Data Structures

Monday, December 13, 2021 3:20 PM

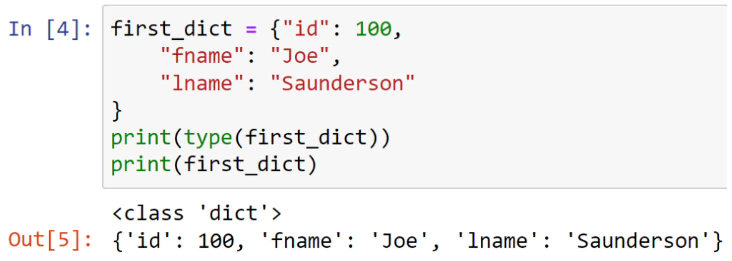
DICTIONARY

A standard data structure is called an array or a map. In Python this is referred to as a dictionary and are used when you have key value pairs of data to store.

Using dictionaries we can store data in a single object.
To create a dictionary we use curly brackets `{}` to which we store the key-pair names followed by a colon `:`. Each key-value pair must be separated by a comma `,`. The name of each key-value pair must also be contained within double quotes`""`.

Dictionaries are capable of holding different data types due to Pythons flexibility and can use any data type for keys and values, including other structures of data such as tuples which we will look at shortly.

The following is an example of a basic dictionary:

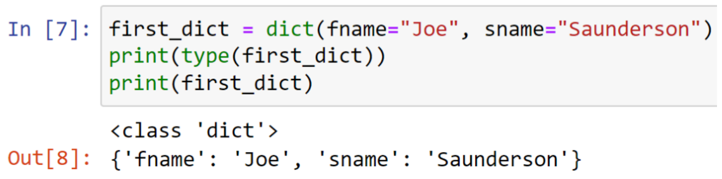


```
In [4]:  first_dict = {"id": 100,
             "fname": "Joe",
             "lname": "Saunderson"
         }
         print(type(first_dict))
         print(first_dict)

         <class 'dict'>
Out[5]:  {'id': 100, 'fname': 'Joe', 'lname': 'Saunderson'}
```

Look at the code as if it was a map, there are 3 inputs and outputs. In Python, the inputs are called keys and the outputs are called values, hence the term "key-value". Notice that this dictionary contains two data types, an integer and two strings.

Using the `type()` method we can also confirm that first_dict is an instance of the dictionary class.

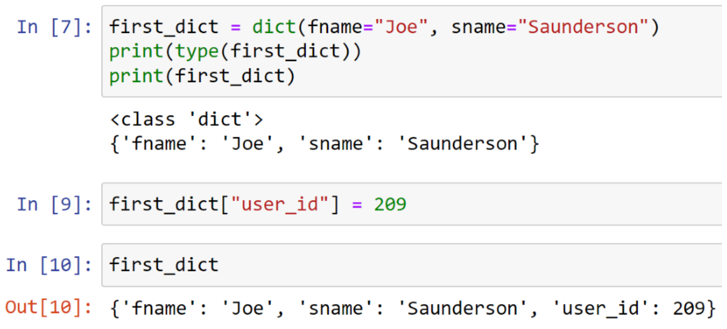
Additionally we can create a dictionary by casting much like a variable which we explored earlier in this guide, see the below example:



```
In [7]:  first_dict = dict(fname="Joe", sname="Saunderson")
         print(type(first_dict))
         print(first_dict)

         <class 'dict'>
Out[8]:  {'fname': 'Joe', 'sname': 'Saunderson'}
```

Like before we can use the `type()` method to confirm that first_dict is indeed an instance of the dictionary class.

You can also add additional data to a dictionary by specifying the dictionary name followed by square brackets. The key value goes inside the square brackets between double quotes and the value goes outside the square brackets, see example:



```
In [7]:  first_dict = dict(fname="Joe", sname="Saunderson")
         print(type(first_dict))
         print(first_dict)

         <class 'dict'>
         {'fname': 'Joe', 'sname': 'Saunderson'}

In [9]:  first_dict["user_id"] = 209

In [10]: first_dict

Out[10]: {'fname': 'Joe', 'sname': 'Saunderson', 'user_id': 209}
```

You can access data in a dictionary by using the `print()` method, then putting the dictionary name inside the parenthesis and then specifying the key name inside the square brackets that come after, see this following example:

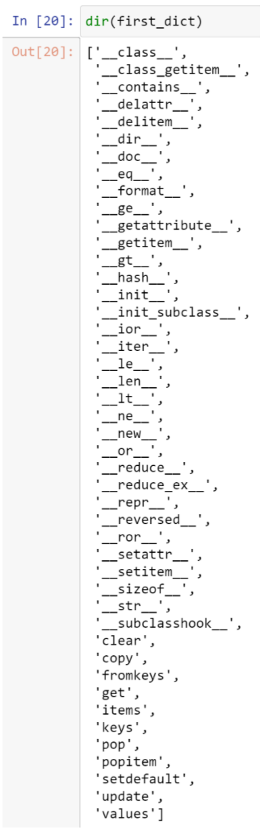
```
In [13]: first_dict = dict(fname="Joe", sname="Saunderson")
         print(type(first_dict))
         print(first_dict)

         <class 'dict'>
Out[10]: {'fname': 'Joe', 'sname': 'Saunderson'}
```

```
In [15]: print(first_dict["fname"])

Out[10]: Joe
```

When trying to access data that is not in a dictionary, the Python interpreter will return a key error. A way around this is to use the in operator to check if the key is in the dictionary. We can write a simple if statement with a condition that output this, see example:

```
In [17]: if "email" in first_dict:
             print(first_dict["email"])
         else:
             print("first_dict does not contain the key for email")

Out[18]: first_dict does not contain the key for email
```

There is another more advanced way to check if a dictionary contains a certain key-value pair or not, this is done using the `dir()` function. This function shows the directory for a dictionary which shows us a list of methods, see example:



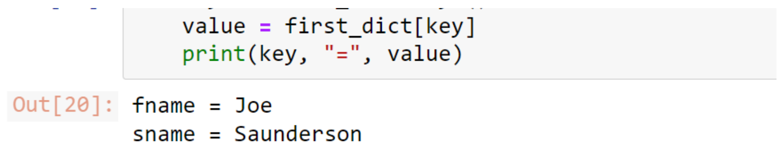
```
In [20]: dir(first_dict)

Out[20]: ['__class__',
          '__class_getitem__',
          '__contains__',
          '__delattr__',
          '__delitem__',
          '__dir__',
          '__doc__',
          '__eq__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__getitem__',
          '__gt__',
          '__hash__',
          '__init__',
          '__init_subclass__',
          '__ior__',
          '__iter__',
          '__le__',
          '__len__',
          '__lt__',
          '__ne__',
          '__new__',
          '__or__',
          '__reduce__',
          '__reduce_ex__',
          '__repr__',
          '__reversed__',
          '__ror__',
          '__setattr__',
          '__setitem__',
          '__sizeof__',
          '__str__',
          '__subclasshook__',
          'clear',
          'copy',
          'fromkeys',
          'get',
          'items',
          'keys',
          'pop',
          'popitem',
          'setdefault',
          'update',
          'values']
```

The get method is the one of interest. Using the `help()` function we can see what the get method does by simply putting the dictionary_name.get within the parenthesis. The get method allows you to try and get a value for a specific key and allows you to specify a default value should it not exist.

It is possible to iterate all the key-value pairs in a dictionary by using the keys method which we can see using the `dir()` function. A way to do this is use a for loop to get the value of each key, see example:

```
In [19]: first_dict = dict(fname="Joe", sname="Saunderson")
         print(type(first_dict))
         print(first_dict)

         <class 'dict'>
         {'fname': 'Joe', 'sname': 'Saunderson'}
```

```
In [21]: for key in first_dict.keys():
```

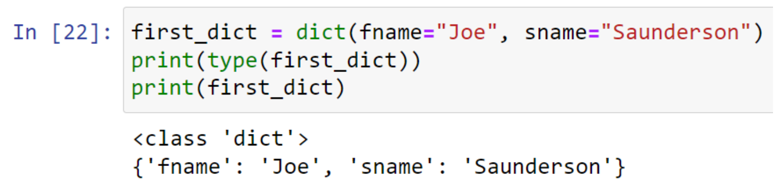


```
        value = first_dict[key]
        print(key, "=", value)
```

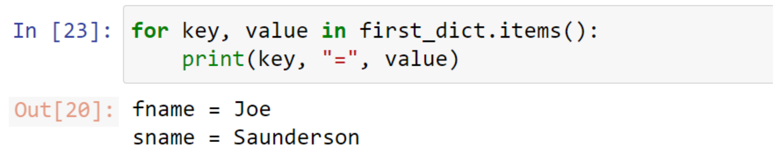
```
Out[20]: fname = Joe
         sname = Saunderson
```

Here we displaying the key-value pairs found within the first_dict dictionary using the .key method.

A more simple way to achieve the above is to use the `.items()` method which will give you both the key and the value, see example:



```
In [22]: first_dict = dict(fname="Joe", sname="Saunderson")
         print(type(first_dict))
         print(first_dict)

         <class 'dict'>
         {'fname': 'Joe', 'sname': 'Saunderson'}
```



```
In [23]: for key, value in first_dict.items():
             print(key, "=", value)

Out[20]: fname = Joe
         sname = Saunderson
```

As you can see the result is the same.

TUPLES

Tuples are similar to lists in the fact they are ordered sequences of data and share similar behavior. The difference is that tuples are immutable (meaning they cannot be changed) sequences of data and contain a sequence of data surrounded by parathesis instead of square brackets.

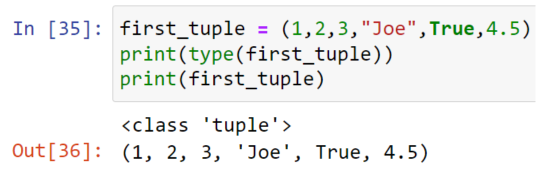
With this is mind, Python has made optimizations that allow tuples to be made more quickly than lists.

Both lists and tuples support the `len()` function which shows the length of each structure.

Tuples have a main difference in that they have much less methods available to them if we use the `dir()` function.

If we use the `sizeof()` method for both a list and tuple, we can discover that tuples occupy less memory than lists.

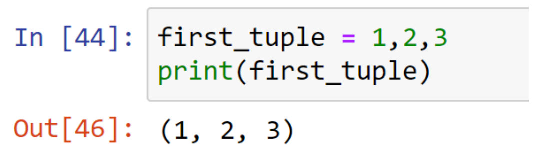
This is an example of a basic tuple:



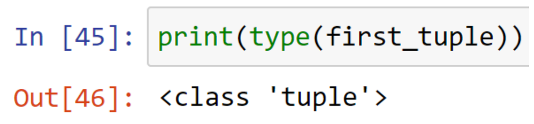
```
In [35]: first_tuple = (1,2,3,"Joe",True,4.5)
         print(type(first_tuple))
         print(first_tuple)

         <class 'tuple'>
Out[36]: (1, 2, 3, 'Joe', True, 4.5)
```

We use the `type()` function like we did with the dictionary to confirm that first tuple is an instance of the tuple class.

There is another way to make a tuple and that is to leave out the parenthesis altogether, the next example shows this, we can like before use the `type()` function to confirm that even though it is written this way it is still an instance of the tuple class.



```
In [44]: first_tuple = 1,2,3
         print(first_tuple)

Out[46]: (1, 2, 3)
```



```
In [45]: print(type(first_tuple))

Out[46]: <class 'tuple'>
```

Additionally, you can create an empty tuple by simply using empty parenthesis and we can create some more tuples, this time with one tuple only containing one item, see example:

```
In [40]: first_tuple = ()
         tuple1 = ("a")
         tuple2 = ("a","b")
         tuple3 = ("a","b","c")

         print(first_tuple)
         print(tuple1)
         print(tuple2)
         print(tuple3)

Out[41]: ()
         a
         ('a', 'b')
         ('a', 'b', 'c')
```

Notice that the tuple containing one element is displayed like a string and not a tuple and that's because it is, tuple1 is a string. To get around this, a tuple that has only one element it is required to add a comma at the end, see the next example:

```
In [42]: first_tuple = ()
         tuple1 = ("a",)
         tuple2 = ("a","b")
         tuple3 = ("a","b","c")

         print(first_tuple)
         print(tuple1)
         print(tuple2)
         print(tuple3)

Out[41]: ()
         ('a',)
         ('a', 'b')
         ('a', 'b', 'c')
```

Here we have updated tuple1 to have a comma at the end, now it is displayed as a tuple. The reason for the need of a comma at the end is due to tuple assignment.

Say for example you have a tuple that contains multiple pieces of data, accessing the data is done similarly to how it is done in lists, by indexing. Tuple however provide a faster way to do this by assigning all elements in a tuple to different variables in a single line, see example:

```
In [47]: first_tuple = (33, "Joe", "Destiny 2")
         age, name, likes = first_tuple

In [48]: print("Age =", age)
         print("Name =", name)
         print("Likes =", likes)

Out[41]: Age = 33
         Name = Joe
         Likes = Destiny 2
```

As you can see, the first element is assigned to age, the second to name and the third to likes which the Python interpreter assigns the values for you. We can then print each value to confirm.