# prefixtree Documentation

***Release 0.1***

**Aaron Iles**

September 17, 2012

# CONTENTS

*prefixtree* implements Python `dict` and `set` like objects using a trie or prefix tree. Tries are ordered, tree based data structures. Using tries adds unique features to dict and set like objects:

- Keys are returned in sorted order.
- Slice operations for getting, setting and deleting values.

Python's builtin `dict` is implemented using a hash table. While they are an excellent, general purpose container that have been heavily optimised. There are use cases where tree based containers are a better solution.

CHAPTER

ONE

# TABLE OF CONTENTS

## 1.1 Overview

### 1.1.1 Duck Typing

*prefixtree* provides two container classes, `PrefixDict` and `PrefixSet`. They are implementations of the `MutableMapping` and `MutableSet` `Abstract Base Classes`. Any modules that adhere to the principle of *duck-typing* should be able to accept a `PrefixtDict` or `PrefixSet` in place of a `dict` or `set`.

### 1.1.2 Compatability

*prefixtree* is implemented to be compatible with Python 2.x and Python 3.x. It has been tested against the following Python implementations:

- CPython 2.6
- CPython 2.7
- CPython 3.2
- PyPy 1.9.0

Continuous integration testing is provided by Travis CI.

### 1.1.3 Benchmarks

Benchmarks show that *prefixtree* is 200 times slower than the builtin `dict` and requires 10 times the memory.

| Collection | Memory | Run Time |
|---|---|---|
| **dict** | 40MB | 0.34s |
| **PrefixDict** | 453MB | 67s |

The following script was used to benchmark the memory usage and cpu utilisation of `PrefixDict`.

```
"""Test memory consumption and processing time with PrefixDict.

Use words from '/usr/share/dict/words' as keys for PrefixDict and measure
memory consumption and load time.
"""
import resource
import sys
```

```
from prefixtree import PrefixDict


if __name__ == '__main__':
    start = resource.getrusage(resource.RUSAGE_SELF)
    glossary = PrefixDict()
    with open('/usr/share/dict/words') as words:
        for word in words:
            glossary[word.strip()] = None
    stop = resource.getrusage(resource.RUSAGE_SELF)
    rss_mb = (stop.ru_maxrss- start.ru_maxrss) / 1024.0 / 1024.0
    tused = (stop.ru_utime + stop.ru_stime)
    sys.stdout.write('{0} MB\n'.format(rss_mb))
    sys.stdout.write('{0} seconds\n'.format(tused))
```

They are compared to the results for the following script testing the builtin `dict`:

```
"""Test memory consumption and processing time with builtin dict.

Use words from '/usr/share/dict/words' as keys for dict and measure memory
consumption and load time.
"""
import resource
import sys


if __name__ == '__main__':
    start = resource.getrusage(resource.RUSAGE_SELF)
    glossary = {}
    with open('/usr/share/dict/words') as words:
        for word in words:
            glossary[word.strip()] = None
    stop = resource.getrusage(resource.RUSAGE_SELF)
    rss_mb = (stop.ru_maxrss- start.ru_maxrss) / 1024.0 / 1024.0
    tused = (stop.ru_utime + stop.ru_stime)
    sys.stdout.write('{0} MB\n'.format(rss_mb))
    sys.stdout.write('{0} seconds\n'.format(tused))
```

The benchmarks where run using:

- CPython 3.2, 64-bit
- Max OSX 10.7.4
- 2010 Macbook Pro

The benchamrks values were averaged from three runs of each benchmark script.

## 1.2 Guide

### 1.2.1 Introduction

*prefixtree* provides both `PrefixDict`, a *dictionary* like object, and `PrefixSet`, a set like object. Both are implemented using prefix trees, or tries.

#### Tries

Tries, also known as prefix trees, are an ordered tree data structure. Trie minimise the ammount of memory required to store keys if the keys frequently share the same prefix.

In addition to minimising memory, the keys in tries are ordered. This allows prefix tree based dicts and sets to support slicing operations.

---

**Note:** Memory minimisation is an academic property of the data structure. Comparing a pure Python trie to an optimised C hash table may not demonstrate any memory savings.

---

#### Keys

The keys used in *prefixtree* collections must be a `str` or `bytes` object. Unicode strings will be encoded to bytes before storage and after retrieval. Because of this `'\u2641'` and `b'\xe2\x99\x81'` are equivalent keys.

### 1.2.2 Installation

Use pip to install *prefixtree* from PyPI.

```
$ pip install --use-mirrors filemagic
```

The `--use-mirrors` argument is optional. However, it is a good idea to use this option as it both reduces the load on PyPI as well as continues to work if PyPI is unavailable.

The `prefixtree` module should now be availabe from the Python shell.

```
>>> import prefixtree
```

The sections bellow will describe how to use `PrefixDict` and `PrefixSet`.

### 1.2.3 PrefixDict

This dictionary like object, implemented using a trie, is an implementation of the `MutableMapping` `Abstract Base Class`. `PrefixDict` supports the same construction methods as the builtin `dict` object.

```
>>> from prefixtree import PrefixDict
>>> pd = PrefixDict()
>>> pd['a'] = Ellipsis
>>> 'a' in pd
True
```

The most significant difference between `PrefixDict` and builtin `dict` object is that `PrefixDict` supports using a `slice` when getting, setting and deleting keys. When a `slice` is used to get values an *iterator* is returned.

```
>>> pd.update([('a', 0), ('b', 1), ('c', 2)])
>>> list(pd['a':'b'])
[0, 1]
```

Unlike slices for *sequence* objects, such as `list` and `tuple`, slices on `PrefixDict` are inclusive of both the start and the stop values. Step values of 1 and -1 are supported. Indicating forward and reverse iteration.

```
>>> list(pd['a':'c':-1])
[2, 1, 0]
```

When setting a range of values using a slice from a `PrefixDict`, the new values are iterated over in order, replacing the current values from the slice.

```
>>> pd[:'b'] = [3, 4]
>>> pd['a']
3
>>> pd['b']
4
```

If there are fewer new values than there are values in the slice an `ValueError` exception is raised. The exception i raised after updating all possible values from the `PrefixDict`.

```
>>> pd['b':] = [5]
Traceback (most recent call last):
    ...
ValueError: Fewer new elements to than slice length
>>> pd['b']
5
```

Deleting slices works similar to getting slices. They are also inclusive of both the start and the stop value.

```
>>> del pd['b':'b']
>>> 'b' in pd
False
```

In addition to the standard `dict` interface, a `PrefixDict` has the following additional methods.

- `commonprefix()`
- `startswith()`

`commonprefix`‘() returns the longest common prefix between the supplied key and the keys already in the `PrefixDict`.

```
>>> pd.commonprefix('aa')
'a'
```

`startswith()` iterates over all keys that begin with the supplied prefix.

```
>>> pd = PrefixDict(aa=0, ab=1, ac=2)
>>> list(pd.startswith('a'))
['aa', 'ab', 'ac']
```

Matching keys are returned in order. The order can be reversed by passing `True` for the `reverse` parameter.

### 1.2.4 PrefixSet

This set like object, implemented using a trie, is an implementation of the `collections.MutableSet`. `Abstract Base Class`.

```
>>> from prefixtree import PrefixSet
>>> ps = PrefixSet()
>>> ps.add('abc')
>>> 'abc' in ps
True
```

`PrefixSet` supports the same construction methods as the builtin `set` object.

## 1.3 Issues

Source code for *prefixtree* is hosted by GitHub.

Continuous integration testing is hosted by Travis CI.

And bug reports or feature requests can be made using GitHub's issues system.

## 1.4 API

Collection classes implemented using prefix trees.

### 1.4.1 PrefixDict

**class** `prefixtree.`**`PrefixDict`**`(`[*arg*]`)`
Implementation of `MutableMapping` that conforms to the interface of `dict`.

A *PrefixDict* has some extensions to the interface of `dict`.

**`d[i:j:k]`**
Return iterable from *i* to *j* in the direction *k*.

If *i* or *j* is *None* the slice is open ended. *k* may be -1 or 1. If -1 the values are returned in reverse order.

**`commonprefix`**`(`*key*`)`
Find the longest common prefix between the key provided and the keys in the *PrefixDict*.

**`startswith`**`(`*prefix*`)`
Iterate over all keys that begin with the supplied prefix.

### 1.4.2 PrefixSet

**class** `prefixtree.`**`PrefixSet`**`(`[*iterable*]`)`
Implementation of `MutableSet` that conforms to the interface of `set`.