Published in Towards Data Science

Emma Boudreau Follow
Aug 24, 2021 · 7 min read · · Listen

Save

Everything You Need to Know about Indexing in Python

An overview on indexing different data-types and structures in Python.

(src = https://pixabay.com/images/id-1690423/)

Introduction

In general-purpose programming, it is quite common to work with data structures. Data structures are types that are composed of smaller data-types. An example of a data structure is list, or a dictionary. Data structures allow us to organize and work with several components conveniently as members of the same consistent variable. As you might imagine, this makes data structures a very important component in Data Science. Given that data structures are made of smaller components, there surely is a way to access individual components based on some feature. For this, we use indexing.

Indexing is important because it allows us to call a portion of a data structure quite effortlessly in order to work with components inside of a structure individually. Of course, this is a very important thing to master for Data Science, as it is probably quite frequently that a Data Scientist is going to be working with data structures.

Notebook

. . .

Indexing Types

Before we get into working with indexes, let us look at what types can actually be indexed. In the Python programming language, types can be attributed the ability to be indexed using the __getitem__() method in a given class. This means that we can apply the indexing methodology to any sort of types by simply adding this method with an accurate return. To try this, we will first make a class. Consider the following example:

```
class Winners:
    def __init__(self, first, second, third):
        self.first, self.second = first, second
        self.third = third
```

Now we can add the __getitem__() method to this new class. This will add the ability to easily get our placements out of the class using simple indexes.

```
def __getitem__(self, x):
```

For this example, I thought the best method of attack was to create a dictionary on the initialization of this class's constructor. With this method, we will be able to simply call indexes of the dictionary based on numbers in order to receive the placements for this race.

```
class Winners:
    def __init__(self, first, second, third):
        self.first, self.second = first, second
        self.third = third
        self.index = dict({1 : self.first, 2 : self.second,
                           3 : self.third})
```

Now finally, we will finish off our __getitem__() method by simply calling the dictionary key with the number provided:

```
class Winners:
     def __init__(self, first, second, third):
           self.first, self.second = first, second
           self.third = third
           self.index = dict({1 : self.first, 2 : self.second,
                              3 : self.third})
    def __getitem__(self, x):
        return(self.index[x])
```

While this is certainly fine, let us remember that we are only logging winners here, so anyone beyond the point of 4 is not contained in this class. That in mind, if we were to call the index of 4 on this class, we would get a KeyError in return. Whenever we create software, especially in classes that users of our software likely will not ever look at, we will want to make throws to make errors a little more obvious than that. That in mind, we will add a try and catch into this method in order to print out a more detailed error. The final result looks a little something like this:

```
class Winners:
    def __init__(self, first, second, third):
        self.first, self.second = first, second
        self.third = third
        self.index = dict({1 : self.first, 2 : self.second,
                           3 : self.third})
    def __getitem__(self, x):
        try:
             return(self.index[x])
        except KeyError:
             print("""KeyError!\nOnly keys 1-3 are stored in this
class!""")
```

Now let us try and index this type. First of course we will need to initialize a new instance of this object, then we will index it. This is done with the [] syntax:

```
race_winners = Winners("Nancy", "Bobby", "Reagan")
```

First let us try to index it with a 4 in order to see what kind of return we get!

```
race_winners[4]
```

```
KeyError!
Only keys 1-3 are stored in this class!
```

And now we will print 1:3 in the indexes of this class:

```
print(race_winners[1])

print(race_winners[2])

print(race_winners[3])

Nancy
Bobby
Reagan
```

Index-able types in Python

The Python programming language comes with several data-types and data-structures that can be indexed right off the bat. The first that we are to take a look at in this article

```
dct = dict({"A" : [5, 10, 15], "B" : [5, 10, 15]})
```

We can index a dictionary using a corresponding dictionary key. This will give us the value pair for this given key. Which conveniently will give us our next data structure, the list:

```
lst = dct["A"]
```

A list can be indexed with the position of the element we would like to access. For example, the second element of our new lst list is ten. We can call on this ten using

```
lst[1]
```

Of course, this is one, not two, as indexes start at zero in Python. Needless to say, with lists indexes can certainly come in handy. Another type we can index is a string:

```
"Hello"[1]

'e'
```

Setting Indexes

Equally as important as calling indexes is setting indexes. Setting indexes allows us to not only create new positions on lists and other iterable data-structures, but also alter existing values inside of data structures. Additionally, we can push keys into dictionaries, add columns to Pandas dataframes, and more using this method. In Python, index setting calls the __setitem__() method. Let us go ahead and write a function for this:

```
def __setitem__(self, x, y):
    pass
```

37

Now we will write a bit of logic into this new function, allowing it to set a corresponding dictionary key's value into our new input value:

```
def __setitem__(self, x, y):

self.index[x] = y
```

Now we can set indexes of our race positions to new values:

```
print(race_winners.index.values())

dict_values(['Nancy', 'Bobby', 'Reagan'])

race_winners[2] = "John"

print(race_winners.index.values())
```

```
dict_values(['Nancy', 'John', 'Reagan'])
```

Of course, this same exact concept applies for all of the data-structures in Python. We can apply this to lists, dictionary values, but NOT strings as follows:

```
z = [5, 10]
z[1] = 1
d = dict({"h" : 5, "z" : 6})
d["z"] = 5
assert d["z"] == d["h"]
assert z[1] < z[0]
```

Important Functions

Now that we understand the basics of indexing, let us look a bit into some vital functions we might want to use when working with data structures. Many of these functions are incredibly useful for working with copious amounts of data. It should be noted that these are primarily for lists and matrices, though some functions might work on other structures. This will be the list we are going to work with for these functions:

```
lst = [5, 1, 6, 4, 5, 7, 3,
5, 4, 3, 1, 2, 3, 4,]
```

1. insert()

The first useful function is insert. Insert will allow you to place any value at any index in an array. This can be useful if you want a specific array to contain a specific component in a location. It is very easy to use, just add the two arguments, position and then the value that you wish to add.

```
lst.insert(5, 3)
```

2. append()

The next function is the append function. This function is very frequently used to generate lists, usually in iteration. This will add a value to the next available index, and only requires a single value as an argument.

```
lst.append(5)
```

While a better method for this particular case might be lambda with mapping, or perhaps the apply method if this were a series type, in this case I will use an iterative loop and the append function in order to demonstrate how it can be used in this way:

```
lst2 = []

for z in lst:

     lst2.append(z * 3 + 2)
```

3. remove()

Remove will remove a value from a given list. Note that this takes a value, not an index! It will also only remove only one instance of the given value.

```
lst.remove(1)
```

4. extend()

The extend function is essentially just the append() method, but allows us to append a list to the end of our list:

```
lst.extend([101, 503])
```

5. count()

Count will return an integer that is a count of all instances of an element in a given list.

```
lst.count(5)

4
```

A great example of a use case for this function is calculating the mode. Observe how I use count and a dictionary to get the mode in an iterative loop:

```
digit_n = {}

for x in set(lst):

    cnt = lst.count(x)

    digit_n[cnt] = x

     mode = digit_n[max(digit_n.keys())]

print(mode)

5
```

6. sort()

The last vital list function I wanted to discuss is the sort() function. This function takes no parameters, and will sort a list logically, or as specified with key-word arguments. In the following example, I use sort to find the median of our list:

```
lst.sort()
medianindex = int(len(lst) / 2)

print(medianindex)

print(lst[medianindex])
```

. . .

Conclusion

Needless to say, indexing is an important thing to know well when it comes to writing just about any code involving data. The standard functions of the list class can also come in handy very frequently for manipulating, creating, and getting insights from data. I think this is just about all anyone could want to know about indexing in Python. Thank you very much for reading, and I hope you have a wonderful rest of your day or night!

Python Data Science Programming Software Development Coding