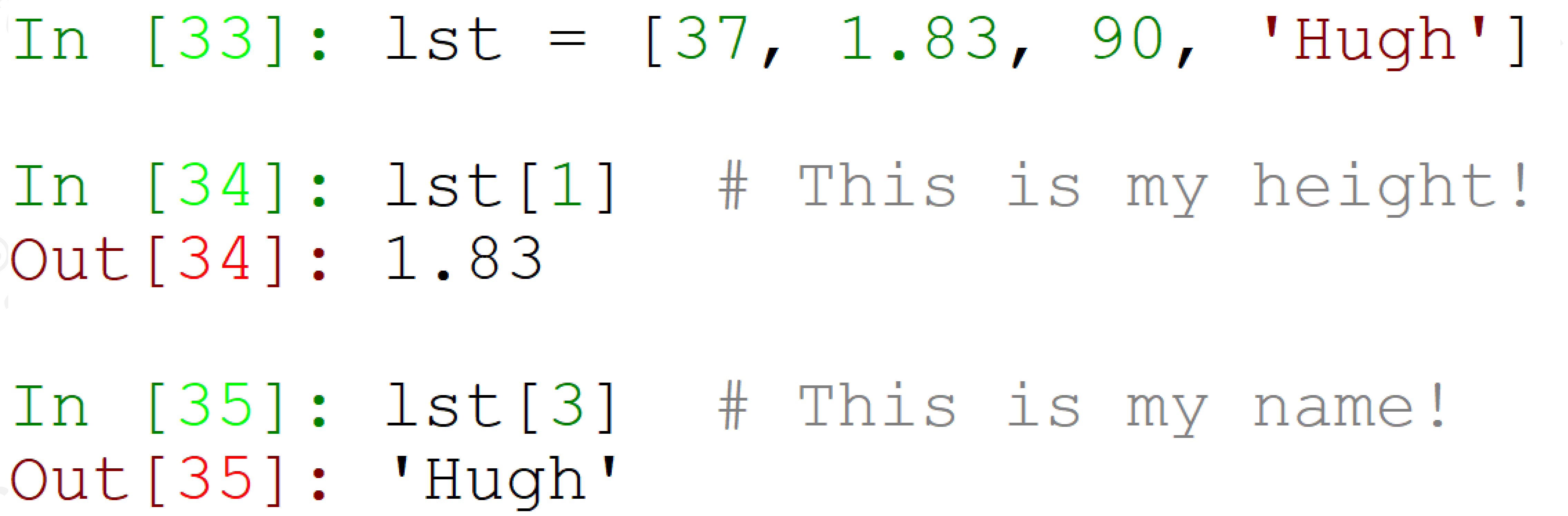
Module 5.4.2: Dictionaries

Lists store values by index.



```
In [33]: lst = [37, 1.83, 90, 'Hugh']

In [34]: lst[1]   # This is my height!
Out[34]: 1.83

In [35]: lst[3]   # This is my name!
Out[35]: 'Hugh'
```

Pringles vs. Suitcase

Lists are like a jar of Pringles –

Dictionaries are like a suitcase! It is filled with values that have ***names***.

Our Suitcase – Values Have Names!

Creating a Dictionary

Dictionaries are created with curly brackets!

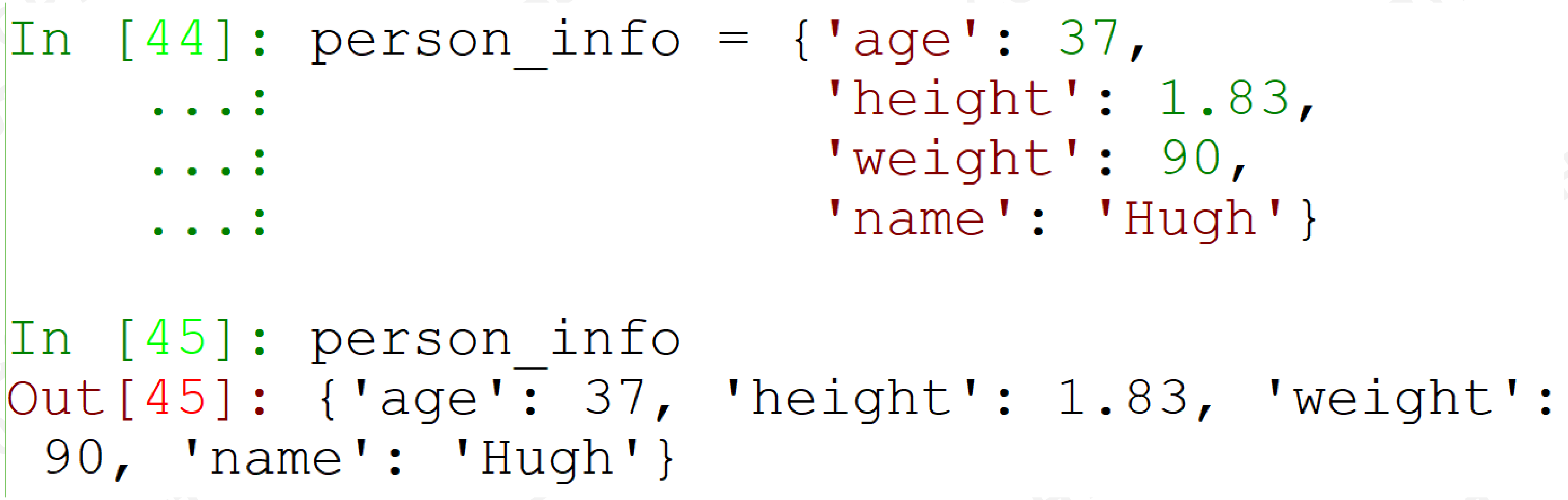
```
In [42]: empty_dict = {}

In [43]: dict_with_one_val = {'key': 'value'}
```

Every new item in the dictionary has a:

- ***key*** – the name of the value
- ***value*** – the value itself

The key and value are separated by a colon.

Now let's save our info again!

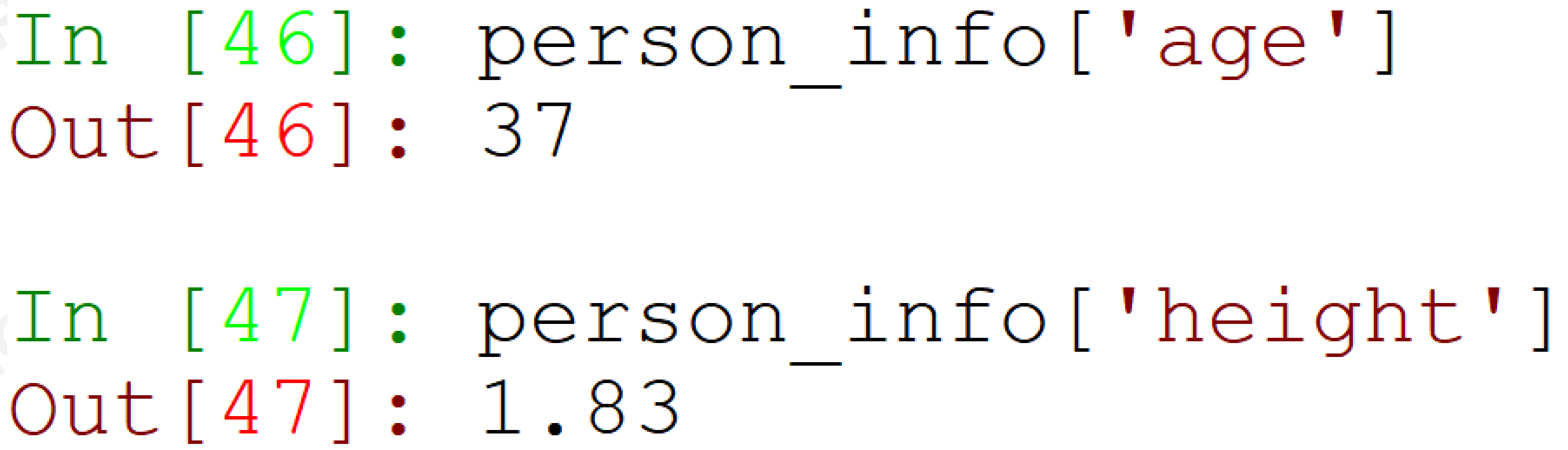


```
In [44]: person_info = {'age': 37,
    ...:                'height': 1.83,
    ...:                'weight': 90,
    ...:                'name': 'Hugh'}

In [45]: person_info
Out[45]: {'age': 37, 'height': 1.83, 'weight':
 90, 'name': 'Hugh'}
```

As you can see, order has no importance in a dictionary, because the index is not counted, but named!

Getting a Value from a Dictionary

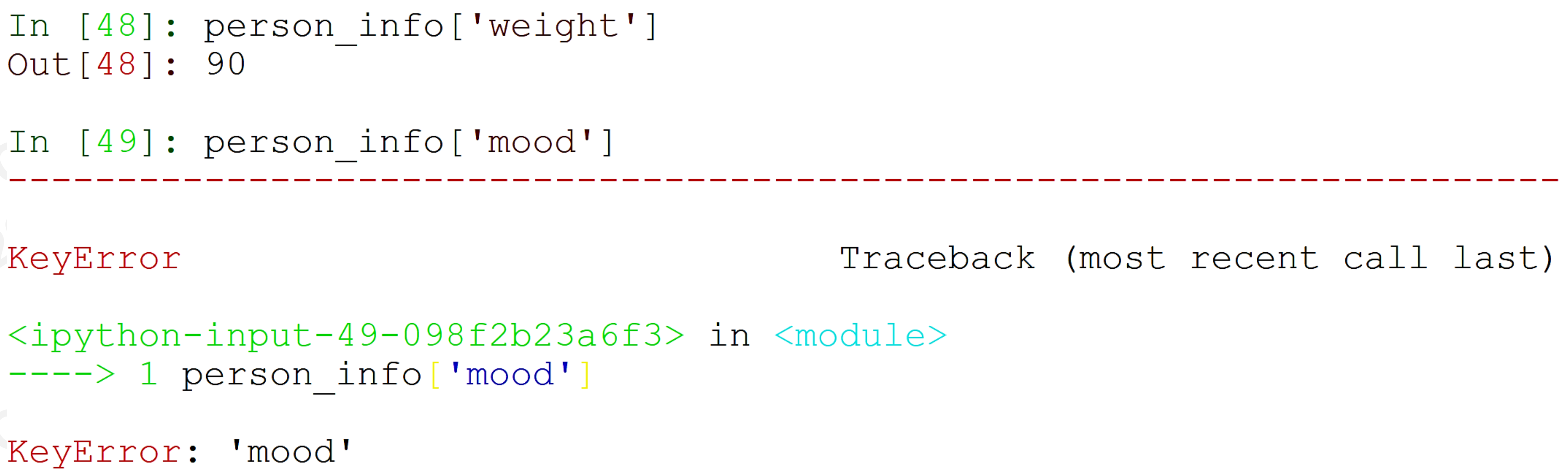
The value of a key can be fetched by using square brackets (like in lists!), but this time with the key name, not the index ID.



```
In [46]: person_info['age']
Out[46]: 37

In [47]: person_info['height']
Out[47]: 1.83
```

What if there is no such key?

A ***KeyError*** is raised! It means that no such key was found in the dictionary.

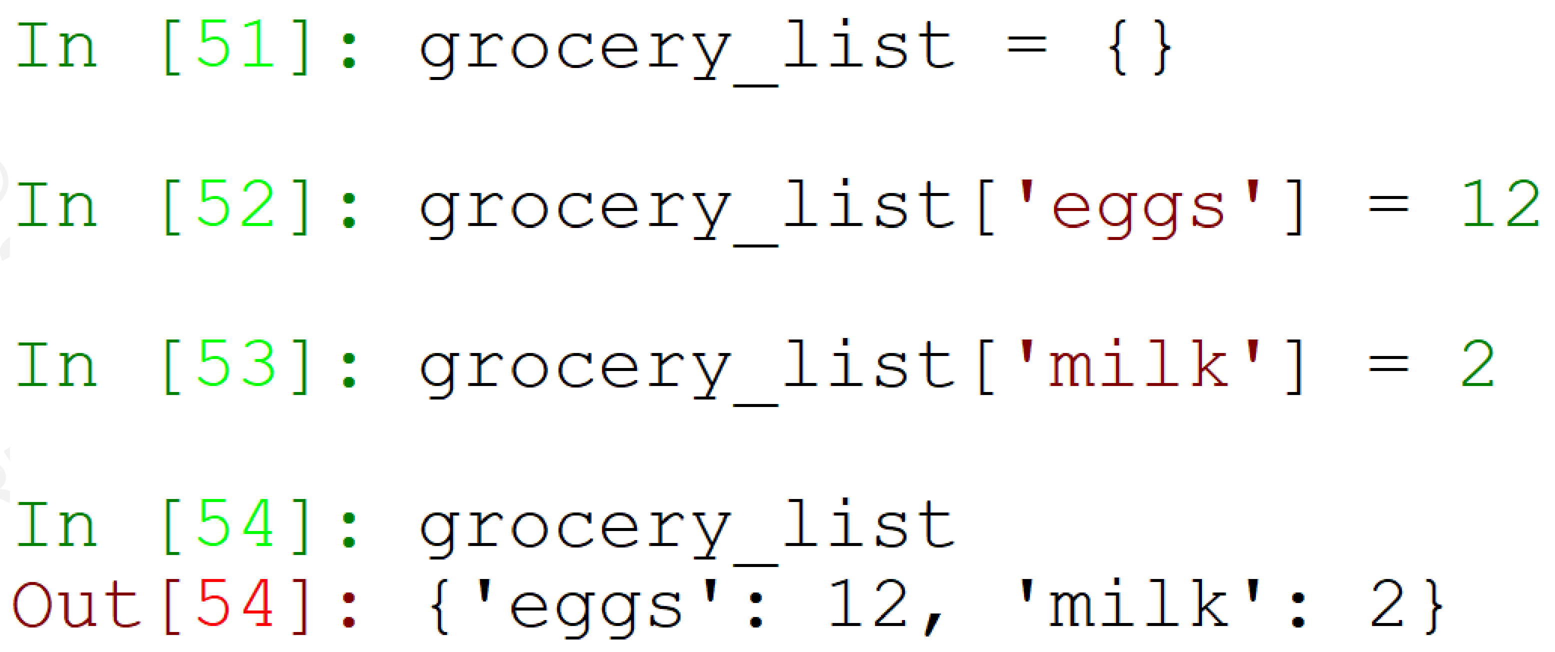


```
In [48]: person_info['weight']
Out[48]: 90

In [49]: person_info['mood']
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-49-098f2b23a6f3> in <module>
----> 1 person_info['mood']

KeyError: 'mood'
```

Adding Key/Values to a Dictionary

A new key/value pair can be added to the dictionary easily.

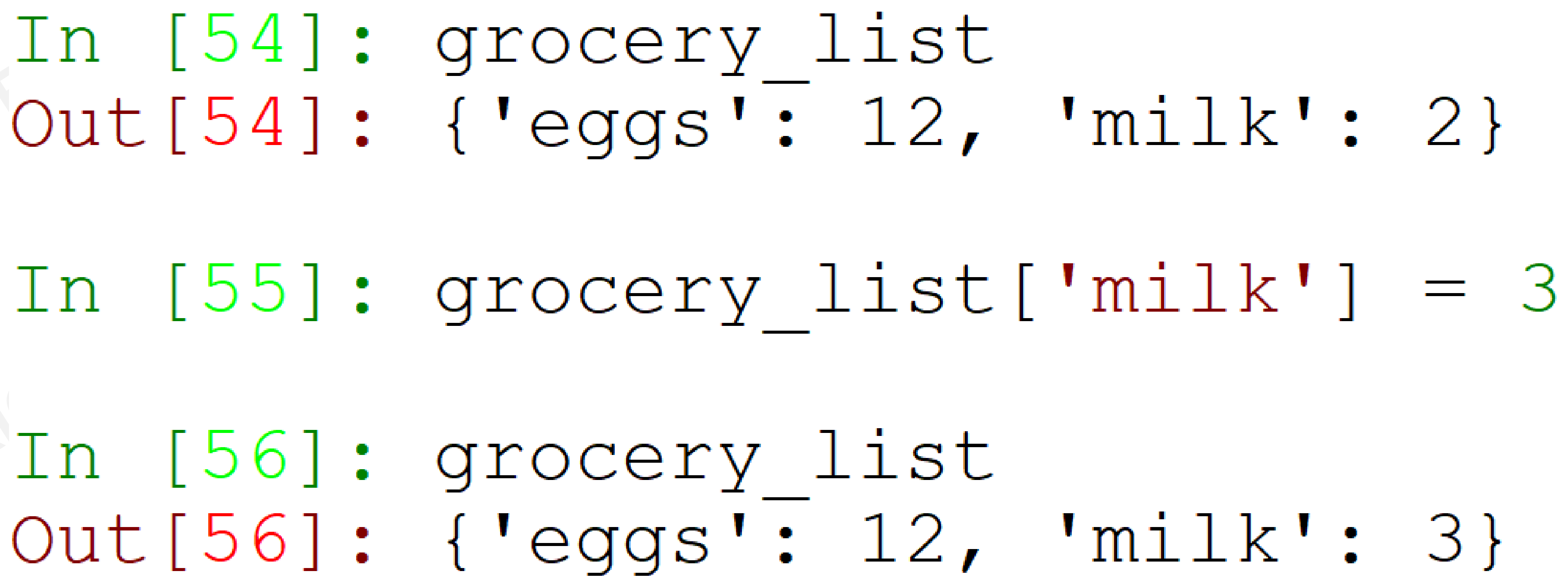
Use the key as an index, but this time assign it a value – just like creating variables!



```
In [51]: grocery_list = {}

In [52]: grocery_list['eggs'] = 12

In [53]: grocery_list['milk'] = 2

In [54]: grocery_list
Out[54]: {'eggs': 12, 'milk': 2}
```

Adding Existing Keys

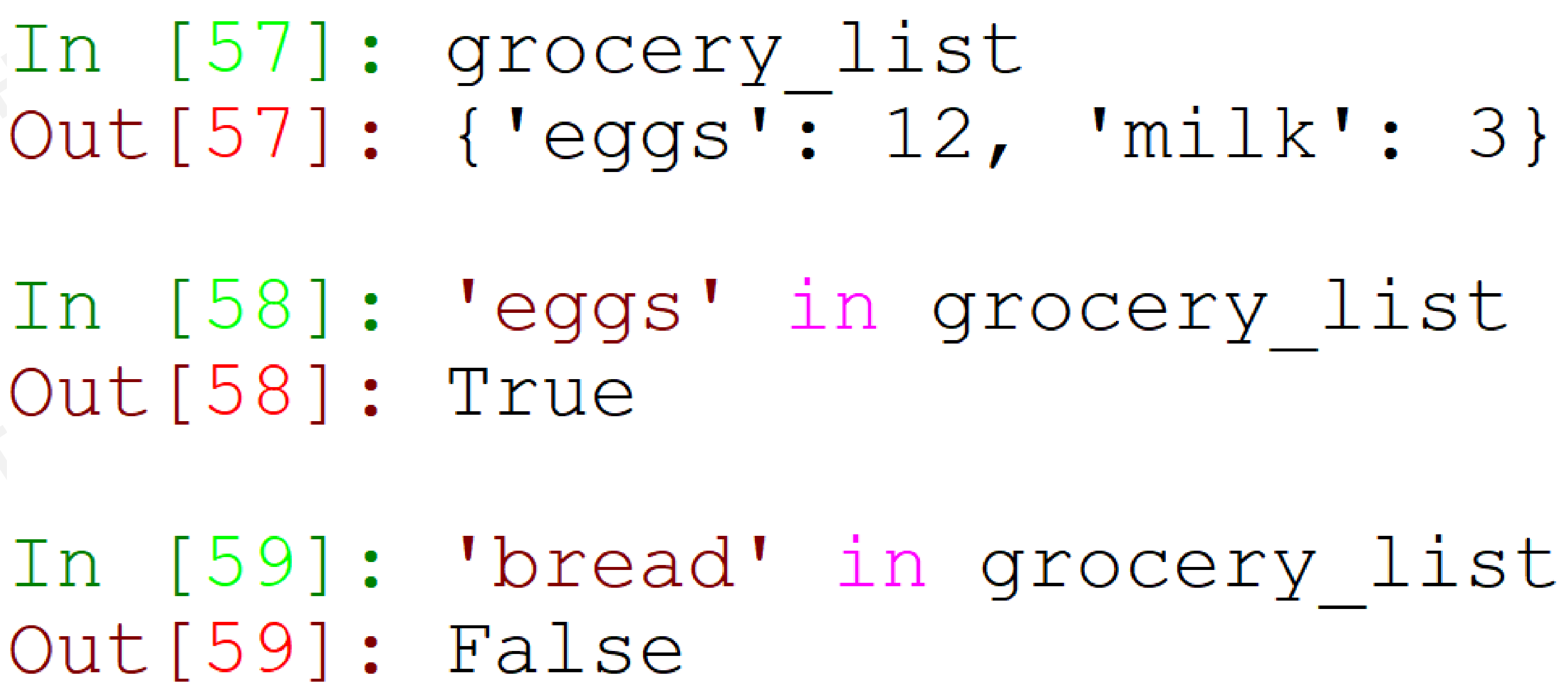
Just like in variables, assigning a new value **erases** the last value, and switches it with the new value.



```
In [54]: grocery_list
Out[54]: {'eggs': 12, 'milk': 2}

In [55]: grocery_list['milk'] = 3

In [56]: grocery_list
Out[56]: {'eggs': 12, 'milk': 3}
```

Checking if a Key is in a Dictionary

Just like with lists, we can use the ***in*** keyword:

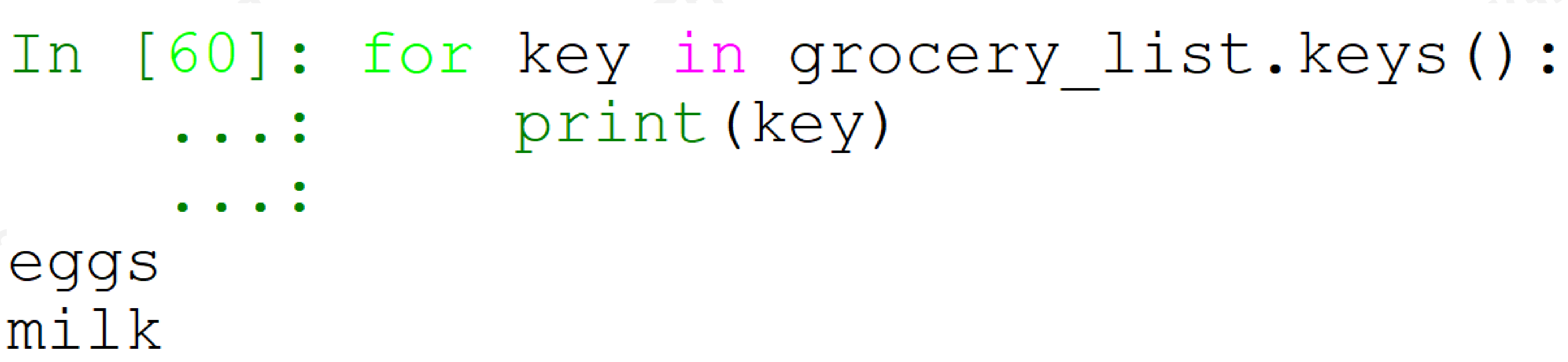


```
In [57]: grocery_list
Out[57]: {'eggs': 12, 'milk': 3}

In [58]: 'eggs' in grocery_list
Out[58]: True

In [59]: 'bread' in grocery_list
Out[59]: False
```

Dictionary Methods

dict.keys()

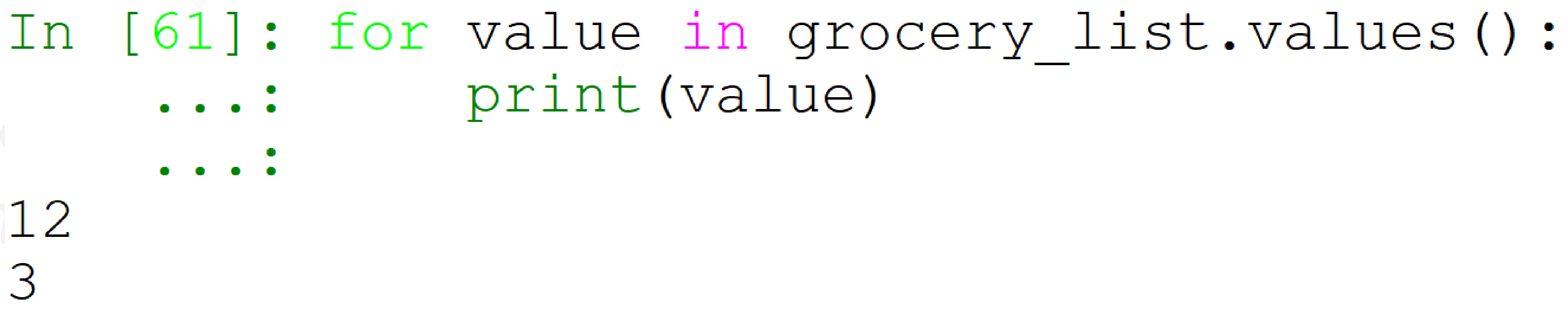
The ***keys*** method returns a sequence of all keys in the dictionary:



```
In [60]: for key in grocery_list.keys():
    ...:     print(key)
    ...:
eggs
milk
```

`dict.values()`

The ***values*** method returns a sequence of all values in the dictionary:

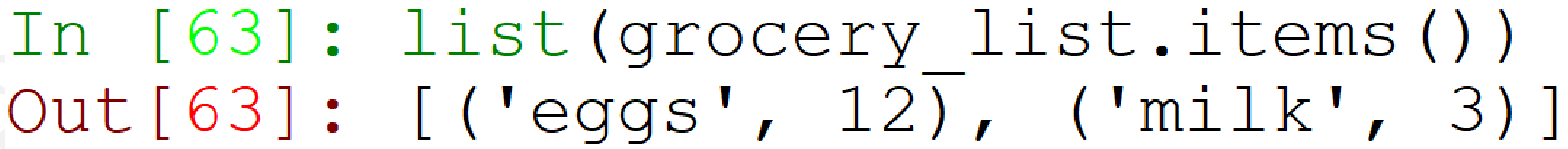


```
In [61]: for value in grocery_list.values():
    ...:     print(value)
    ...:
12
3
```

dict.items()

The ***items*** method returns a sequence of *tuples*.

Each tuple is built up of *(key, value)*.

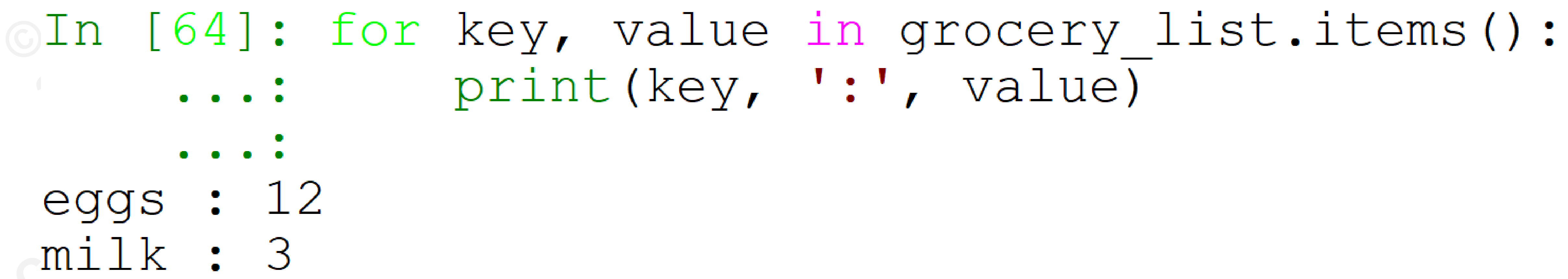


```
In [63]: list(grocery_list.items())
Out[63]: [('eggs', 12), ('milk', 3)]
```

Unpacking `dict.items()`

When each list item is a *tuple of length 2*, we can use unpacking in our for loop!

This means that each iteration in our loop assigns 2 values – one to *key* and one to *value*.



```
In [64]: for key, value in grocery_list.items():
    ...:     print(key, ':', value)
    ...:
eggs : 12
milk : 3
```

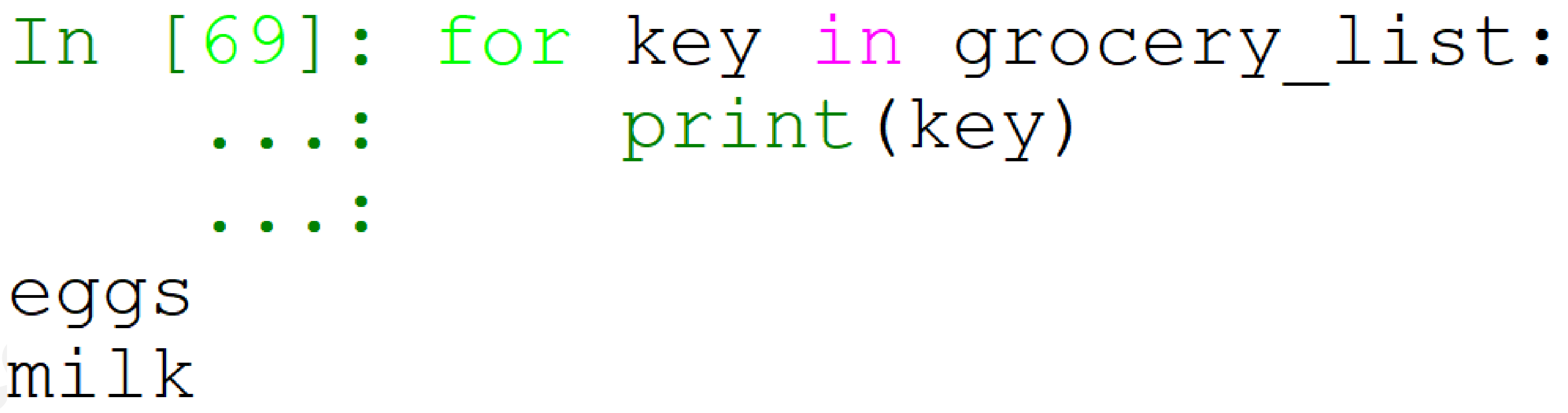
Using a *for* loop with no method

What happens if we just use a ***for*** loop on a dictionary itself?

Well, what happened when we used the ***in*** keyword?

It looked in the dictionary’s keys!

So it should do the same…



```
In [69]: for key in grocery_list:
    ...:     print(key)
    ...:
eggs
milk
```

dict.pop()

The *pop* method:

1. Receives a key
2. Removes that key from the dictionary
3. Returns the key's value

```
In [65]: grocery_list
Out[65]: {'eggs': 12, 'milk': 3}

In [66]: grocery_list.pop('milk')
Out[66]: 3

In [67]: grocery_list
Out[67]: {'eggs': 12}
```

Are there other dictionary methods?

Of course!

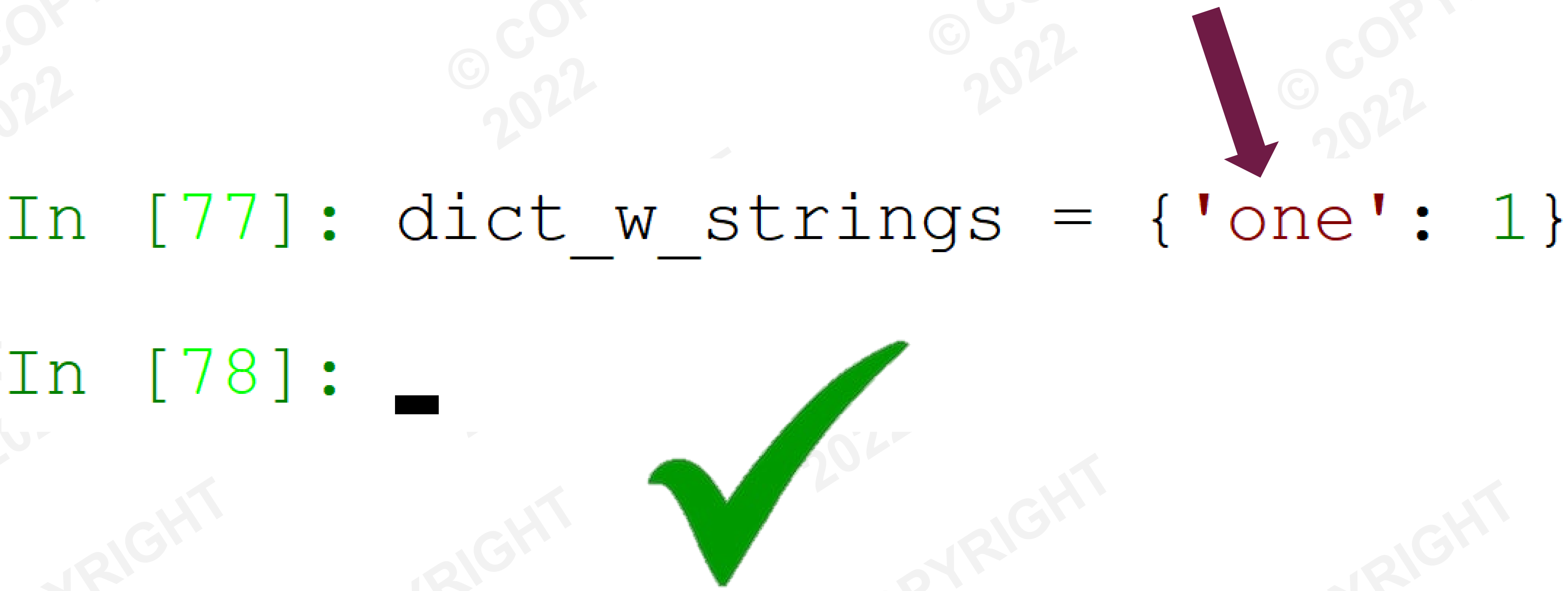
Use *`dir(dict)`* to find them.

Read about them with *`help(dict.method)`*, *?*, or in the Python Documentation!

What types can we use as dictionary keys?

Let’s try!

Strings?



```
In [77]: dict_w_strings = {'one': 1}

In [78]: _
```

Integers?

```
In [78]: dict_w_ints = {1: 'one'}

In [79]:
```

✓

Floating Points?

```
In [79]: dict_w_floats = {0.5: 'half'}

In [80]: _
```

✓

Tuples?

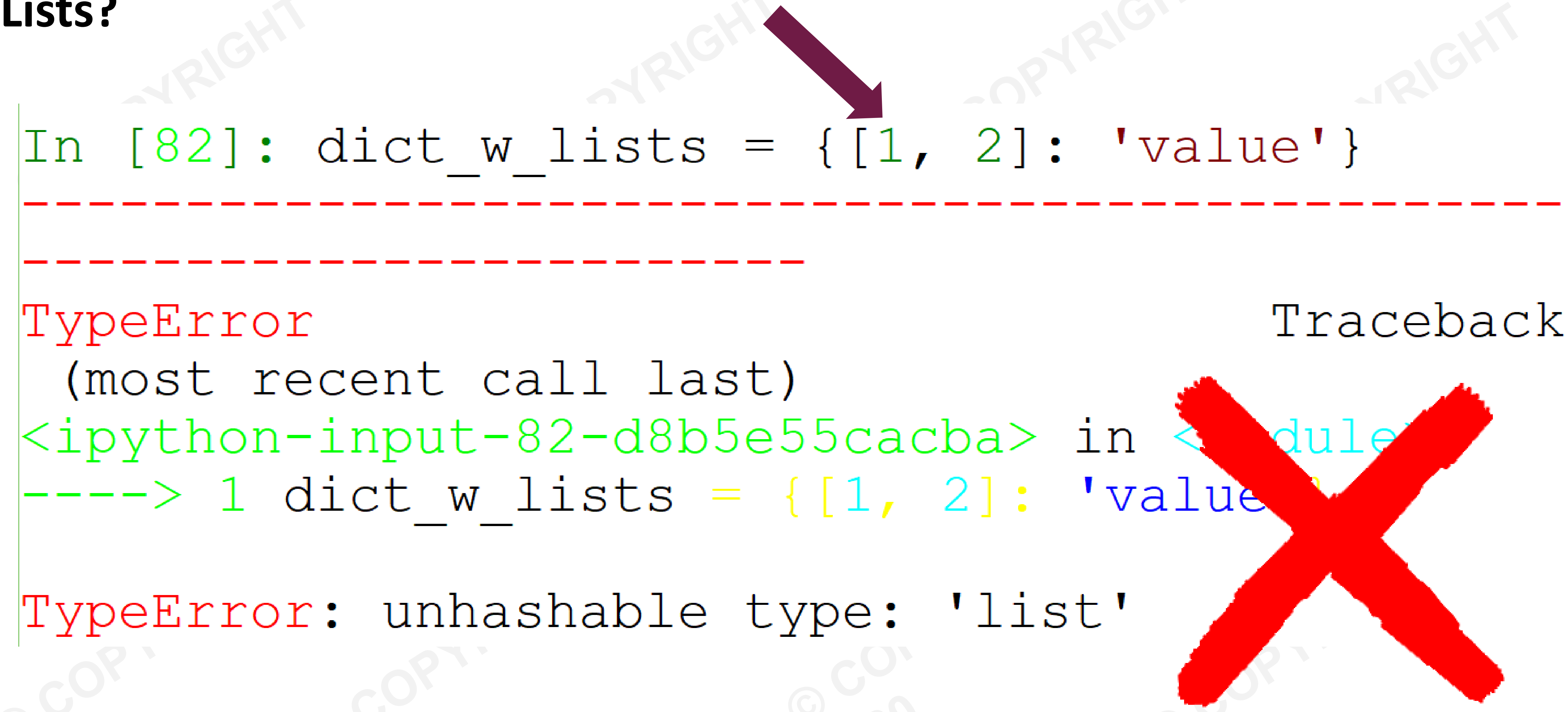
```
In [80]: dict_w_tuples = {(0, 1): True}

In [81]: 
```

✓

Booleans?

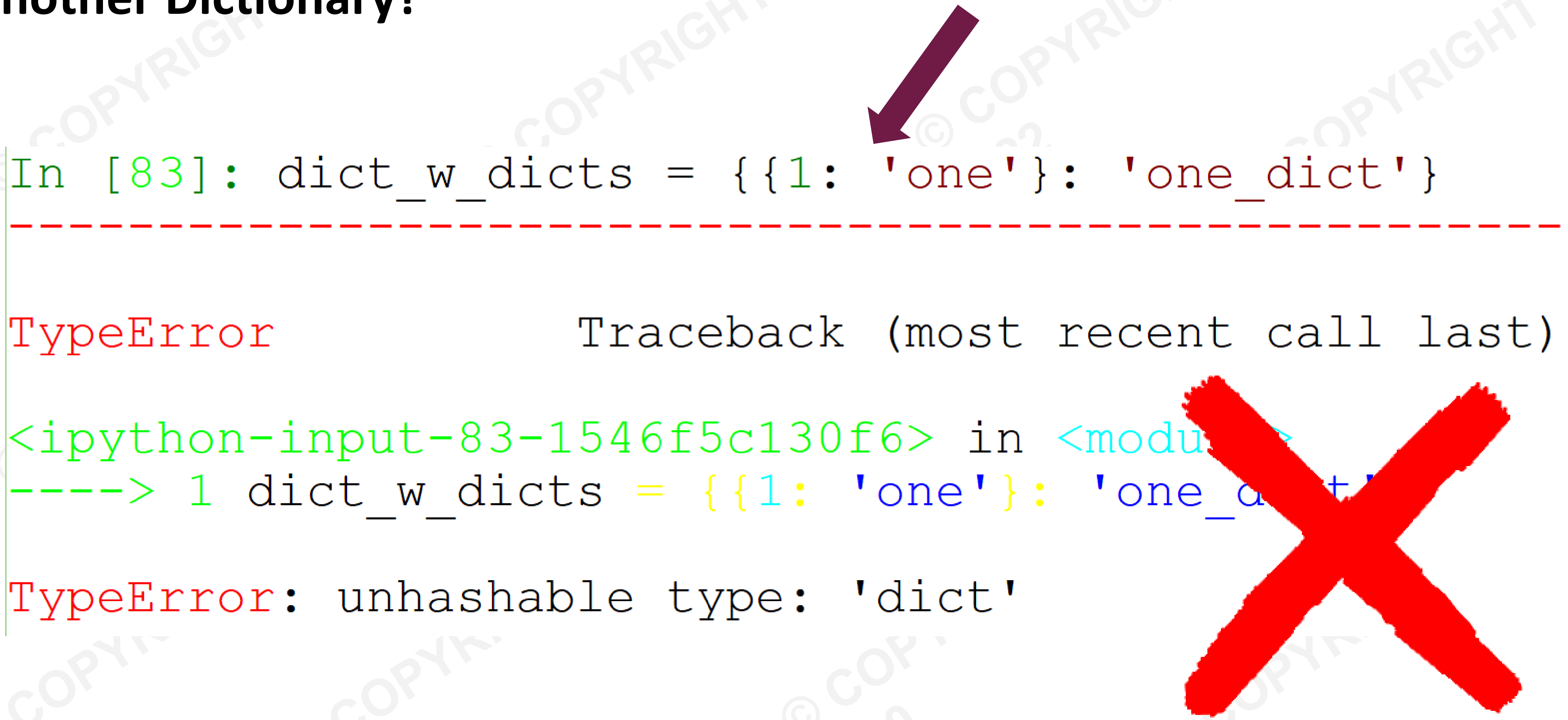
```
In [81]: dict_w_bools = {True: 1}

In [82]: _
```

Lists?



```
In [82]: dict_w_lists = {[1, 2]: 'value'}
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-82-d8b5e55cacba> in <module>
----> 1 dict_w_lists = {[1, 2]: 'value'}

TypeError: unhashable type: 'list'
```

Another Dictionary?



```
In [83]: dict_w_dicts = {{1: 'one'}: 'one_dict'}
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-83-1546f5c130f6> in <modu
----> 1 dict_w_dicts = {{1: 'one'}: 'one_d
TypeError: unhashable type: 'dict'
```

So, what types can we use as dictionary keys?

What is the main difference between a ***list*** and a ***tuple***?

A list is ***mutable*** – it can be changed!

Do we want our keys to be able to change in the middle of a run? No!

If they change, we will not be able to find our values!

So, dictionary keys can only be ***immutable***.

Is a dictionary mutable or immutable?

A dictionary is **mutable**!

So, it can be changed, just like a list.

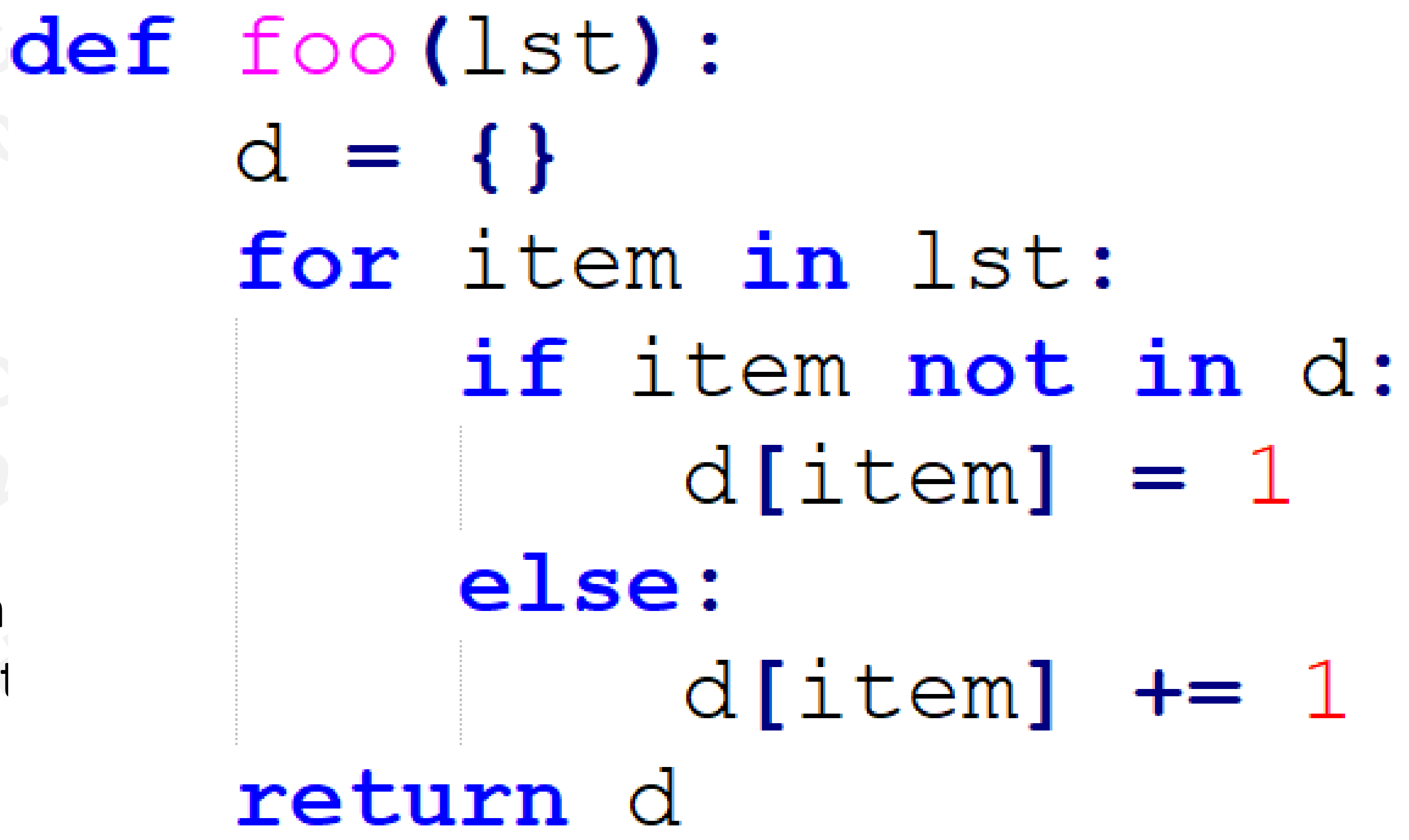
So, be careful when you send dictionaries to functions that can change them!

Remember to use the ***`dict.copy()`*** method if it is needed.

What does this function do?

A Counter!

It counts the number of tim
each item appears in the list



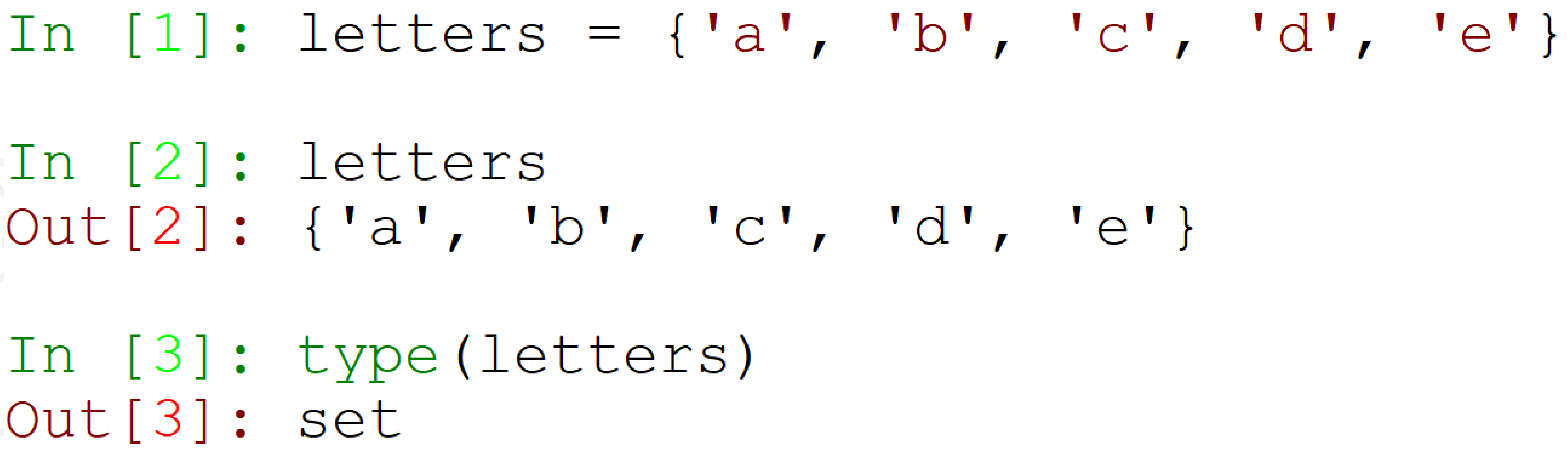
```
def foo(lst):
    d = {}
    for item in lst:
        if item not in d:
            d[item] = 1
        else:
            d[item] += 1
    return d
```

Sets

Advanced Data Structures

Creating a Set

Sets are a new sequence type, created by using curly brackets:



```
In [1]: letters = {'a', 'b', 'c', 'd', 'e'}

In [2]: letters
Out[2]: {'a', 'b', 'c', 'd', 'e'}

In [3]: type(letters)
Out[3]: set
```

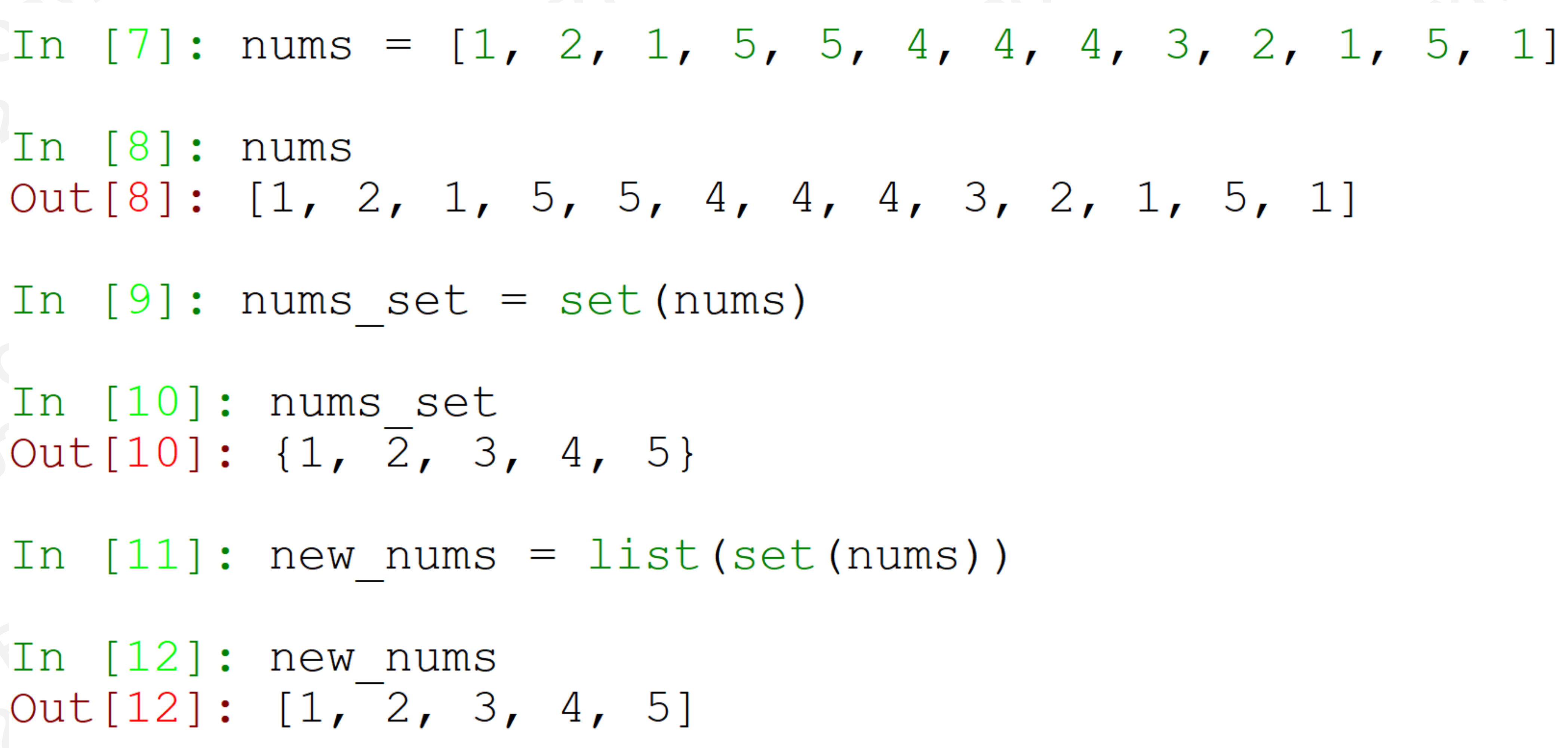
Set Qualities

Sets have two main qualities:

- Sets don’t have any specific **order**.
    - They don’t even support indexing!
- Sets hold only **distinct** values.
    - This means that there are no duplicate values!

```
In [5]: letters = {'a', 'a', 'b', 'a', 'b', 'c', 'a'}

In [6]: letters
Out[6]: {'a', 'b', 'c'}
```

Creating a Distinct List



```
In [7]: nums = [1, 2, 1, 5, 5, 4, 4, 4, 3, 2, 1, 5, 1]

In [8]: nums
Out[8]: [1, 2, 1, 5, 5, 4, 4, 4, 3, 2, 1, 5, 1]

In [9]: nums_set = set(nums)

In [10]: nums_set
Out[10]: {1, 2, 3, 4, 5}

In [11]: new_nums = list(set(nums))

In [12]: new_nums
Out[12]: [1, 2, 3, 4, 5]
```

be

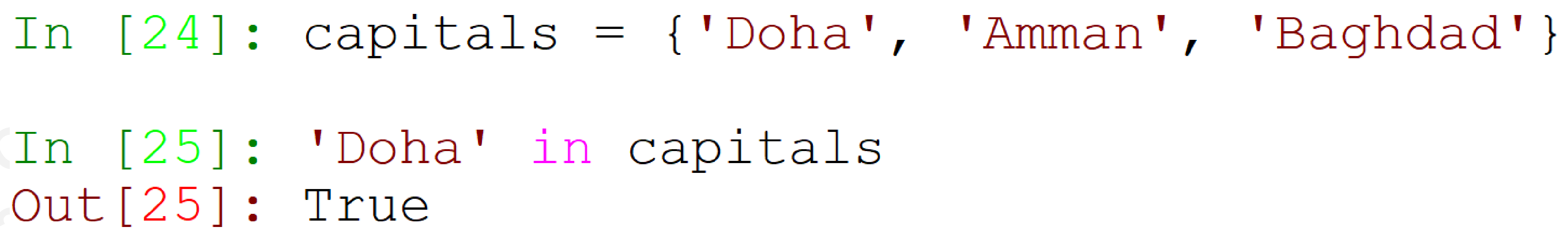
Let's Create a *unique* Function!

```
def unique(lst):
    return list(set(lst))
```

```
In [18]: unique([1, 'a', 1, 1, 'a', 'a'])
Out[18]: ['a', 1]

In [19]: unique([1, 1, 1.5, 1.5, -111, -111])
Out[19]: [1, -111, 1.5]
```

Using the *in* Keyword

Just like in lists, tuples, and dicts – using the ***in*** keyword can check if an item is in the set.



```
In [24]: capitals = {'Doha', 'Amman', 'Baghdad'}

In [25]: 'Doha' in capitals
Out[25]: True
```

Set Methods and Operators

Method Name	Explanation	Example
add	Adds a new value to the set	{1, 2, 3}.add(4)
remove	Removes a value from the set	{1, 2, 3}.remove(2)
intersection	Returns a set with the items in the set that appear in the second set.	{1, 3, 5}.intersection({3, 4, 5}) >>> {3, 5}

set1 – set2	Returns a set with the items in set1 that don't appear in set2.	{1, 3, 5} – {3, 4, 5}
		>>> {1}

issubset	Returns True/False, depending on if all items in the set appear in the second set.	{0, 4, 7}.issubset(set(range(10))) >>> True

Mutable vs. Immutable Types

Mutable	Immutable
list	int
dict	float
set	str
	tuple
	bool

What did we learn?

- Pointers
- Mutable vs. Immutable Types
- Downside of sending lists to functions
- Dictionaries
- Sets