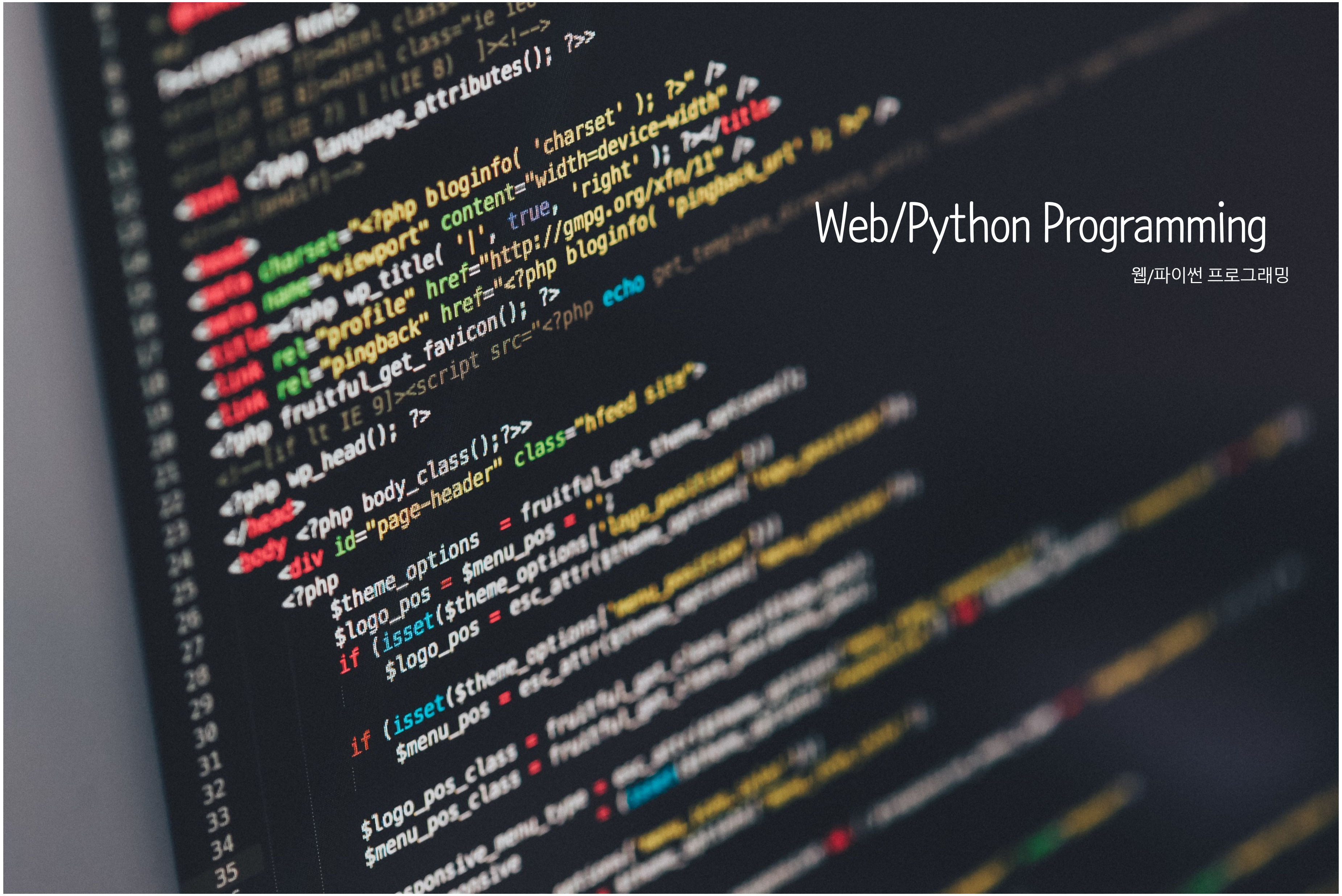


Web/Python Programming

웹/파이썬 프로그래밍

Today

- Searching
  - Searching a list
  - Binary search

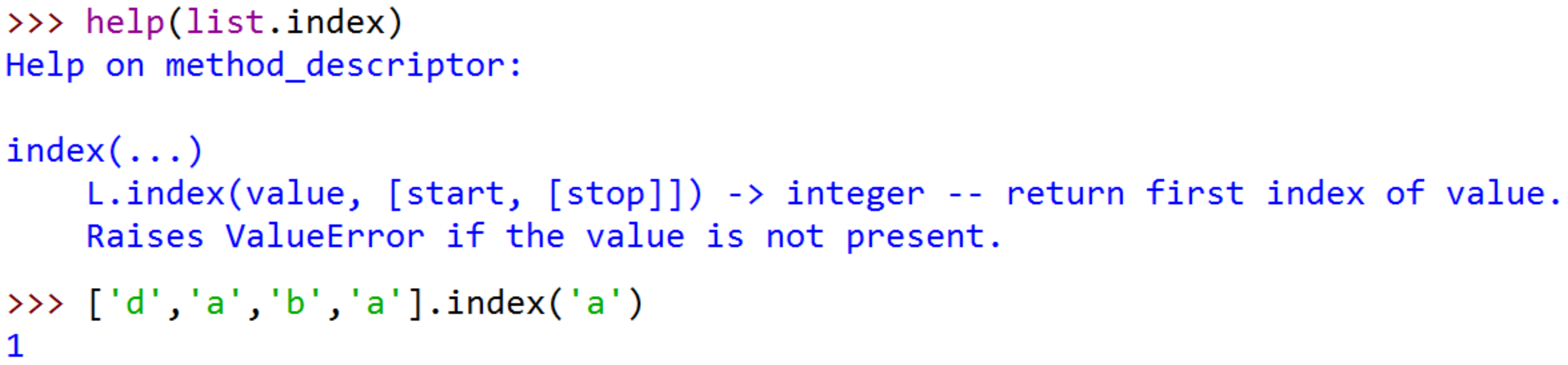
- Sorting algorithms*
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Mergesort

Searching and sorting

- How to organize, store and retrieve data
- According to IBM, 90% of the data in the world has been generated in the past two years.

- Several examples of both slower and faster algorithms to analyze a large amount of data

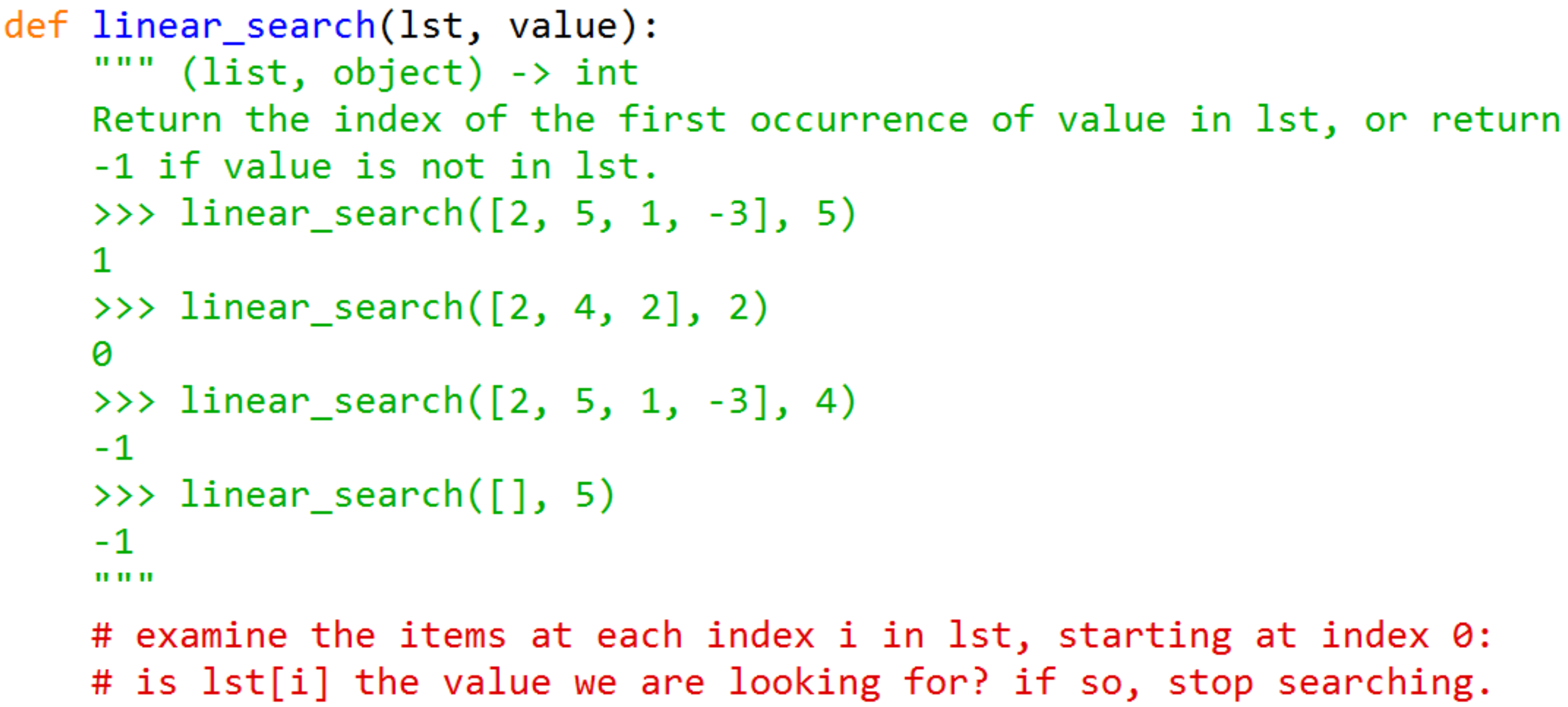
Searching a list



```
>>> help(list.index)
Help on method_descriptor:

index(...)
    L.index(value, [start, [stop]]) -> integer -- return first index of value.
    Raises ValueError if the value is not present.

>>> ['d','a','b','a'].index('a')
1
```

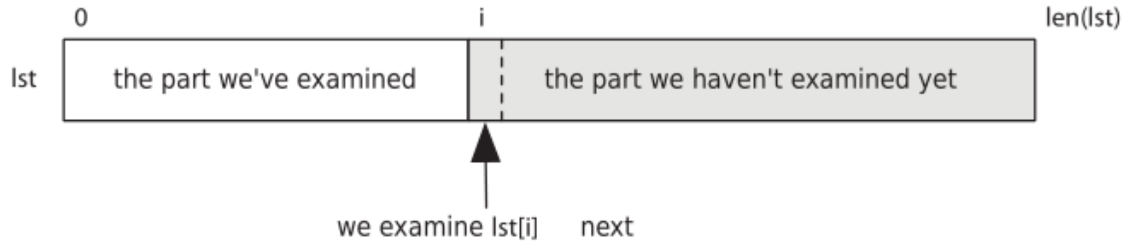
- Linear search starts at index 0 and looks at each item one by one.
- At each index, we ask: “Is the value we are looking for at the current index?”

Linear search



```python
def linear_search(lst, value):
    """ (list, object) -> int
    Return the index of the first occurrence of value in lst, or return
    -1 if value is not in lst.
    >>> linear_search([2, 5, 1, -3], 5)
    1
    >>> linear_search([2, 4, 2], 2)
    0
    >>> linear_search([2, 5, 1, -3], 4)
    -1
    >>> linear_search([], 5)
    -1
    """
    # examine the items at each index i in lst, starting at index 0:
    # is lst[i] the value we are looking for? if so, stop searching.
```

Linear search

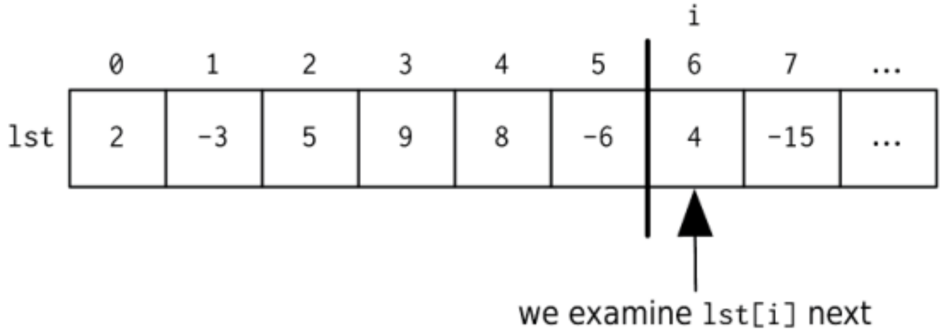


0 i len(lst)

lst the part we've examined the part we haven't examined yet

we examine lst[i] next

`lst=[2,-3,5,9,8,-6,4,15,…]`



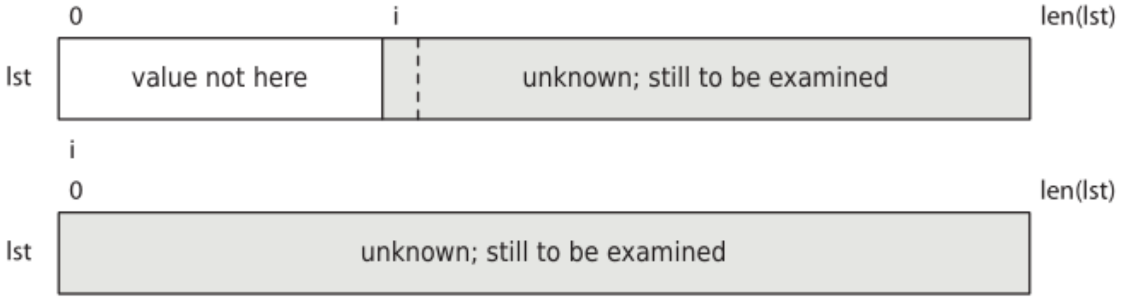
	0	1	2	3	4	5	6	7	...
lst	2	-3	5	9	8	-6	4	-15	...

i

we examine `lst[i]` next

`lst[0:i]` doesn’t contain value,

`0 <= i <= len(lst)`



0 i len(lst)

lst value not here unknown; still to be examined

i 0 len(lst)

lst unknown; still to be examined

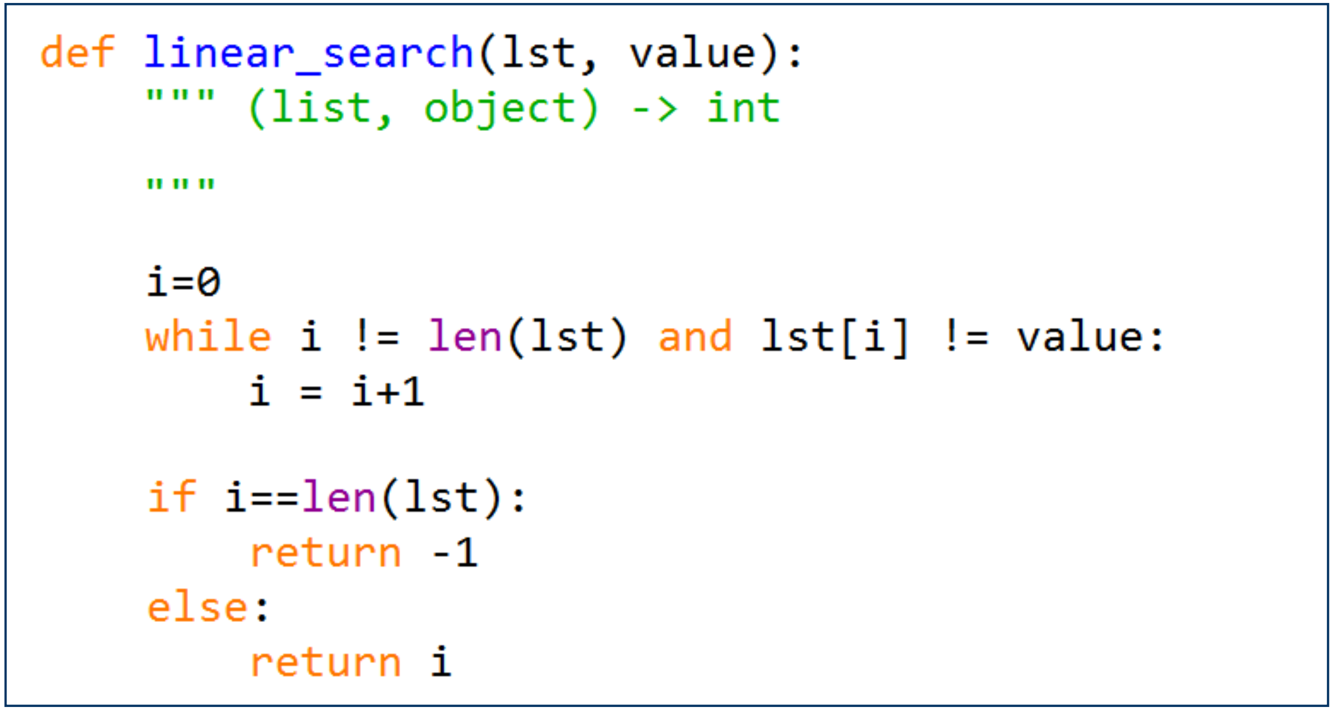
While loop version of linear search

```
Examine every index i in lst, starting at index 0:
    Is lst[i] the value we are looking for? If so, stop searching
```

```
i=0  # The index of the next item in lst to examine
While the unknown section isn't empty, and lst[i] isn't the value we are looking for:
    add 1 to i
```

While loop version of linear search

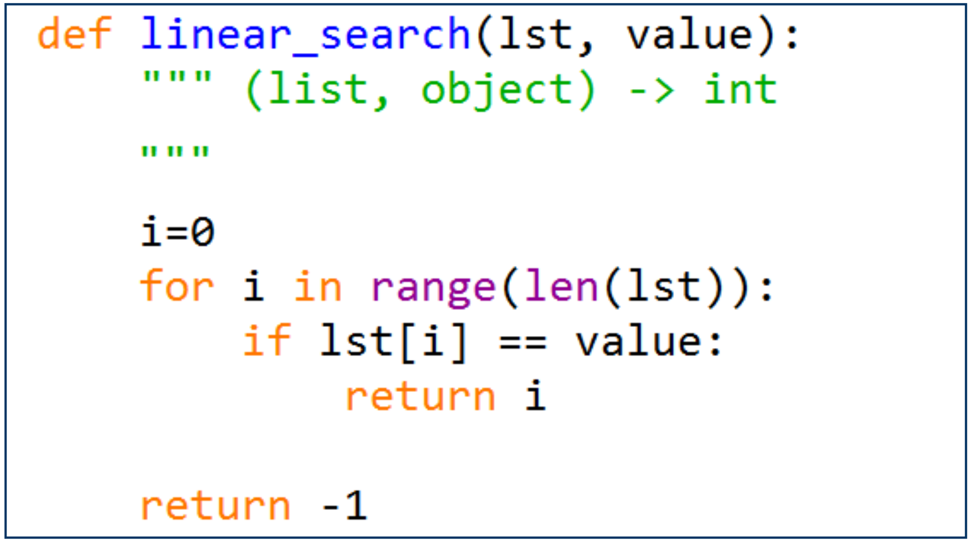
```
i=0  # The index of the next item in lst to examine
While the unknown section isn’t empty, and lst[i] isn’t the value we are looking for:
    add 1 to i
```



```
def linear_search(lst, value):
    """ (list, object) -> int

    """

    i=0
    while i != len(lst) and lst[i] != value:
        i = i+1

    if i==len(lst):
        return -1
    else:
        return i
```

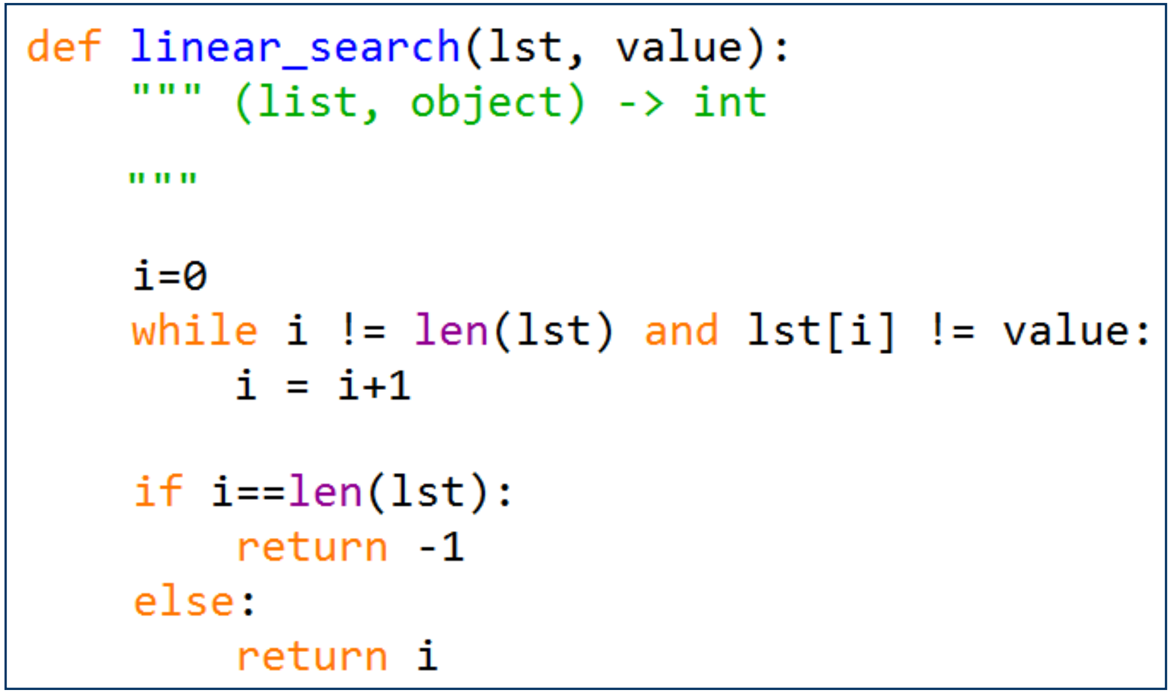
For loop version of linear search

```
i=0  # The index of the next item in lst to examine
For each index i in lst:
    If lst[i] is the value we are looking for:
        return i
If we get here, value was not in lst, so we return -1
```

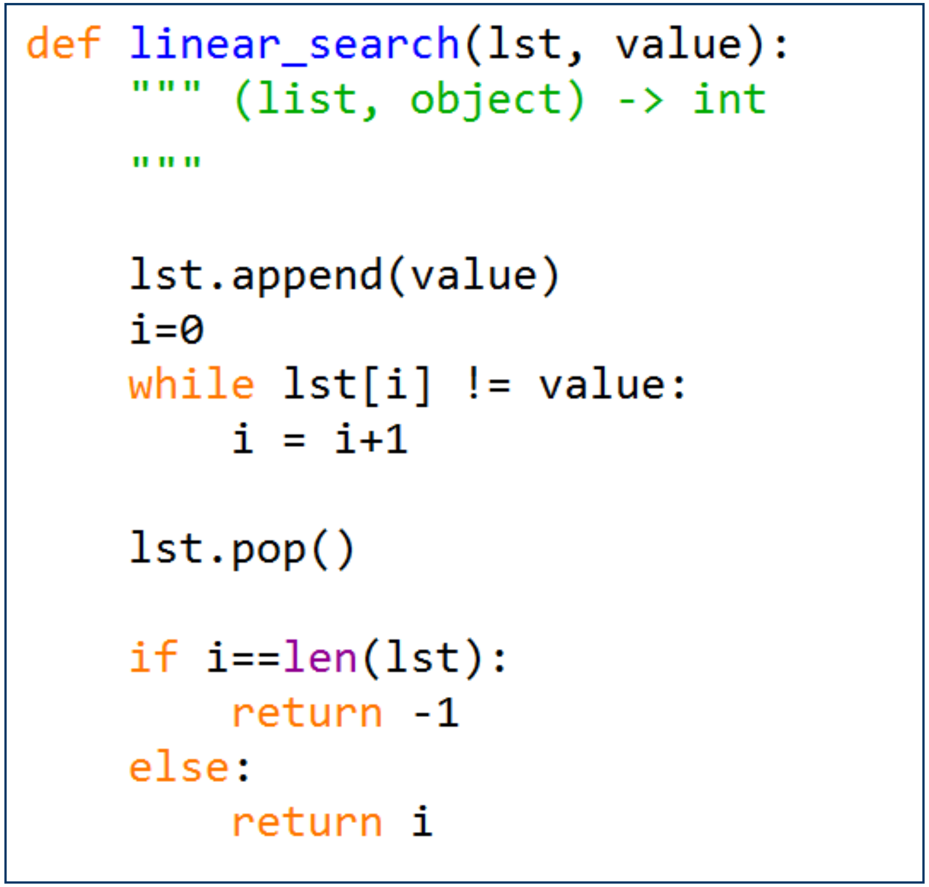


```python
def linear_search(lst, value):
    """ (list, object) -> int
    """
    i=0
    for i in range(len(lst)):
        if lst[i] == value:
            return i

    return -1
```

Sentinel search

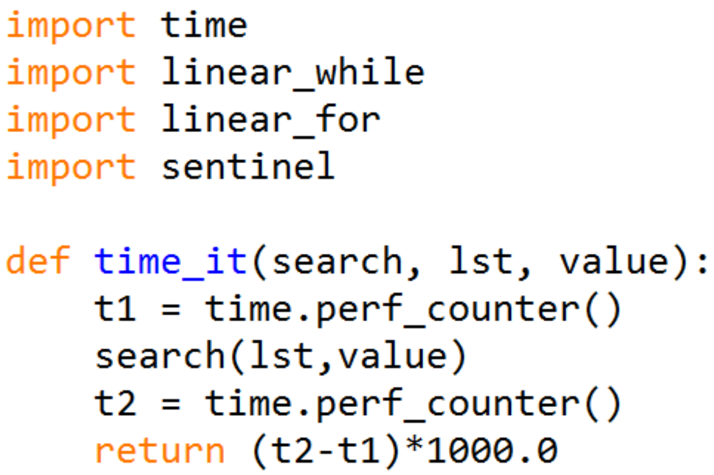


```python
def linear_search(lst, value):
    """ (list, object) -> int

    """

    i=0
    while i != len(lst) and lst[i] != value:
        i = i+1

    if i==len(lst):
        return -1
    else:
        return i
```

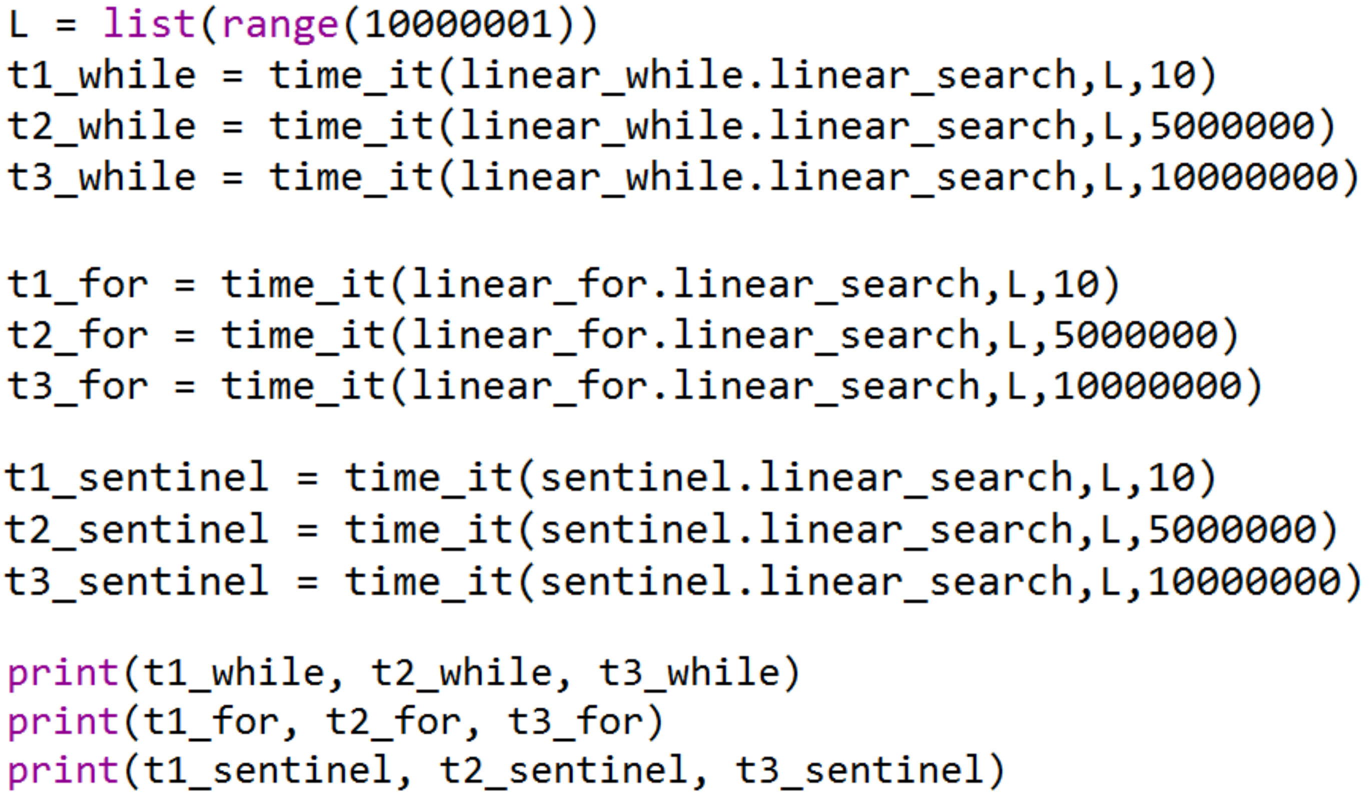


```python
def linear_search(lst, value):
    """ (list, object) -> int

    """

    lst.append(value)
    i=0
    while lst[i] != value:
        i = i+1

    lst.pop()

    if i==len(lst):
        return -1
    else:
        return i
```

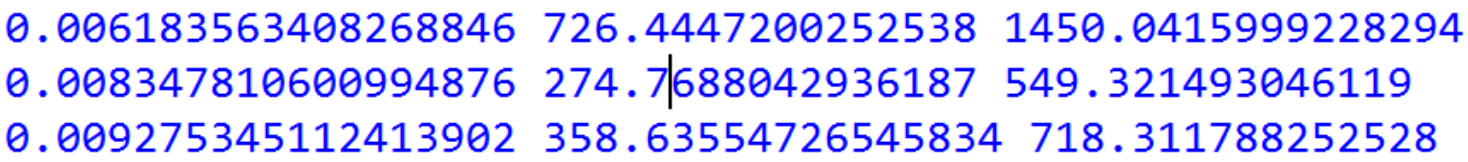
Let’s time the functions



```
import time
import linear_while
import linear_for
import sentinel

def time_it(search, lst, value):
    t1 = time.perf_counter()
    search(lst,value)
    t2 = time.perf_counter()
    return (t2-t1)*1000.0
```

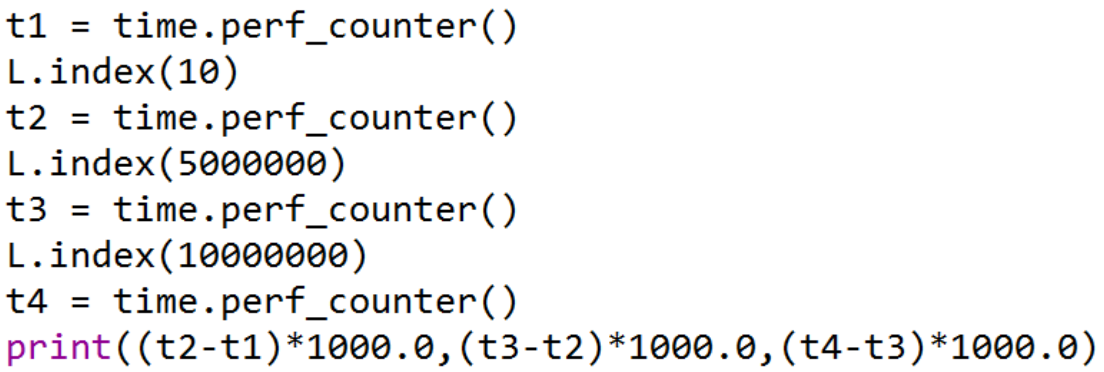


```
L = list(range(10000001))
t1_while = time_it(linear_while.linear_search,L,10)
t2_while = time_it(linear_while.linear_search,L,5000000)
t3_while = time_it(linear_while.linear_search,L,10000000)

t1_for = time_it(linear_for.linear_search,L,10)
t2_for = time_it(linear_for.linear_search,L,5000000)
t3_for = time_it(linear_for.linear_search,L,10000000)

t1_sentinel = time_it(sentinel.linear_search,L,10)
t2_sentinel = time_it(sentinel.linear_search,L,5000000)
t3_sentinel = time_it(sentinel.linear_search,L,10000000)

print(t1_while, t2_while, t3_while)
print(t1_for, t2_for, t3_for)
print(t1_sentinel, t2_sentinel, t3_sentinel)
```



```
0.006183563408268846 726.4447200252538 1450.0415999228294
0.008347810600994876 274.7688042936187 549.321493046119
0.009275345112413902 358.63554726545834 718.311788252528
```

Let’s time the functions

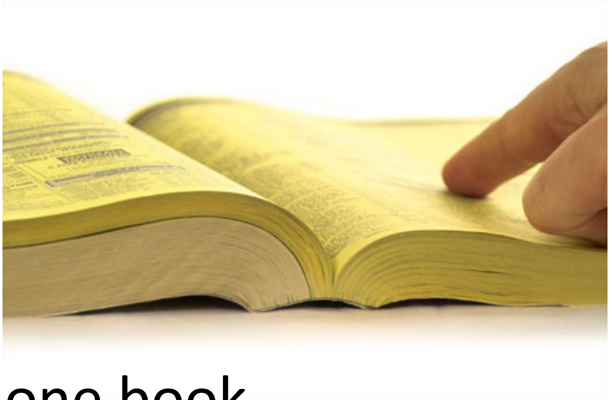


```
t1 = time.perf_counter()
L.index(10)
t2 = time.perf_counter()
L.index(5000000)
t3 = time.perf_counter()
L.index(10000000)
t4 = time.perf_counter()
print((t2-t1)*1000.0,(t3-t2)*1000.0,(t4-t3)*1000.0)
```

Case	While	for	sentinel	List.index
First	0.006	0.008	0.009	0.017
Middle	726	275	359	60
Last	1450	549	718	121

The time grows linearly with the amount of data being processed.

Binary search



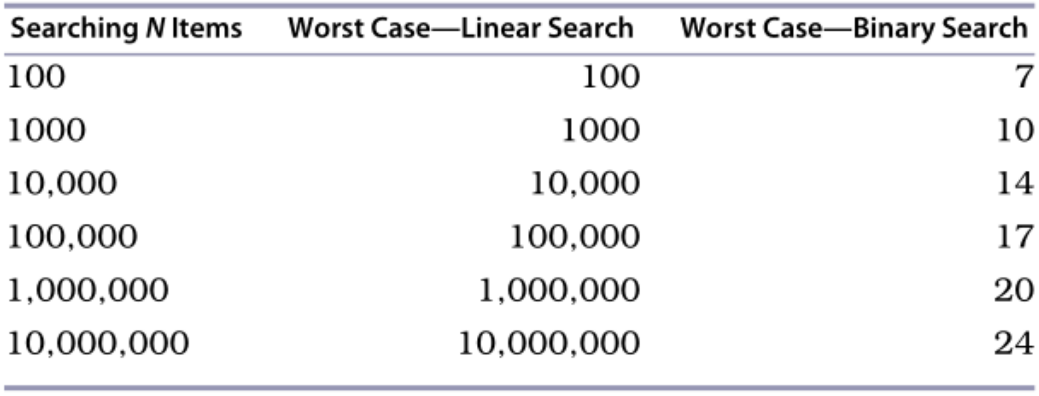
- Is there a faster way to find values?
- Yes, provided the list is sorted

- Suppose you are searching a name in a thick phone book…

- In binary search, each step divides the remaining data into two equal parts and discards one of the two halves.

How fast is the binary search?

- One step divides two values
- Two steps divides four values
- 3 steps divides $2^3 = 8$ values
- N values can be searched in roughly $\log_2 N$ steps

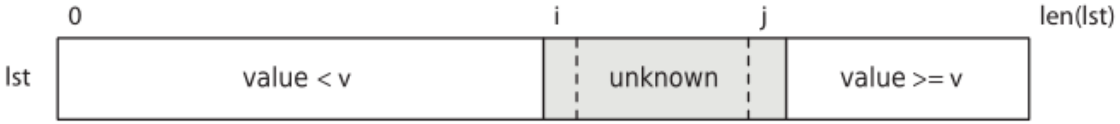


Searching *N* Items	Worst Case—Linear Search	Worst Case—Binary Search
100	100	7
1000	1000	10
10,000	10,000	14
100,000	100,000	17
1,000,000	1,000,000	20
10,000,000	10,000,000	24

Table 18—Logarithmic Growth

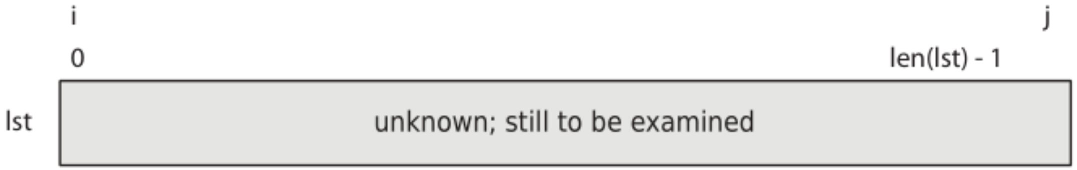
Binary search: overview

- Keep track of three parts of the list
  - Left part: values that are smaller than the value we are searching for
  - Right part: values that are equal to or larger than the value we are searching for
  - Middle part: values that we haven’t yet examined – the unknown section

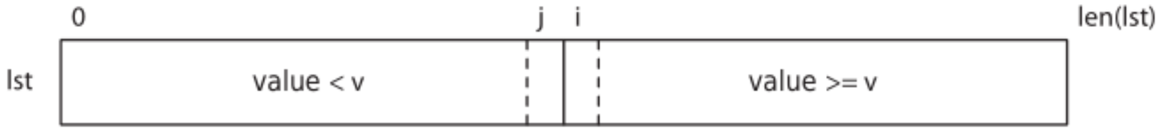


- At the beginning:

  the entire section is unknown



- We are done when that unknown section is empty (i==j+1)

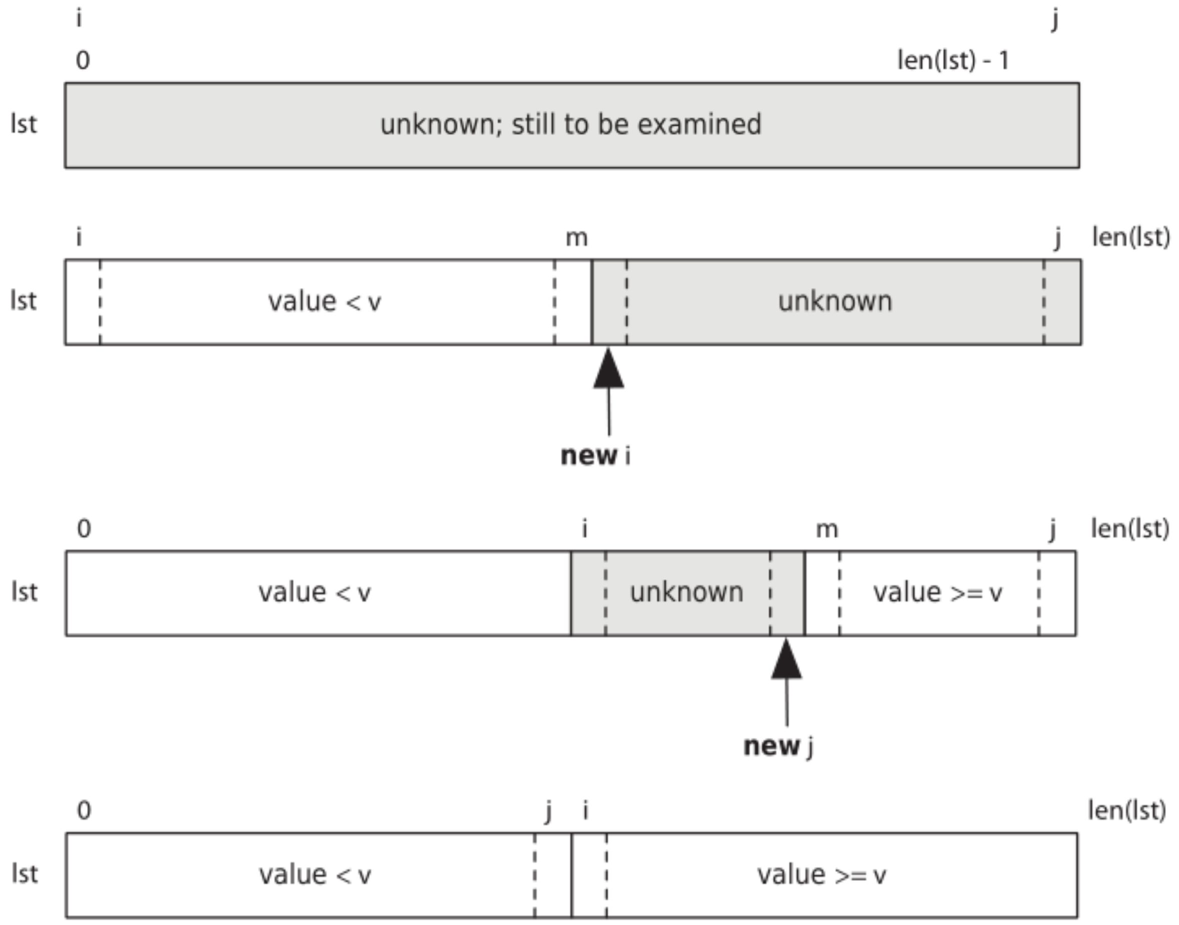


Binary search: overview

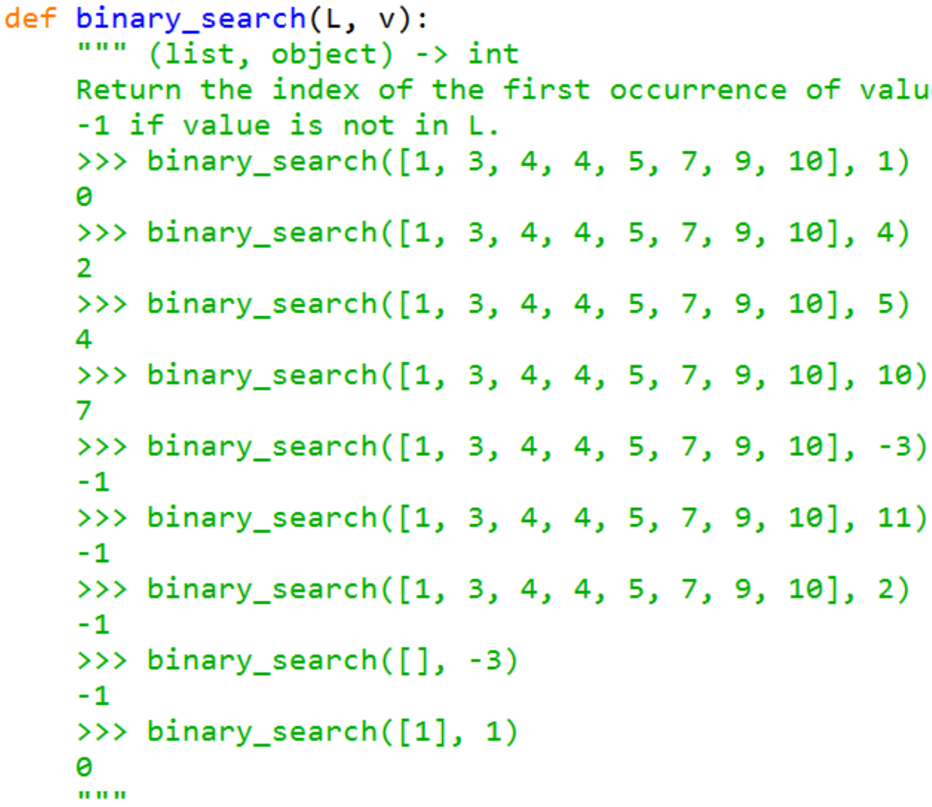
- lst is a sorted list
- We are looking for v
- i=0, j=len(lst)-1

- If lst[m] < v, i -> m+1

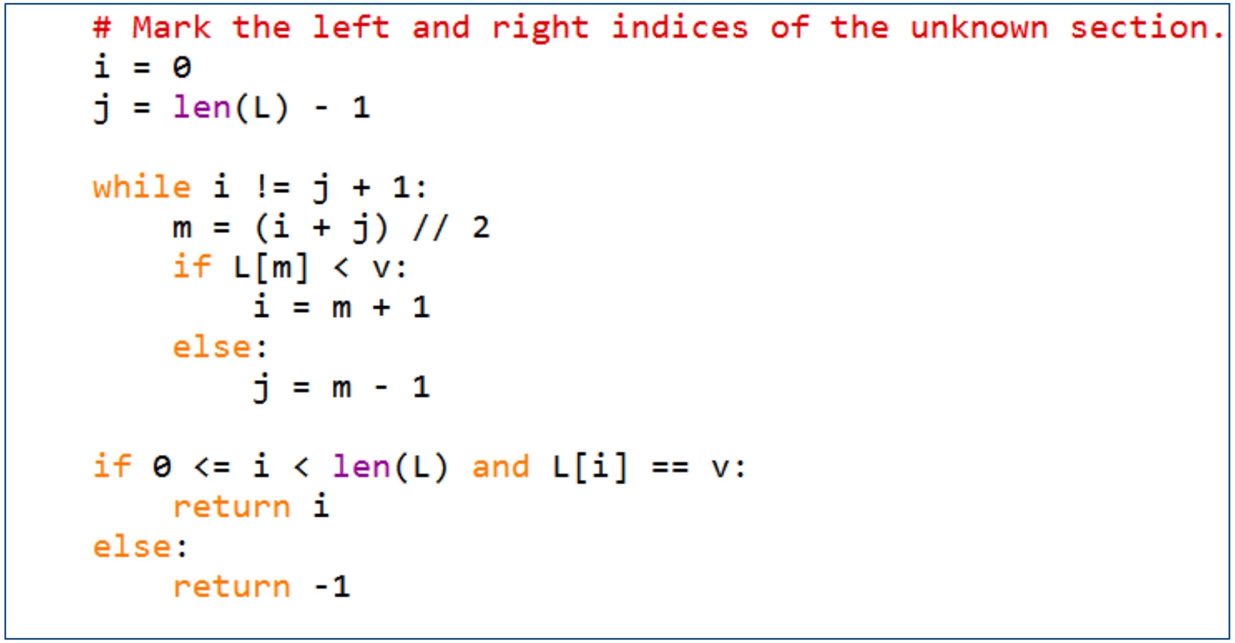
- If lst[m] > v, j -> m-1



Binary search: Python code



```
def binary_search(L, v):
    """ (list, object) -> int
    Return the index of the first occurrence of value in L, or return
    -1 if value is not in L.
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 1)
    0
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 4)
    2
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 5)
    4
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 10)
    7
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], -3)
    -1
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 11)
    -1
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 2)
    -1
    >>> binary_search([], -3)
    -1
    >>> binary_search([1], 1)
    0
    """
```



```
    # Mark the left and right indices of the unknown section.
    i = 0
    j = len(L) - 1

    while i != j + 1:
        m = (i + j) // 2
        if L[m] < v:
            i = m + 1
        else:
            j = m - 1

    if 0 <= i < len(L) and L[i] == v:
        return i
    else:
        return -1
```

How to test?

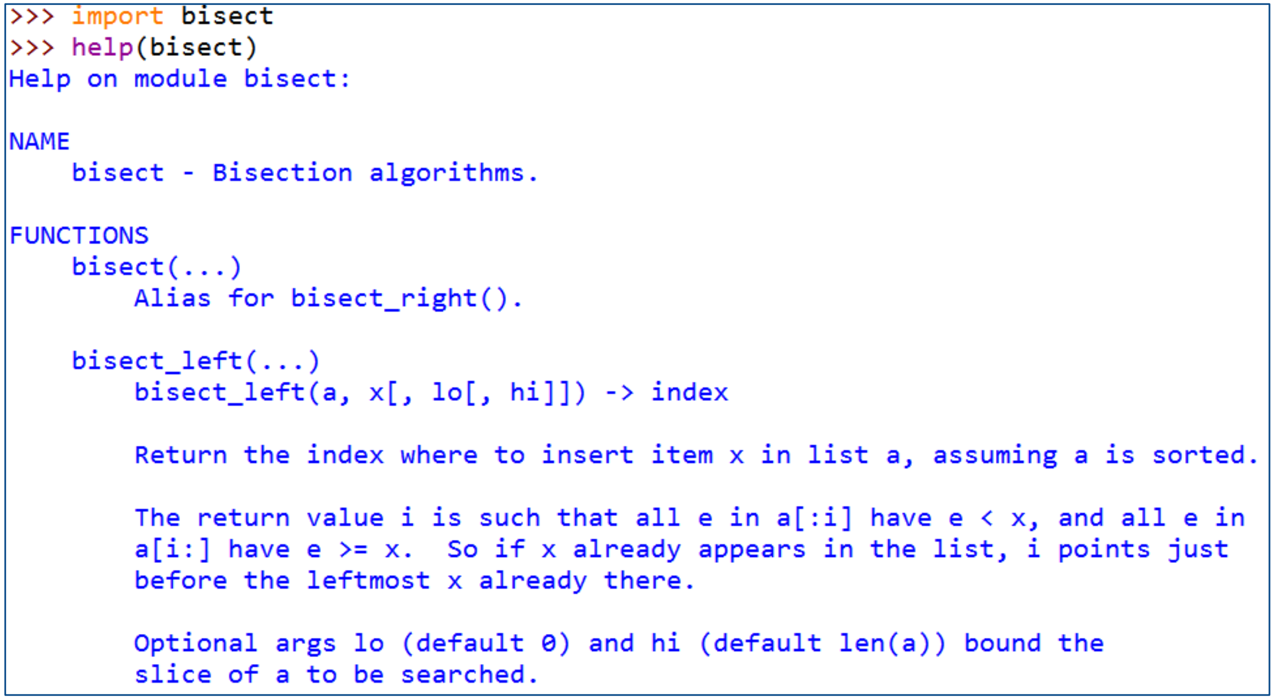
- The value is the first item.
- The value occurs twice. We want the index of the first one.
- The value is in the middle of the list.
- The value is the last item.
- The value is smaller than everything in the list.
- The value is larger than everything in the list.
- The value isn’t in the list, but it is larger than some and smaller than others.
- The list has no items.
- The list has one item

Binary search running time

Case	While	for	sentinel	List.index	Binary search
First	0.006	0.008	0.009	0.017	0.020
Middle	726	275	359	60	0.013
Last	1450	549	718	121	0.013

- Built-in binary search

- How to sort the list?



```
>>> import bisect
>>> help(bisect)
Help on module bisect:

NAME
    bisect - Bisection algorithms.

FUNCTIONS
    bisect(...)
        Alias for bisect_right().

    bisect_left(...)
        bisect_left(a, x[, lo[, hi]]) -> index

        Return the index where to insert item x in list a, assuming a is sorted.

        The return value i is such that all e in a[:i] have e < x, and all e in
        a[i:] have e >= x.  So if x already appears in the list, i points just
        before the leftmost x already there.

        Optional args lo (default 0) and hi (default len(a)) bound the
        slice of a to be searched.
```

Summary

- Linear search is the simplest way to find a value in a list; but on average, the time required is directly proportional to the length of the list.
- Binary search is much faster – the average time is proportional to the logarithm of the list’s length – but it works only if the list is in sorted order.