[('babybaby', 4), ('gggg', 3), ('generation', 2),
 ('girls', 1)]
```

defaultdict

```
>>> from collections import defaultdict
>>> by_len = defaultdict(lambda: [])
>>> for w in words:
      by_len[len(w)].append(w)
>>> by_len
{1: ['A', 'a', 'B', 'b', 'C', ...],
 2: ['aa', 'Ab', 'ad', 'ae', 'Ah', ...], ..}
```

- `setdefault` 하는 것과 같은 효과
- 기본값이 아니라 기본값을 반환하는 함수!

defaultdict

```
>>> from collections import defaultdict
>>> by_len = defaultdict(list)
>>> for w in words:
      by_len[len(w)].append(w)
>>> by_len
{1: ['A', 'a', 'B', 'b', 'C', ...],
 2: ['aa', 'Ab', 'ad', 'ae', 'Ah', ...], ..}
```

- 대부분 타입명도 함수(callable)

nested defaultdicts

각 길이마다 첫 글자별로 모아 보자!

`A[4]['h'] = ['haab', 'haaf', 'habu', 'hack', ..]`

`A[5]['h'] = ['habit', 'hache', 'hacky', 'haddo', ..]`

```
A = defaultdict(lambda: defaultdict(list))
for w in words:
    A[len(w)][w[0]].append(w)
```

infinite defaultdicts

```
>>> infinite_dict = lambda: defaultdict(infinite_dict)
>>> inf = infinite_dict()
>>> inf['Users'][0]['username'] = 'jongman'
>>> inf['Users'][0]['email'] = 'jongman@gmail.com'
>>> inf.items()
[('Users',
defaultdict(<function <lambda> at 0x1025d0938>,
            {0: defaultdict(<function <lambda> at 0x1025d093
                            {'username': 'jongman',
                             'email': 'jongman@gmail.com'})}))
```

Counter

```
>>> from collections import Counter
>>> length_count = Counter(map(len, words))
# dict처럼 원소에 접근
>>> print length_count[1], length_count[2], length_count[3]
52 160 1420
# 없는 키는 0을 반환
>>> print length_count[1237812]
0
# 가장 흔한 길이
>>> print length_count.most_common(3)
[(9, 32403), (10, 30878), (8, 29989)]
```

shelve.Shelf

```
from shelve import open
shelf = open('test') # test.db에 사전 내용을 기록
shelf['hello'] = 1
shelf['world'] = 2
shelf.close()
```

```
shelf = open('test') # 이미 있는 test.db를 읽어옴
print shelf.items()
```

```
[('hello', 1), ('world', 2)]
```

shelve.Shelf: 주의

- 문자열 키만 가능
- `close()`를 빼먹으면 저장이 안됨
  - context manager!

```
from shelve import open
from contextlib import closing
with closing(open('a.shelf')) as shelf:
    shelf['hello'] = 1
    shelf['world'] = 2
```

Resources

- PyCon 2013: Raymond Hettinger
- PyCon 2010: The Mighty Dictionary
- CPython 소스 코드: dictnotes.txt, dictobject.c
- Module of the Week: defaultdict, shelve, OrderedDict, Counter
- Beautiful Code, 18장

감사합니다

Q&A