

Introduction to Python

Based on tutorial by Liang Huang

Zen of Python

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.



About Python

- Named after Monty Python
- Invented by Guido van Rossum, Benevolent Dictator for Life (BDFL)
- Versions
 - 2.7.x is the end of the 2.x series (this tutorial)
 - 3.x is incompatible and not yet widely adopted
- For more information: <http://www.python.org>
 - Tutorial: <http://docs.python.org/tutorial/>
 - Reference: <http://docs.python.org/reference/>
 - Library: <http://docs.python.org/library/>

About Python



- Named after Monty Python
- Invented by Guido van Rossum, Benevolent Dictator for Life (BDFL)
- Versions
 - 2.7.x is the end of the 2.x series (this tutorial)
 - 3.x is incompatible and not yet widely adopted
- For more information: <http://www.python.org>
 - Tutorial: <http://docs.python.org/tutorial/>
 - Reference: <http://docs.python.org/reference/>
 - Library: <http://docs.python.org/library/>

Hello, world.

Java	Python
<pre>public class Hello { public static void main (String argv[]) { System.out.println("Hello, world."); } }</pre>	<pre>print("Hello, world.")</pre>

Hello, world.

C

```
#include <stdio.h>
int argc (int argc, char *argv[]) {
    printf("Hello, world.\n");
}
```

Python

```
print("Hello, world.")
```

Hello, world.

Perl	Python
<pre>print("Hello, world.\n");</pre>	<pre>print("Hello, world.")</pre>

Key Word In Context

Beautiful

~ is better

Complex

~ is better

Explicit

~ is better

Flat

~ is better

Simple

~ is better

Sparse

~ is better

better

Beautiful is ~ than ugly.

Explicit is ~ than implicit.

Simple is ~ than complex.

Complex is ~ than

complicated.

Flat is ~ than nested.

Sparse is ~ than dense.

complex.

better than ~

complicated.

better than ~

dense.

better than ~

implicit.

better than ~

is

Beautiful ~ better than

Explicit ~ better than

Simple ~ better than

Complex ~ better than

Flat ~ better than

Sparse ~ better than

nested.

better than ~

than

is better ~ ugly.

is better ~ implicit.

is better ~

complex.

is better ~

complicated.

is better ~

nested.

is better ~ dense.

ugly.

better than ~

Perl

```
$size = 2;
while (<>) {
    @words = split;
    for ($i=0; $i<@words; $i++) {
        $start = $i-$size;
        $start = 0 if $start < 0;
        $before = [@words[$start..$i-1]];
        $stop = $i+$size;
        $stop = $#words if $stop > $#words;
        $after = [@words[$i+1..$stop]];
        push(@{$index{$words[$i]}},
            [$before,$after]);
    }
}
for $word (sort (keys %index)) {
    print $word, "\n";
    for $context (@{$index{$word}}) {
        ($before, $after) = @$context;
        print "  ", join(" ", @$before),
            " ~ ", join(" ", @$after), "\n";
    }
}
```

Python

```
import fileinput, collections
size = 2
index = collections.defaultdict(list)
for line in fileinput.input():
    words = line.split()
    for i, word in enumerate(words):
        start = max(i-size, 0)
        before = words[start:i]
        after = words[i+1:i+size+1]
        index[word].append((before,after))
for word in sorted(index):
    print word
    for before, after in index[word]:
        print("  " + " ".join(before) +
            " ~ " + " ".join(after))
```

Java

```
import java.io.*;
import java.util.*;
class Context { public String[] before, after; }
public class kwic {
    public static int size = 2;
    public static String join(String strings[]) {
        StringBuilder sb = new StringBuilder();
        Boolean first = true;
        for (String s : strings) {
            if (!first) sb.append(" ");
            sb.append(s);
            first = false;
        }
        return sb.toString();
    }
    public static void main(String args[]) {
        try {
            TreeMap<String,List<Context>> index = new TreeMap<String,List<Context>>();
            BufferedReader in = new BufferedReader(new FileReader(args[0]));
            String line;
            while ((line = in.readLine()) != null) {
                String[] words = line.trim().split("\\s+");
                for (int i=0; i<words.length; i++) {
                    if (!index.containsKey(words[i]))
                        index.put(words[i], new ArrayList<Context>());
                    Context context = new Context();
                    int start = Math.max(0, i-size),
                        stop = Math.min(words.length, i+size+1);
                    context.before = Arrays.copyOfRange(words, start, i);
                    context.after = Arrays.copyOfRange(words, i+1, stop);
                    index.get(words[i]).add(context);
                }
            }
            in.close();
            for (String word : index.keySet()) {
                System.out.println(word);
                for (Context context : index.get(word)) {
                    System.out.println(" " + join(context.before) +
                                         " ~ " + join(context.after));
                }
            }
        } catch (IOException e) {
        }
    }
}
```

Python

```
import fileinput, collections
size = 2
index = collections.defaultdict(list)
for line in fileinput.input():
    words = line.split()
    for i, word in enumerate(words):
        start = max(i-size, 0)
        before = words[start:i]
        after = words[i+1:i+size+1]
        index[word].append((before,after))
for word in sorted(index):
    print word
    for before, after in index[word]:
        print(" " + " ".join(before) +
              " ~ " + " ".join(after))
```

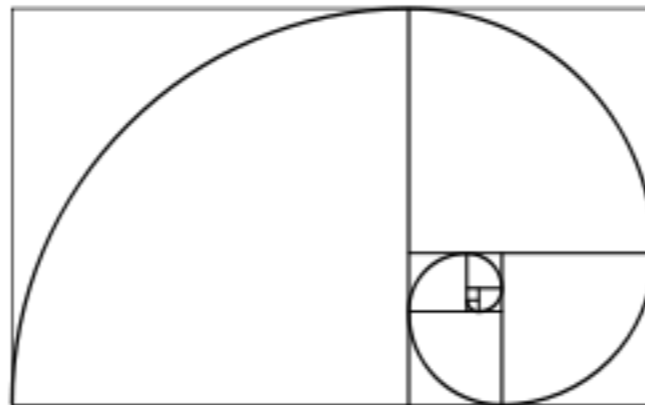
Fibonacci: It's as easy as 1, 1, 2, 3

$$x_1 = 1$$

$$x_2 = 1$$

$$x_n = x_{n-1} + x_{n-2}$$

$$\lim_{n \rightarrow \infty} \frac{x_{n+1}}{x_n} = \frac{1 + \sqrt{5}}{2} = 1.6180339887 \dots$$



Fibonacci: It's as easy as 1, 1, 2, 3

```
a = b = 1
```

```
i = 0
```

```
while i < 10:
```

```
    print(a)
```

```
    c = a + b
```

```
    a = b
```

```
    b = c
```

```
    i += 1
```

```
print("done")
```

variables do not need to be declared

block always introduced by colon (:)

block *must* be indented (usually 4 spaces)

equivalent to `i = i + 1` but `i++` is right out

Variables

```
>>> a
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
NameError: name 'a' is not
defined
>>> a = 1
>>> a
1
>>> type(a)
<type 'int'>
>>> a = "foo"
>>> type(a)
<type 'str'>
```

```
>>> del a
>>> a
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
NameError: name 'a' is not
defined
>>> type(a)
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
NameError: name 'a' is not
defined
```

Fibonacci: It's as easy as 1, 1, 2, 3

```
a = b = 1           1
i = 0              2
while i < 10:      1
    print(b/a)     1
    c = a + b      1
    a = b          1
    b = c          1
    i += 1         1
                  1
                  1
```

Fibonacci: It's as easy as 1, 1, 2, 3

<code>a = b = 1</code>	1.0
<code>i = 0</code>	2.0
<code>while i < 10:</code>	1.5
<code>print(float(b)/a)</code>	1.666666666667
<code>c = a + b</code>	1.6
<code>a = b</code>	1.625
<code>b = c</code>	1.61538461538
<code>i += 1</code>	1.61904761905
	1.61764705882
	1.61818181818

Fibonacci: It's as easy as 1, 1, 2, 3

```
from __future__ import division 1.0
a = b = 1                        2.0
i = 0                            1.5
while i < 10:                    1.666666666667
    print(b/a)                   1.6
    c = a + b                    1.625
    a = b                        1.61538461538
    b = c                        1.61904761905
    i += 1                       1.61764705882
                                1.61818181818
```

Functions

```
def fib(n):  
    a = b = 1  
    i = 0  
    while i < n:  
        print(a)  
        c = a + b  
        a = b  
        b = c  
        i += 1
```

```
>>> fib(6)  
1  
1  
2  
3  
5  
8  
>>> i  
Traceback (most recent call  
last):  
  File "<stdin>", line 1, in  
<module>  
NameError: name 'i' is not  
defined
```

Functions

```
def fib(n):  
    global i           illustration only!  
    a = b = 1  
    i = 0  
    while i < n:  
        print(a)  
        c = a + b  
        a = b  
        b = c  
        i += 1
```

```
>>> fib(6)  
1  
1  
2  
3  
5  
8  
>>> i  
6
```

Lists

```
def fib(n):
```

```
    a = b = 1
```

```
    i = 0
```

```
    while i < n:
```

```
        print(a)
```

```
        c = a + b
```

```
        a = b
```

```
        b = c
```

```
        i += 1
```

```
def fib(n):
```

```
    a = b = 1
```

```
    for i in range(n):
```

```
        print(a)
```

```
        c = a + b
```

```
        a = b
```

```
        b = c
```

Lists

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> ws = [1, 2]
>>> len(ws)
2
>>> ws[0]
1
>>> ws.append("red")
>>> ws += ["blue"]
>>> ws
[1, 2, 'red', 'blue']
```

Lists

```
def fib(n):  
  
    a = b = 1  
  
    for i in range(n):  
        print(a)  
        c = a + b  
        a = b  
        b = c
```

```
def fib(n):  
  
    a = b = 1  
    xs = []  
    for i in range(n):  
        xs.append(a)  
        c = a + b  
        a = b  
        b = c  
    return xs
```

```
>>> fib(6)  
[1, 1, 2, 3, 5, 8]
```

Tuples

```
def fib(n):  
    a = b = 1  
    xs = []  
    for i in range(n):  
        xs.append(a)  
        c = a + b  
        a = b  
        b = c  
    return xs
```

```
def fib(n):  
    a = b = 1  
    xs = []  
    for i in range(n):  
        xs.append(a)  
        a, b = b, a + b  
    return xs
```

Tuples

```
>>> point = (3, 4)
>>> (x, y) = point
>>> x
3
>>> y
4
>>> (x, y) = (y, x)
>>> x
4
>>> y
3
>>> def hypot((x,y)):
...     return (x**2+y**2)**0.5
>>> hypot(point)
5.0
```

```
>>> points = [(3, 4), (5, 12)]
>>> for (x,y) in points:
...     print x
3
5
>>> (1)
1
>>> (1,)
(1,)
>>> point = 1, 2      parens optional
>>> point == (1, 2)
True
>>> point == 1, 2
(False, 2)
```

Iterators

```
def fib(n):  
    a = b = 1  
    xs = []  
    for i in range(n):           inefficient  
        xs.append(a)  
        a, b = b, a + b  
    return xs
```

Iterators

```
def fib(n):  
    a = b = 1  
    xs = []  
    for i in xrange(n):  
        xs.append(a)  
        a, b = b, a + b  
    return xs
```

Iterators

```
>>> xrange(10)
xrange(10)
>>> for i in xrange(10):
...     print i
0
1
2
3
4
5
6
7
8
9
```

Iterators

```
def fib(n):  
    a = b = 1  
    xs = []  
    for i in xrange(n):  
        xs.append(a)  
        a, b = b, a + b  
    return xs
```

```
def fib(n):  
    a = b = 1  
  
    for i in xrange(n):  
        yield a  
        a, b = b, a + b
```

Iterators

```
>>> fib(10)
<generator object fib at ...>
>>> for x in fib(10):
...     print x
...
1
1
2
3
5
8
13
21
34
55
```

```
>>> f = fib(5)
>>> for x in f:
...     print x
...
1
1
2
3
5
>>> for x in f:
...     print x
...
>>>
```

A word puzzle

Which English word contains the most other distinct English words?

banana

a

ba

an

na

ban

ana

nan

nana

A word puzzle

```
words = []
for line in open("words"):
    words.append(line.rstrip())
table = []
for word in words:
    n = len(word)
    subwords = []
    for i in xrange(n):
        for j in xrange(i+1, n+1):
            subword = word[i:j]
            if (subword in words and
                subword not in subwords):
                subwords.append(subword)
    table.append((len(subwords), word))
count, word = max(table)
print("{0} has {1} subwords".format(word, count))
```

iterator over lines
remove trailing whitespace

substring from *i* to *j*-1
list membership

maximize count, then word
count))

Strings

```
>>> "foo"
'foo'
>>> 'bar'
'bar'
>>> len("foo")
3
>>> "foo" + "bar"
'foobar'
>>> "foo" * 3
'foofoofoo'
```

```
>>> s1 = s2 = "foo"
>>> s2 += "bar"
>>> s1
'foo'
>>> s2
'foobar'
```

```
>>> s = "banana"
>>> s[0]
'b'
>>> s[0:2]
'ba'
>>> s[:2]
'ba'
>>> s[4:]
'na'
>>> s[-1]
'a'
>>> s[-2:]
'na'
```

Strings

```
>>> s = "Colorless green ideas sleep furiously\n"
>>> s.rstrip()
'Colorless green ideas sleep furiously'
>>> w = s.split()
>>> w
['Colorless', 'green', 'ideas', 'sleep', 'furiously']
>>> " ".join(w)
'Colorless green ideas sleep furiously'
>>> s = "{0} green ideas sleep {1}"
>>> s.format(3, 'wildly')      Python 2.6 and later
'3 green ideas sleep wildly'
>>> s = "Colorless {adj} ideas {verb} furiously"
>>> s.format(adj='yellow', verb='recline')
'Colorless yellow ideas recline furiously'
```

A word puzzle

```
words = []
for line in file("words"):
    words.append(line.rstrip())
table = []
for word in words:
    n = len(word)
    subwords = []
    for i in xrange(n):
        for j in xrange(i+1, n+1):
            subword = word[i:j]
            if (subword in words and                inefficient
                subword not in subwords):
                subwords.append(subword)
    table.append((len(subwords), word))
count, word = max(table)
print("{0} has {1} subwords".format(word, count))
```

A word puzzle

```
words = {}
for line in file("words"):
    words[line.rstrip()] = True
table = []
for word in words:
    n = len(word)
    subwords = {}
    for i in xrange(n):
        for j in xrange(i+1, n+1):
            subword = word[i:j]
            if subword in words:
                subwords[subword] = True
    table.append((len(subwords), word))
count, word = max(table)
print("{0} has {1} subwords".format(word, count))
```

Dictionaries (a.k.a. maps, hash tables)

```
>>> d = {1:"January", "Jan":"January",
...      2:"February", "Feb":"February",
...      3:"March", "Mar":"March"}
>>> d["Feb"]
'February'
>>> 3 in d
True
>>> for k in d:
...     print(d[k])
January
February
March
February
January
March
```

A word puzzle

```
words = set()
for line in file("words"):
    words.add(line.rstrip())
table = []
for word in words:
    n = len(word)
    subwords = set()
    for i in xrange(n):
        for j in xrange(i+1, n+1):
            subword = word[i:j]
            if subword in words:
                subwords.add(subword)
    table.append((len(subwords), word))
count, word = max(table)
print("{0} has {1} subwords".format(word, count))
```

Sets

```
>>> set([1, 1, 2, 3])
set([1, 2, 3])
>>> {1, 1, 2, 3}          Python 2.7
set([1, 2, 3])
>>> {1, 4, 9, 16} | {1, 2, 4, 8, 16}
set([1, 2, 4, 8, 9, 16])
>>> {1, 4, 9, 16} & {1, 2, 4, 8, 16}
set([16, 1, 4])
```

A word puzzle

```
words = {line.rstrip() for line in file("words")}
```

```
table = []
for word in words:
    n = len(word)
    subwords = set()
    for i in xrange(n):
        for j in xrange(i+1, n+1):
            subword = word[i:j]
            if subword in words:
                subwords.add(subword)
    table.append((len(subwords), word))
count, word = max(table)
print("{0} has {1} subwords".format(word, count))
```

Comprehensions

```
>>> ["banana"[i:i+2] for i in xrange(5)]  
['ba', 'an', 'na', 'an', 'na']
```

```
>>> i
```

```
4
```

note `i` is leaked

```
>>> list("banana"[i:i+2] for i in xrange(5))  
['ba', 'an', 'na', 'an', 'na']
```

```
>>> set("banana"[i:i+2] for i in xrange(5))  
set(['na', 'ba', 'an'])
```

```
>>> {"banana"[i:i+2] for i in xrange(5)}  
set(['na', 'ba', 'an'])
```

Python 2.7

```
>>> {"banana"[i:j] for i in xrange(6) for j in xrange(i+1,7)}  
set(['a', 'b', 'ba', 'nana', 'na', 'nan', 'an', 'anana', 'anan',  
'n', 'bana', 'ban', 'banan', 'banana', 'ana'])
```

A word puzzle

```
words = {line.rstrip() for line in file("words")}
table = []
for word in words:
    n = len(word)
    subwords = set()
    for i in xrange(n):
        for j in xrange(i+1, n+1):
            subword = word[i:j]
            if subword in words:
                subwords.add(subword)
    table.append((len(subwords), word))
count, word = max(table)
print("{0} has {1} subwords".format(word, count))
```

A word puzzle

pseudolamellibranchiate

o a i
do am li an hi
la el ran at
udo me lib chi
dol mel bra chia
lam bran ate
ame ranch
ell branch
lame libra
pseudo branchia
branchiate
lamellibranch
lamellibranchiate

Object-oriented programming in Python

- Philosophy
 - Most similar to Smalltalk, but even more free-form
 - *Much* more free-form than C++ or Java
- Everything is an object
 - even atomic types `int`, `str`, etc.
 - even functions
- But you don't have to use OOP when you don't want to

The simplest class

```
>>> class Point(object):  
...     pass                                just a no-op statement  
...  
>>> p = Point()  
>>> p  
<__main__.Point object at 0x1004cc810>  
>>> p.x = 1  
>>> p.y = 2  
>>> p.x  
1
```

Methods

```
>>> class Point(object):
...     def hypot(self):
...         return (self.x**2+self.y**2)**0.5
...
>>> p1 = Point()
>>> p1.x, p1.y = 3, 4
>>> p2 = Point()
>>> p2.x, p2.y = 5, 12
>>> p1.hypot()
5.0
>>> p2.hypot()
13.0
>>> Point.hypot(p1)
5.0
```

`self` is a reference to the object. (It doesn't actually have to be called `self`.)

Constructors

```
>>> import math
>>> class Point(object):
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def hypot(self):
...         return (self.x**2+self.y**2)**0.5
...     def angle(self):
...         return math.atan(float(self.y)/self.x)
>>> p1 = Point(4, 3)
>>> p1.hypot()
5.0
>>> p1.angle()
0.64350110879328437
```

Inheritance

```
>>> import math
>>> class Polar(Point):
...     def __init__(self, r, theta):
...         x = r*math.cos(theta)
...         y = r*math.sin(theta)
...         Point.__init__(self, x, y)
...
>>> p1 = Polar(5, 1.23)
>>> p1.hypot()
5.0
>>> p1.angle()
1.23
```

There is a **super** keyword, but it's ugly

Special methods

```
class Point(object):
    :
    def __str__(self):
        return "Point({0},{1})".format(str(self.x), str(self.y))

>>> print(Point(3,4))
Point(3,4)
>>> print([Point(3,4)])
[<__main__.Point object at 0x1004cc710>]
```

Special methods

```
class Point(object):
    :
    def __str__(self):
        return "Point({0},{1})".format(str(self.x), str(self.y))
    def __repr__(self):
        return "Point({0},{1})".format(repr(self.x), repr(self.y))

>>> print(Point(3,4))
Point(3,4)
>>> print([Point(3,4)])
[Point(3,4)]
```

Special methods

```
class Point(object):  
    :  
    def __eq__(self, other):  
        return (isinstance(other, Point) and  
                (self.x, self.y) == (other.x, other.y))
```

```
>>> p1 = Point(3,4)
```

```
>>> p2 = Point(3,4)
```

```
>>> p1 == p2
```

```
True
```

```
>>> p1 is p2
```

```
False
```

Special methods

```
class Point(object):
    :
    def __eq__(self, other):
        return (isinstance(other, Point) and
                (self.x, self.y) == (other.x, other.y))

    def __hash__(self):
        return hash((self.x, self.y))

>>> points = {}
>>> points[Point(3,4)] = 5
>>> points[Point(3,4)]
5
```

Special methods

```
class Point(object):
    :
    def __add__(self, other):
        return Point(self.x+other.x, self.y+other.y)

>>> Point(3,4)+Point(5,12)
Point(8,16)
```

Directed graphs

```
class Node(Point):  
    def __init__(self, x, y, neighbors=[]): default argument  
        Point.__init__(self, x, y)  
        self.neighbors = neighbors
```

```
>>> a = Node(3, 4)  
>>> b = Node(5, 12)  
>>> c = Node(7, 24, [a, b])  
>>> a.neighbors += [b]  
>>> b.neighbors += [c]  
>>> a.neighbors  
[Point(5,12), Point(7,24)]
```

uh-oh

Directed graphs

```
class Node(Point):
    def __init__(self, x, y, neighbors=[]): default argument
        Point.__init__(self, x, y)
        self.neighbors = neighbors
```

```
>>> a = Node(3, 4)
>>> b = Node(5, 12)
>>> c = Node(7, 24, [a, b])
>>> a.neighbors += [b]
>>> b.neighbors += [c]
>>> a.neighbors
[Point(5,12), Point(7,24)]
>>> a.neighbors is b.neighbors
True
```

uh-oh

Directed graphs

```
class Node(Point):
    def __init__(self, x, y, neighbors=None): default argument
        Point.__init__(self, x, y)
        self.neighbors = neighbors or []

>>> a = Node(3, 4)
>>> b = Node(5, 12)
>>> c = Node(7, 24, [a, b])
>>> a.neighbors += [b]
>>> b.neighbors += [c]
>>> a.neighbors
[Point(5,12)]
```

Directed graphs

```
class Node(Point):
    :
    def dfs(self):
        nodes = []
        for neighbor in self.neighbors:
            nodes.extend(neighbor.dfs())
        return nodes

>>> a = Node(3, 4)
>>> b = Node(5, 12)
>>> c = Node(7, 24, [a, b])
>>> a.neighbors += [b]
>>> b.neighbors += [c]
>>> a.dfs()
```

Directed graphs

```
class Node(Point):
    :
    def dfs(self, memo=None):
        if memo is None: memo = set()
        memo.add(self)
        nodes = [self]
        for neighbor in self.neighbors:
            if neighbor not in memo:
                nodes.extend(memo=neighbor.dfs(memo))
        return nodes

    :
>>> a.dfs()
[Point(3,4), Point(5,12), Point(7,24)]
```

Key Word In Context

Beautiful

~ is better

Complex

~ is better

Explicit

~ is better

Flat

~ is better

Simple

~ is better

Sparse

~ is better

better

Beautiful is ~ than ugly.

Explicit is ~ than implicit.

Simple is ~ than complex.

Complex is ~ than

complicated.

Flat is ~ than nested.

Sparse is ~ than dense.

complex.

better than ~

complicated.

better than ~

dense.

better than ~

implicit.

better than ~

is

Beautiful ~ better than

Explicit ~ better than

Simple ~ better than

Complex ~ better than

Flat ~ better than

Sparse ~ better than

nested.

better than ~

than

is better ~ ugly.

is better ~ implicit.

is better ~

complex.

is better ~

complicated.

is better ~

nested.

is better ~ dense.

ugly.

better than ~

Key Word In Context

```
size = 2
index = {}
for line in open("zen"):
    words = line.split()
    for wi in xrange(len(words)):
        word = words[wi]
        start = max(wi-size, 0)
        stop = min(wi+size+1, len(words))
        before = words[start:wi]
        after = words[wi+1:stop]
        if word not in index:
            index[word] = []
        index[word].append((before, after))
```

Key Word In Context

```
import fileinput
size = 2
index = {}
for line in fileinput.input():
    words = line.split()
    for wi in xrange(len(words)):
        word = words[wi]
        start = max(wi-size, 0)
        stop = min(wi+size+1, len(words))
        before = words[start:wi]
        after = words[wi+1:stop]
        if word not in index:
            index[word] = []
        index[word].append((before, after))
```

Key Word In Context

```
import fileinput
size = 2
index = {}
for line in fileinput.input():
    words = line.split()
    for wi, word in enumerate(words):

        start = max(wi-size, 0)
        stop = min(wi+size+1, len(words))
        before = words[start:wi]
        after = words[wi+1:stop]
        if word not in index:
            index[word] = []
        index[word].append((before, after))
```

Key Word In Context

```
import fileinput, collections
size = 2
index = collections.defaultdict(list)
for line in fileinput.input():
    words = line.split()
    for wi, word in enumerate(words):

        start = max(wi-size, 0)
        stop = min(wi+size+1, len(words))
        before = words[start:wi]
        after = words[wi+1:stop]

        index[word].append((before, after))
```

Key Word In Context

```
keys = index.keys()
keys.sort()
for word in keys:
    print(word)
    for context in index[word]:
        before, after = context
        print(" " + " ".join(before) +
              " ~ " + " ".join(after))
```

Key Word In Context

```
for word in sorted(index):
    print(word)
    for context in index[word]:
        before, after = context
        print(" " + " ".join(before) +
              " ~ " + " ".join(after))
```

Key Word In Context

```
for word in sorted(index):
    print(word)
    for before, after in index[word]:
        print(" " + " ".join(before) +
              " ~ " + " ".join(after))
```

Key Word In Context

```
import fileinput, collections
size = 2
index = collections.defaultdict(list)
for line in fileinput.input():
    words = line.split()
    for wi, word in enumerate(words):
        start = max(wi-size, 0)
        stop = min(wi+size+1, len(words))
        before = words[start:wi]
        after = words[wi+1:stop]
        index[word].append((before, after))
for word in sorted(index):
    print(word)
    for before, after in index[word]:
        print("  " + " ".join(before) +
              " ~ " + " ".join(after))
```