Everything You Can Do with Python's bisect Module

MARTIN Nov 8, 2023 | #Python

While Python's `bisect` module is very simple - containing really just 2 functions - there's a lot one can do with it, including searching data efficiently, keeping any data sorted, and much more - and in this article we will explore all of it!

What is Bisect(ion)?

Before we start playing with the module, let's first explain what *bisect(ion)* actually is. An official definition:

> *"Bisection is the division of a given curve, figure, or interval into two equal parts (halves)."*

Which in plain English means that it implements a binary search. In practice that means that we can use it to - for example - insert elements into a list while maintaining the list in sorted order:

```python
import bisect

some_list = [0, 6, 1, 5, 8, 2]
some_list.sort()
print(some_list) # [0, 1, 2, 5, 6, 8]

i = bisect.bisect_left(some_list, 4)
print(i)  # 3

some_list.insert(i, 4)
print(some_list)  # [0, 1, 2, 4, 5, 6, 8]
# OR
bisect.insort_left(some_list, 4)
print(some_list)  # [0, 1, 2, 4, 5, 6, 8]
```

In this basic example, we first sort a list because we can only use functions from `bisect` on sorted iterable. We then use `bisect_left` on the list to find an index where the second argument (`4`) should be inserted to maintain sorted order. We then proceed to do just that - insert the number `4` at index `3`.

Well, that's cool, but why should you care about this module, though? Well, let me show you all the useful things you can do with it...

Binary Search

As already mentioned, `bisect` implements binary search so the most obvious use for it is just that:

```
from bisect import bisect_left

def binary_search(a, x, lo=0, hi=None):
    if hi is None:
        hi = len(a)
    pos = bisect_left(a, x, lo, hi)  # find insertion position
    return pos if pos != hi and a[pos] == x else -1  # don't walk off the end

print(binary_search([0, 1, 2, 5, 6, 8], 5))  # 3
print(binary_search([0, 1, 2, 5, 6, 8], 4))  # -1
```

The parameters of `binary_search` function above follow the same pattern as the functions in `bisect` module. That is - we look for value `x` in the list `a` between index `lo` and `hi`.

The only interesting line is the `return` statement, where we test whether the value `x` is actually in the list, if yes we return its position, otherwise we return `-1`.

Successive Equal Values

There are however more interesting things we can do with `bisect` module, for example finding successive equal values in a list:

```
from bisect import bisect_left, bisect_right

some_list = [5, 10, 15, 15, 15, 20, 25, 40]
# Find all the 15's
value = 15
start = bisect_left(some_list, value)
end = bisect_right(some_list, value)
print(f'Successive values of {value} from index {start} to {end}: {some_list[start:
# Successive values of 15 from index 2 to 5: [15, 15, 15]
```

start and end of the span of successive values we're looking for.

Mapping from Intervals to Values

Now, let's imagine that we have a series of intervals/ranges, and we want to return corresponding ID/value. A naive, ugly solution could look something like:

```python
def interval_to_value(val):
    if val <= 100:
        return 0
    elif 100 < val <= 300:
        return 1
    elif 300 < val <= 500:
        return 2
    elif 500 < val <= 800:
        return 3
    elif 800 < val <= 1000:
        return 4
    elif val > 1000:
        return 5
```

But, there's a much nicer solution using `bisect_left`:

```python
def interval_to_value(val):
    return bisect_left([100, 300, 500, 800, 1000], val)
```

This isn't just very clean solution but also super fast. It can be also extended in case you'd need a non-natural ordering or, for example, if you wanted to return something different, like a string:

```python
i = interval_to_value(325)
a = ['absent', 'low', 'average', 'high', 'very high', 'extreme']
print(a[i])  # average
```

Closest Key in Dictionary

Now, let's say we have a mapping in form of a dictionary, and we want to lookup values for a specified key. If the key exist then cool, but if it doesn't, then we want to return the value of the closest key:

```python
import collections
some_dict = collections.OrderedDict(
target = 10.5
index = bisect_left(list(some_dict.keys()), target)  # 6

items = list(some_dict.items())

# Check which one is closer:
print(f'Distance for to index {index}: {distance1}')    # Distance for to index 6:
print(f'Distance for to index {index-1}: {distance2}')  # Distance for to index 5:

print('Closest value:')
if distance1 < distance2:
    print(items[index])
else:
    print(items[index-1])

# Closest value: (10, 25)
```

Here we use `OrderedDict` to make sure that we have the keys in correct order. We then use `bisect_left` on them to find insertion point. Finally, we need to check whether the next or the previous index is closer to the `target`.

Prefix Search

Another thing you can use `bisect` for is prefix search - let's assume we have a very large word list and want to lookup words based on a given prefix:

```
def prefix_search(wordlist, prefix):
    try:
        index = bisect_left(wordlist, prefix)
        return wordlist[index].startswith(prefix)
    except IndexError:
        return False


words = ['another', 'data', 'date', 'hello', 'text', 'word']

print(prefix_search(words, 'dat'))  # True
print(prefix_search(words, 'xy'))  # False
```

The function above only check whether a word with specified prefix exists in the list, but this could be easily modified to loop from the `index` and return all the words starting with `prefix`.

Sorted Custom Objects

So far we've only used built-in types, but functions from `bisect` module can be also applied to custom types. Let's say that we have a list of custom objects, and we want to maintain their order in the list based on some attribute:

```
from bisect import insort_left

class CustomObject:
    def __init__(self, val):
        self.prop = val  # The value to compare

    def __lt__(self, other):
        return self.prop < other.prop

    def __repr__(self):
        return 'CustomObject({})'.format(self.prop)

some_objects = sorted([CustomObject(7), CustomObject(1), CustomObject(3), CustomObj

insort_left(some_objects, CustomObject(2))
print(some_objects)  # [CustomObject(1), CustomObject(2), CustomObject(3), CustomOb
```

This code snippet uses the fact that `bisect` uses `__lt__` magic method to compare objects. However, having to use `bisect` functions all the time might be a bit inconvenient, to avoid that you could implement a `SortedCollection` described in this more complete example/recipe.

Key Function

Functions in `bisect` module also support more complicated comparison/search using the the `key` function parameter:

```
some_list = [1, 3, 7, 16, 25]
some_list.reverse()
insort_left(some_list, 10, key=lambda x: -1 * x)
print(some_list)  # [25, 16, 10, 7, 3, 1]
```

Here we use the `key` function to implement a reverse order binary search, just remember that the list also has to be sorted in reverse order to begin for searching list of tuples, it's however possible to do just:

```
list_of_tuples = [(1, 3), (3, 8), (5, 4), (10, 12)]

index = bisect_left(list_of_tuples, (5, ))  # 2
print(list_of_tuples[2])  # (5, 4)
```

Omitting the second value in the `tuple` forces `bisect_left` to compare only based on the first value. If you wanted to be explicit and use the `key` function anyway, then `key=lambda i: i[0]` would work too.

With that said, the key function is somewhat unintuitive - one would expect it to work like e.g. `sorted` function:

```
some_tuples = [
    (0, 10),
    (2, 12),
    (3, 15),
    (5, 20),
]

print(sorted(some_tuples, key=lambda t: t[0] + t[1]))  # Works
```

But it doesn't:

```
index = bisect_left(some_tuples, (4, 17), key=lambda t: t[0] + t[1])  # Doesn't wor
# Expectation: index = 3
# Reality: "TypeError: '<' not supported between instances of 'int' and 'tuple'"

def key_func(t):
    return t[0] + t[1]

index = bisect_left(some_tuples, key_func((4, 17)), key=key_func)
print(index)  # 3
```

Instead, we have to define a key function, pass it as a `key` argument and invoke it on the second argument, too.

That's because (from docs):

> *"key specifies a key function of one argument that is used to extract a comparison key from each element in the array. To support searching complex records, the key function is not applied to the x value."*

design decision.

Conclusion

In my opinion, for such a tiny module, that's a lot of things you can use it for.

Also, besides all the above useful things you can do with `bisect`, I want to highlight how fast this is - especially if you have list that's already sorted. For example consider this example on Stack Overflow showing that `bisect_left` is much faster than `in` operator.

It being *binary search* naturally means that it runs in `O(log(n))`, but it's further helped by being precompiled in C, so it's always going to be faster than anything you write yourself.

Subscribe:

E-mail Address

Submit