MODULE 02 - 066: Python - Dictionary View Object / `.keys()`

Introduction

Dictionary view objects were introduced in **Python 3** to provide a **dynamic view** of dictionary keys, values, and items. Unlike lists, these views reflect changes in the dictionary **in real time**. Understanding them is crucial when working with large datasets, iterating over dictionaries, or ensuring thread safety in your programs.

Python Documentation: Dictionary View Objects

Understanding Dictionary View Objects

A dictionary in Python is a collection of key-value pairs:

```
players = {
    "ss": "Correa",
    "2b": "Altuve",
    "3b": "Bregman",
    "DH": "Gattis",
    "OF": "Springer",
}
```

Python provides three view objects:

Method	Description
`.keys()`	Returns a view of the dictionary's keys
`.values()`	Returns a view of the dictionary's values
`.items()`	Returns a view of key-value pairs as tuples

Let's inspect these views:

```
print(players.keys())   # dict_keys(['ss', '2b', '3b', 'DH', 'OF'])
print(players.values()) # dict_values(['Correa', 'Altuve', 'Bregman', 'Gattis', 'Springer'])
print(players.items())  # dict_items([('ss', 'Correa'), ('2b', 'Altuve'), ...])
```

The output shows `dict_keys`, `dict_values`, and `dict_items`, indicating that these are **view objects**, not lists.

Why Use Dictionary View Objects?

Unlike lists, dictionary views **update dynamically** as the dictionary changes.

```
team = {"yankees": ["Judge", "Stanton"]}
k_view = team.keys()
print(k_view)  # dict_keys(['yankees'])

team["red sox"] = ["Price", "Betts"]  # Modify dictionary
print(k_view)  # dict_keys(['yankees', 'red sox'])  View updated!
```

The `dict_keys` view **reflects changes** without needing to recreate the object.

However, **view objects do not support indexing**:

```
print(players.keys()[0])  #  TypeError: 'dict_keys' object does not support indexing
```

To convert a view into a list:

```
key_list = list(players.keys())
print(key_list[0])  #  'ss'
```

Python Documentation: Dictionary Keys

Working with `.items()`

`.items()` returns **key-value pairs as tuples**, which can be useful for iteration.

```
for position, player in players.items():
    print(f"{position}: {player}")
```

Output:

```
ss: Correa
2b: Altuve
3b: Bregman
DH: Gattis
OF: Springer
```

Python Documentation: Dictionary Items

`.values()` and Mutability Issues

Using `.values()` directly can lead to unintended side effects because it **reflects changes dynamically**:

```
vals = players.values()
print(vals)  # dict_values(['Correa', 'Altuve', 'Bregman', 'Gattis', 'Springer'])

players["C"] = "Maldonado"
print(vals)  # dict_values(['Correa', 'Altuve', 'Bregman', 'Gattis', 'Springer', 'Maldonado'])
```

The view **automatically updates**, which can lead to unpredictable behavior in multi-threaded applications.

Solution: Convert it to a list before using it:

```
safe_values = list(players.values())  # This is now an independent copy
```

Python Documentation: Dictionary Values

Best Practices and Performance Considerations

Use dictionary views for iteration: They are memory efficient because they avoid copying data. **Convert views to lists if modifications are needed:** View objects **do not support indexing**. **Be mindful of dynamic updates:** Use `.copy()` when a static snapshot is required.

```
safe_players = list(players.copy().values())
```

Summary

Dictionary view objects (`.keys()`, `.values()`, `.items()`) provide **real-time** access to dictionary elements. They **do not support indexing** but can be converted to lists. Views **update dynamically** when the dictionary changes. Use `.copy()` to create a thread-safe snapshot.

Video lesson Speech

Now that you have a high-level understanding of how dictionaries work such as how to create them how to add to them and multiple ways on how to query the elements inside of them.

Now let's take a look at a very important topic called the dictionary view object and **this is in the newer versions of python so Python 3 and above**.

medium

Figure 1: medium

large

Figure 2: large

And this is a little bit of a tricky syntax and a tricky concept if you've never seen it before so I want to take our time and walk through it because what dictionary view objects allow us to do is they allow us to peek in and view the values the keys and all of the different items inside of dictionaries.

So I set up two different sample dictionaries here

```
players = {
 "ss" : "Correa",
 "2b" : "Altuve",
 "3b" : "Bregman",
 "DH" : "Gattis",
 "OF" : "Springer",
}
teams = {
 "astros" : ["Altuve", "Correa", "Bregman"],
 "angels": ["Trout", "Pujols"],
 "yankees": ["Judge", "Stanton"],
 "red sox": ["Price", "Betts"]
}
```

Also, I wanted to show you **some variation in syntax** that you may see.

Usually, I will use a syntax where I have the key in a string followed directly by a colon.

However, I have seen a number of times where developers like to have the entire key wrapped in a space area in a string and, then, a space followed by the colon followed by another space and then the value.

I personally like the way that I'm used to doing it just like this.

```
"key":"value"   =   "key"  :  "value"    # Don't worry about blank spaces
```

However, I do want to show you both options because they both are completely valid and you are going to see both in various Python programs.

Okay, now, with all that being said let's see how we could grab only the keys inside of a dictionary.

So with this player dictionary if I want to do that I can say print and then players.keys and it's a function so I can call it with parens right after keys and let's see exactly what this prints out for us. So if I print this you can see that I do get the keys and these look like they're in a list.

However, we have this `dict_keys` right before the list and it wraps the entire thing in parentheses.

So, whenever you see this that means inPython is there is some type of object that is wrapping the values that we want.

In this case, it is called a dictionary view object and so if I open up right here the python documentation and I'm going to put a link in the show notes if you want to reference this.

This is where the python docs talk about dictionary view objects and they also describe them.

Now, sometimes, reading the documentation can seem a little bit tedious and convoluted but I still want to show it to you because the more advanced you get you are going to have to learn how to work through the documentation and understand exactly what it's saying.

So, right here what it's saying is the objects returned by dict_keys values and items are view objects.

They provide a dynamic view on the dictionary's entities which means that **when the dictionary changes the view reflects these changes**.

Now, if that is about as clear as mud.

large

Figure 3: large

large

Figure 4: large

Be careful with .values()

Don't worry I'm going to talk about this because this is a very important but very subtle kind of feature and it can lead to bugs if it's implemented improperly.

Let's take a look at what we have access to here.

We have keys, we have values and then items.

Let's take a look at each one of these so if I say player keys here you already saw this one is where it's grabbing the keys.

Not too much of a surprise there.

.items()

If I say values this returns another dictionary view object and it gives the values in a list and items gives us **the entire key-value pair as a tuple**.

Now we haven't gotten to tuples yet but you can at least see what it returns so if I run items you can see it returns a dictionary object of items and these parens show that it is a tuple so it is returning the key and the value for each one of these elements.

Now let's go back to let's go with values because it's easy to see what the names are.

.values()[] - With indices?

Now if I run the this we have what looks like a list

but what happens if we try to treat it like a list.

There are a few different things we can do so if I try to do zero here which theoretically if I can treat this as a list it means this would bring back the string Correa.

But when I run this it gives me an error

and the error says type error **dict_values object does not support indexing**.

OK, that's a very important thing to know.

So we cannot treat these view objects like true lists and so that is something that may seem a little bit tricky whenever you are trying to grab all the values and you think that you can treat it just like a traditional list.

Most programming languages that have a key-value data structure do have some type of function called keys and values that allow you to simply grab all of those elements and typically they're treated just like an array of some type of more plain collection.

However, in Python, it's returning a view object and I'm going to show you now exactly why they're doing that and also what you need to do in order to get access to the values themselves.

So, without going into a lot of very low-level detail we're going to see what a view is because whenever we're running processes so say that we have a query engine and we're pulling all of these players every time we run one of those queries it's going to run on what is called a thread and so that thread is going to perform that action.

Now, what do you think happens if you run a long query that takes a long time, and then some other user runs the same query, and what happens to this collection?

large

Figure 5: large

medium

Figure 6: medium

large

Figure 7: large

Well, if someone is trying to make a change to the collection, then, what could happen is you could have a conflict you could have one user who thinks that the outfielder is Bregman but some other user had actually change that value.

Sometimes that's a good thing sometimes you do want a live look into those changes.

Other times that's not such a good thing because maybe you expected to know who was already there and actually, this is a baseball typo our outfielder should be Springer and so don't send me notes on that, I fixed it.

And, so that's exactly what we have going on right here with view objects.

If you reference the docs once again it gave us a keyword here and what it is is a dynamic view.

So what this means is that if something changes inside of the dictionary this is actually going to change for us.

So, in other words, we may open up a query start running through our dictionary and if some other user at the exact same time change's one of the values the dictionary view is going to keep it open.

Because it does not close it off which means that we're going to keep all of those active changes available.

Now, if you have never heard of a thread or any of those concepts, that's perfectly fine.

This is getting into much more advanced types of computer science and programming.

I simply want to show you exactly the reason why these types of features were built into the language.

There is a very good chance you won't get into this kind of feature being an issue for a very long time but whenever that does happen hopefully you'll remember back to what the dictionary view object was created for.

So, there are a few ways around this and, so we have our players and values, and we just saw that we can't treat it like a list and so what happens if we actually just want to view these values and we do want to use of them just like traditional lists instead of a view object.

Well, thankfully we can just do that so I can say a list.

I can cast it as a list so if you remember in our section on lists how we talked about converting values into lists.

This is what we can do.

So, now if I run this you're going to see that we get our actual listof value and we can treat this exactly the way we would any list so now if I type 1 I get Altuve which is the first index item inside of that list.

So, now, we can treat these exactly the same.

Now let's address that issue of thread safety.

So which means that if we have some other user that goes or some process that changes one of these values and we have a long-running query occurring what do we do?

Because we don't want the outfielder Springer changing.

We only want the values that were there in the very beginning.

Well, that's where we can copy the list and this is a way you can make your entire process what is called thread-safe which means that we can simply make a quick copy of the list then we can perform our actions.

So I'm gonna say player names and so now I'm going to cut all of this out.

Put it down here and the process to do this is to use the copy function.

large

Figure 8: large

large

Figure 9: large

large

Figure 10: large

So I say copy values:

```
player_names = list( players   .copy().values()      )
```

Now if I pronounce player names you're going to get exactly the same list we have Correa, Altuve, Bregman, Gattis, and Springer.

Now, what's happening under the hood with this though is that a copy is made of players and it's stored and accessed only by us and by our process so if someone else goes and they're making a change to players then it doesn't even matter.

This is now completely safe for us to utilize whereas with our dictionary object because it gives that dynamic view it means that technically you might think you're going to get one set of values and you end up with a different set.

But whenever you use copy then you have the ability to perform any kind of the actions that you want to do and you can be confident on the data that you're working with because it's exactly what you copied.

Now, these concepts like I already mentioned are very advanced.

Usually, you don't have to get into the concept of building thread-safe programs or anything like that until you get into very senior level development.

However the fact that if you ever do want to grab these values or keys or items you're going to see that dictionary object and I want to prepare you to be a professional developer.

And so if you are not aware of all the intricacies and at least have a high-level understanding of what a dictionary view object is then I wouldn't be preparing you the right way.

So let's take a look at some more detailed examples here with nested items.

So I'm just going to cut `print(player_names)`. I'll leave the `player_names = list(players.copy().values())` so that you can view it in the show notes.

Now let's talk about how we can work with nested items so if I want to do something like say team grouping's and I can set this equal to teams.items() and that is gonna give me a dictionary view object and it's inside of it's going to have tuples.

So let's print this out and just see what we have here.

So, now, you can see we have a** dictionary view object and inside of this is a set of these tuple items.**

So we have the Astro's and then we have Altuve, Correa, Bregman, etc.

So we have all of these grouped together one at a time.

And so there are some functions that we can call on this.

So if we reference the documentation again there are a few elements that we can perform with view objects we can't treat them just like lists or like plain tuples but there are a few helpful functions.

For example, if you simply want to see how many items are inside you can use the `len` function.

So right here if I do team groupings and then just type len even though this is a dictionary view object.

If I run this you can see that there are four elements which is exactly correct.

Now, we have not gotten into iterable's and looping yet but as you can see from this next function we can see that we actually do have the ability with a dictionary view object to loop through though so if you want to implement some type of looping mechanism that goes through and list them all out such as on a web page or a mobile app then you can work

large

Figure 11: large

large

Figure 12: large

large

Figure 13: large

with a for loop in all of those type of iterable that will get into later on in the course and dictionary view objects are good for that.

So, that is a very important thing to keep in mind. You're not going to implement it right now we're going to wait till we get into our iteration section.

But we do have a few processes that we can run on them.

Now, to finish off let's see how we can access some of these values because this is going to give us some practice on going through multiple nested collections because as you can see right here we are going to start with the dictionary view object.

Let's convert that first to a list and so now if I print this out we have our list.

But now this is a list that contains tuples inside of it so just because I know it's kind of hard to read right here.

I'm just going to let me copy all of this and put it in a multiline string.

Just so we can see all of it and we can practice how to iterate through it.

So here we're going to have a list and you give some space just so we can see each one of them on their own line.

So we have Astro's we have angels Yankees and then red sox.

But it's using this different kind of structure and don't let the parens confuse you.

We're going to get into tuples later.

But for right now let's take a little preview on how we can work with these.

So if I say list so I cast team groupings this view object as a list and let's see if we can grab the element with an index of one now and we can.

Now we have the Angels Trout and Pujols.

Now we can traverse further down the line.

So, let's say that we want to get the nested collection I can add more brackets and I can chain these together and because this is a tuple which has I'll give you a little bit of a spoiler alert you can treat tuple elements very similar to how you can treat elements inside of the list.

So, if I want to grab the index one this is going to grab me this collection.

So now if I run this again you can see now we have Trout and Pujols

and just for fun let's chain on one more element and also so that you can see that you can pass these chained elements in and these chained lookups just right next to each other. And so now if I run it again you can see we have trout.

So if you ever have a data collection like this where you have a dictionary and you have all of these elements here and you want to grab one of the nested items this is the type of traversal you can perform. So in review what we did was we cast this dictionary view object as a list. Once it was a list we were able to treat it that way we're able to go and grab the values now inside of this list. We wanted to grab the angels and we knew they were index 1 so we added a 1 inside of the square brackets. Then we wanted to get to the player's array. So we knew that was in the index 1 inside of angels. So here we grabbed the player's array and then from there we wanted the first player trout. So we passed in because it's a list we passed in with the square brackets a 0 and that brought back trout.

So we covered a lot in this guide and I know this was a long one but it is a little bit harder to break this one up into smaller pieces because there are many interconnected kinds of concepts.

IMG

Figure 14: IMG

So in review we talked about dictionary view objects. We also reviewed where you could access that inside of the Python 3 documentation talked a little bit about thread safety and what that means when it comes to working with views that to me change their values and how you can protect that by utilizing tools such as copying.

Then we also walk through some of the dictionary view object features such as being able to call values keys and items and then we saw how we could convert those into lists and then traverse them the same way as any other data structure. So very nice if you went through that. If this is still very vague and it's not making sense I highly recommend that you go through this guide a few times try out the examples that we have here. Mix and match them even more so that you can become familiar with what happens when you make certain changes go through the documentation for with the link I'll provide and the more you do that the more you practice. It's going to start to make more and more sense.

Code

```
players = {
 "ss" : "Correa",
 "2b" : "Altuve",
 "3b" : "Bregman",
 "DH" : "Gattis",
 "OF" : "Springer",
}
player_names = list(players.copy().values())
teams = {
 "astros" : ["Altuve", "Correa", "Bregman"],
 "angels": ["Trout", "Pujols"],
 "yankees": ["Judge", "Stanton"],
 "red sox": ["Price", "Betts"],
}
team_groupings = teams.items()

"""
[
 ('astros', ['Altuve', 'Correa', 'Bregman']),
 ('angels', ['Trout', 'Pujols']),
 ('yankees', ['Judge', 'Stanton']),
 ('red sox', ['Price', 'Betts'])
]
"""

print(list(team_groupings)[1][1][0])
```

Resources

- Dictionary View Object Documentation