Python Dictionary - The Ultimate Guide

Photo by Pisit Heng on Unsplash

Python comes with several built-in data types. These are the foundational building blocks of the whole language. They have been optimised and perfected over many years. In this article, we will explore one of the most important: the dictionary (or dict for short).

Unless stated otherwise I will be using Python 3.8 throughout. Dictionary functionality has changed over the last few Python versions. If you are using a version other than 3.8, you will probably get different results to me.

To check what version of Python you are running, enter the following in a terminal window (mine returns 3.8).

```
$ python --version
Python 3.8.0
```

Python Dictionary - Why Is It So Useful?

When I first found out about dictionaries, I wasn't sure if they were going to be very useful. They seemed a bit clunky and I felt like lists would be much more useful. But boy was I wrong!

In real life, a dictionary is a book full of words in alphabetical order. Beside each word is a definition. If it has many meanings, there are many definitions. Each word appears exactly once.

- A book of words in alphabetical order.
- Each word has an associated definition
- If a word has many meanings, it has many definitions
- As time changes, more meanings can be added to a word.
- The spelling of a word never changes.
- Each word appears exactly once.
- Some words have the same definition.

If we abstract this idea, we can view a dictionary as a mapping from a word to its definition. Making this more abstract, a dictionary is a mapping from something we know (a word) to something we don't (its definition).

You can also view a dictionary as a one-way bridge. It connects you from one side (key) to the other (value).
Photo by David Martin on Unsplash

We apply this mapping all the time in real life:

- In our phone, we map our friends' names to their phone numbers
- In our minds, we map a person's name to their face
- We map words to their meaning

This 'mapping' is really easy for humans to understand and makes life much more efficient. We do it all the time without even realising. Thus it makes sense for Python to include this as a foundational data type.

Python Dictionary Structure

A traditional dictionary maps words to definitions. Python dictionaries can contain any data type, so we say they map keys to values. Each is called a key-value pair.

The key 'unlocks' the value. A key should be easy to remember and not change over time. The value can be more complicated and may change over time.

We will now express the same list as above using Python dictionary terminology.

- Python dictionary is a collection of objects (keys and values)
- Each key has an associated value
- A key can have many values
- As time changes, more values can be added to a key (values are mutable)
- A key cannot change (keys are immutable)
- Each key appears exactly once
- Keys can have the same value

Note: we can order dictionaries if we want but it is not necessary to do so. We'll explain all these concepts in more detail throughout the article. But before we do anything, we need to know how to create a dictionary!

Python Create Dictionary

There are two ways to create a dictionary in Python:

1. Curly braces { }
2. The dict() constructor

Curly Braces { }

```
my_dict = {key1: value1,
           key2: value2,
           key3: value3,
           key4: value4,
          key5: value5}
```

We write the key, immediately followed by a colon. Then a single space, the value and finally a comma. After the last pair, replace the comma with a closing curly brace.

You can write all pairs on the same line. I put each on a separate line to aid readability.

Let's say you have 5 friends and want to record which country they are from. You would write it like so (names and countries start with the same letter to make them easy to remember!).

```
names_and_countries = {'Adam': 'Argentina',
                       'Beth': 'Bulgaria',
                       'Charlie': 'Colombia',
                       'Dani': 'Denmark',
                       'Ethan': 'Estonia'}
```

The dict() Constructor

Option 1 - fastest to type

```
my_dict = dict(key1=value1,
               key2=value2,
               key3=value3,
               key4=value4,
               key5=value5)
```

So names_and_countries becomes

```
names_and_countries = dict(Adam='Argentina',
                           Beth='Bulgaria',
                           Charlie='Colombia',
                           Dani='Denmark',
                           Ethan='Estonia')
```

Each pair is like a keyword argument in a function. Keys are automatically converted to strings but values must be typed as strings.

Option 2 - slowest to type, best used with zip()

```
my_dict = dict([(key1, value1),
                (key2, value2),
                (key3, value3),
                (key4, value4),
                (key5, value5)])
```

names_and_countries becomes

```
names_and_countries = dict([('Adam', 'Argentina'),
                            ('Beth', 'Bulgaria'),
                            ('Charlie', 'Colombia'),
                            ('Dani', 'Denmark'),
                            ('Ethan', 'Estonia')])
```

Like with curly braces, we must explicitly type strings as strings. If you forget the quotes, Python interprets it as a function.

Option 2 with zip() - Python list to dict

If you have two lists and want to make a dictionary from them, do this

```
names = ['Adam', 'Beth', 'Charlie', 'Dani', 'Ethan']
countries = ['Argentina', 'Bulgaria', 'Colombia', 'Denmark', 'Estonia']

# Keys are names, values are countries
names_and_countries = dict(zip(names, countries))

>>> names_and_countries
{'Adam': 'Argentina',
'Beth': 'Bulgaria',
'Charlie': 'Colombia',
'Dani': 'Denmark',
'Ethan': 'Estonia'}
```

If you have more than two lists, do this

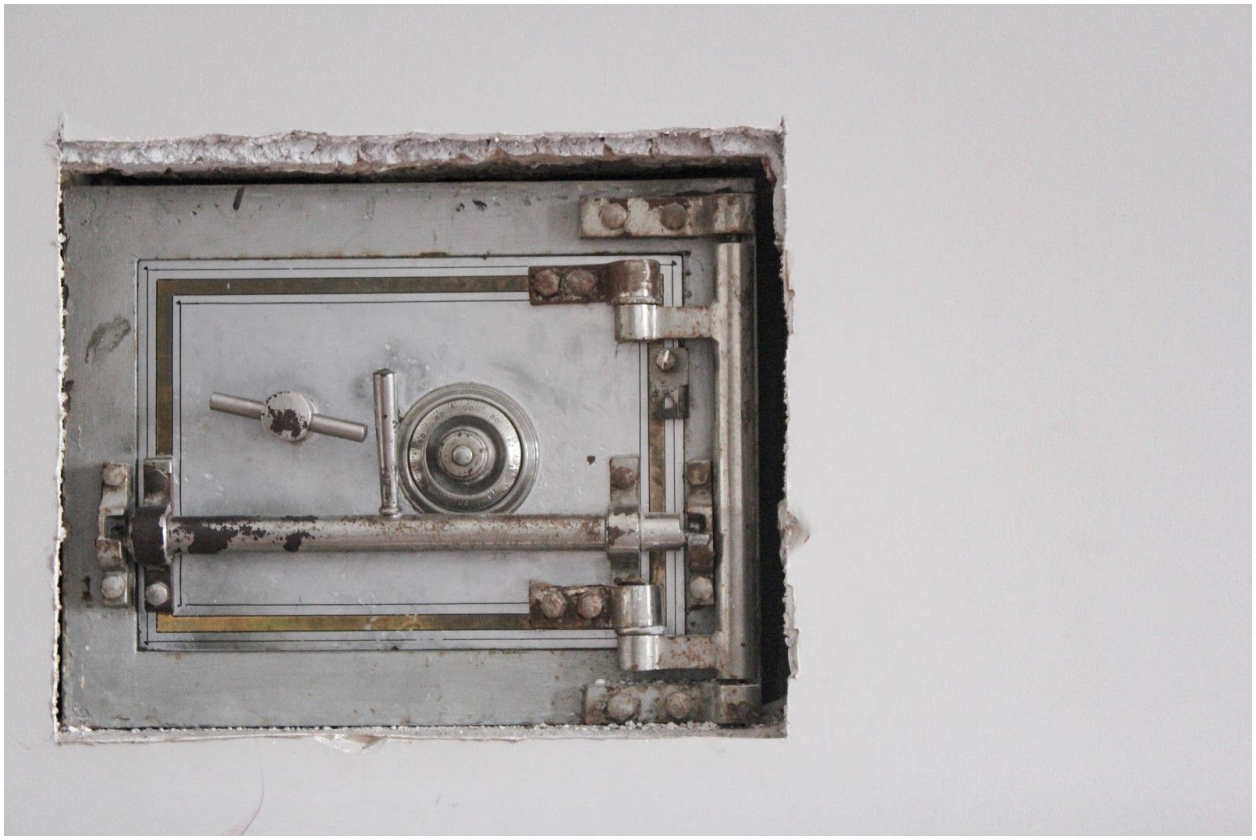
```
names = ['Adam', 'Beth', 'Charlie', 'Dani', 'Ethan']
countries = ['Argentina', 'Bulgaria', 'Colombia', 'Denmark', 'Estonia']
ages = [11, 24, 37, 75, 99]

# Zip all values together
values = zip(countries, ages)

# Keys are names, values are the tuple (countries, ages)
people_info = dict(zip(names, values))

>>> people_info
{'Adam': ('Argentina', 11),
'Beth': ('Bulgaria', 24),
'Charlie': ('Colombia', 37),
'Dani': ('Denmark', 75),
'Ethan': ('Estonia', 99)}
```

This is the first time we've seen a dictionary containing more than just strings! We'll soon find out what can and cannot be a key or value. But first, let's see how to access our data.

Accessing Key-Value Pairs

There are 2 ways to access the data in our dictionaries:

- Bracket notation [ ]
- The get() method



Let's learn how to unlock our dictionaries
Photo by Gabriel Wasylko on Unsplash

Bracket Notation [ ]

```
# Get value for the key 'Adam'
>>> names_and_countries['Adam']
'Argentina'

# Get value for the key 'Charlie'
>>> names_and_countries['Charlie']
'Colombia'

# KeyError if you search for a key not in the dictionary
>>> names_and_countries['Zoe']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Zoe'
```

Type the key into the square brackets to get the corresponding value. If you enter a key not in the dictionary, Python raises a KeyError.

This looks like list indexing but it is completely different! For instance, you cannot access values by their relative position or by slicing.

```
# Not the first element of the dictionary
>>> names_and_countries[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0

# Not the last element
>>> names_and_countries[-1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: -1

# You cannot slice
>>> names_and_countries['Adam':'Dani']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'slice'
```

Python expects everything between the brackets to be a key. So for the first two examples, we have a KeyError because neither 0 nor -1 are keys in the dictionary. But is possible to use 0 or -1 as a key, as we will see soon.

Note: As of Python 3.7, the order elements are added is preserved. Yet you cannot use this order to access elements. It is more for iteration and visual purposes as we will see later.

If we try to slice our dictionary, Python raises a TypeError. We explain why in the Hashing section.

Let's look at the second method for accessing the data stored in our dictionary.

Python Dictionary get() Method

```
# Get value for the key 'Adam'
>>> names_and_countries.get('Adam')
'Argentina'

# Returns None if key not in the dictionary
>>> names_and_countries.get('Zoe')

# Second argument returned if key not in dictionary
>>> names_and_countries.get('Zoe', 'Name not in dictionary')
'Name not in dictionary'

# Returns value if key in dictionary
```

```
>>> names_and_countries.get('Charlie', 'Name not in dictionary')
'Colombia'
```

The get() method takes two arguments:

1. The key you wish to search for
2. (optional) Value to return if the key is not in the dictionary (default is None).

It works like bracket notation. But it will never raise a KeyError. Instead, it returns either None or the object you input as the second argument.

This is hugely beneficial if you are iterating over a dictionary. If you use bracket notation and encounter an error, the whole iteration will stop. If you use get(), no error will be raised and the iteration will complete.

We will see how to iterate over dictionaries soon. But there is no point in doing that if we don't even know what our dictionary can contain! Let's learn about what can and can't be a key-value pair.

Python Dict Keys

In real dictionaries, the spelling of words doesn't change. It would make it quite difficult to use one if they did. The same applies to Python dictionaries. Keys cannot change. But they can be more than just strings. In fact, keys can be any immutable data type: string, int, float, bool or tuple.

Without a key, you can't get in.
Photo by CMDR Shane on Unsplash

```
>>> string_dict = {'hi': 'hello'}
>>> int_dict = {1: 'hello'}
>>> float_dict = {1.0: 'hello'}
>>> bool_dict = {True: 'hello', False: 'goodbye'}
>>> tuple_dict = {(1, 2): 'hello'}

# Tuples must only contain immutable types
>>> bad_tuple_dict = {(1, [2, 3]): 'hello'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

This is the second time we've seen "TypeError: unhashable type: 'list'". So what does 'unhashable' mean?

What is Hashing in Python?

In the background, a Python dictionary is a data structure known as a hash table. It contains keys and hash values (numbers of fixed length). You apply hash() to a key to return its hash value. If we call hash() on the same key many times, the result will not change.

```
# Python 3.8 (different versions may give different results)
>>> hash('hello world!')
1357213595066920515

# Same result as above
>>> hash('hello world!')
1357213595066920515

# If we change the object, we change the hash value
>>> hash('goodbye world!')
-7802652278768862152
```

When we create a key-value pair, Python creates a hash-value pair in the background

```
# We write
>>> {'hello world!': 1}

# Python executes in the background
>>> {hash('hello world!'): 1}

# This is equivalent to
>>> {1357213595066920515: 1}
```

Python uses this hash value when we look up a key-value pair. By design, the hash function can only be applied to immutable data types. If keys could change, Python would have to create a new hash table from scratch every time you change them. This would cause huge inefficiencies and many bugs.

Instead, once a table is created, the hash value cannot change. Python knows which values are in the table and doesn't need to calculate them again. This makes dictionary lookup and membership operations instantaneous and of O(1).

In Python, the concept of hashing only comes up when discussing dictionaries. Whereas, mutable vs immutable data types come up everywhere. Thus we say that you can only use immutable data types as keys, rather than saying 'hashable' data types.

Finally, what happens if you use the hash value of an object as another key in the sa,e dictionary? Does Python get confused?

```
>>> does_this_work = {'hello': 1,
                      hash('hello'): 2}

>>> does_this_work['hello']
1

>>> does_this_work[hash('hello')]
```

```
2
```

It works! The reasons why are beyond the scope of this article. The full implementation of the algorithm and the reasons why it works are described here. All you really need to know is that Python always picks the correct value... even if you try to confuse it!

Python Dictionary Values

There are restrictions on dictionary keys but values have none. Literally anything can be a value. As long as your key is an immutable data type, your key-value pairs can be any combination of types you want. You have complete control!

When inputting values you'll feel as free as this person!
Photo by Mohamed Nohassi on Unsplash

```
>>> crazy_dict = {11.0: ('foo', 'bar'),
                  'baz': {1: 'a', 2: 'b'},
                  (42, 55): {10, 20, 30},
                   True: False}

# Value of the float 11.0 is a tuple
>>> crazy_dict[11.0]
```

```
('foo', 'bar')

# Value of the string 'baz' is a dictionary
>>> crazy_dict.get('baz')
{1: 'a', 2: 'b'}

# Value of the tuple (42, 55) is a set
>>> crazy_dict[(42, 55)]
{10, 20, 30}

# Value of the Bool True is the Bool False
>>> crazy_dict.get(True)
False
```

Note: you must use braces notation to type a dictionary out like this. If you try to use the dict() constructor, you will get SyntaxErrors (unless you use the verbose method and type out a list of tuples... but why would you do that?).

Python Nested Dictionaries

When web scraping, it is very common to work with dictionaries inside dictionaries (nested dictionaries). To access values on deeper levels, you simply chain methods together. Any order of bracket notation and get() is possible.

```
# Returns a dict
>>> crazy_dict.get('baz')
{1: 'a', 2: 'b'}

# Chain another method to access the values of this dict
>>> crazy_dict.get('baz').get(1)
'a'

>>> crazy_dict.get('baz')[2]
'b'
```

We now know how to create a dictionary and what data types are allowed where. But what if you've already created a dictionary and want to add more values to it?

Python Add To Dictionary

```
>>> names_and_countries
{'Adam': 'Argentina',
'Beth': 'Bulgaria',
'Charlie': 'Colombia',
'Dani': 'Denmark',
'Ethan': 'Estonia'}
```

```
# Add key-value pair 'Zoe': 'Zimbabwe'
>>> names_and_countries['Zoe'] = 'Zimbabwe'

# Add key-value pair 'Fred': 'France'
>>> names_and_countries['Fred'] = 'France'

# Print updated dict
>>> names_and_countries
{'Adam': 'Argentina',
'Beth': 'Bulgaria',
'Charlie': 'Colombia',
'Dani': 'Denmark',
'Ethan': 'Estonia',
'Zoe': 'Zimbabwe',     # Zoe first
'Fred': 'France'}      # Fred afterwards
```

Our dictionary reflects the order we added the pairs by first showing Zoe and then Fred.

To add a new key-value pair, we simply assume that key already exists and try to acces it via bracket notation

```
>>> my_dict['new_key']
```

Then (before pressing return) use the assignment operator '=' and provide a value.

```
>>> my_dict['new_key'] = 'new_value'
```

You cannot assign new key-value pairs via the get() method because it's a function call.

```
>>> names_and_countries.get('Holly') = 'Hungary'
File "<stdin>", line 1
SyntaxError: cannot assign to function call
```

To delete a key-value pair use the del statement. To change the value of an existing key, use the same bracket notation as above.

```
# Delete the Zoe entry
>>> del names_and_countries['Zoe']

# Change Ethan's value
>>> names_and_countries['Ethan'] = 'DIFFERENT_COUNTRY'

>>> names_and_countries
{'Adam': 'Argentina',
'Beth': 'Bulgaria',
'Charlie': 'Colombia',
'Dani': 'Denmark',
```

```
'Ethan': 'DIFFERENT_COUNTRY',  # Ethan has changed
'Fred': 'France'}                   # We no longer have Zoe
```

As with other mutable data types, be careful when using the del statement in a loop. It modifies the dictionary in place and can lead to unintended consequences. Best practice is to create a copy of the dictionary and to change the copy. Or you can use, my personal favourite, dictionary comprehensions (which we will cover later).

Python Dict Copy Method

```
>>> my_dict = {'a': 1, 'b': 2}

# Create a shallow copy
>>> shallow_copy = my_dict.copy()

# Create a deep copy
>>> import copy
>>> deep_copy = copy.deepcopy(my_dict)
```

To create a shallow copy of a dictionary use the copy() method. To create a deep copy use the deepcopy() method from the built-in copy module. We won't discuss the distinction between the copy methods in this article for brevity.

Checking Dictionary Membership

Let's say we have a dictionary with 100k key-value pairs. We cannot print it to the screen and visually check which key-value pairs it contains.

Thankfully, the following syntax is the same for dictionaries as it is for other objects such as lists and sets. We use the 'in' keyword.

```
# Name obviously not in our dict
>>> 'INCORRECT_NAME' in names_and_countries
False

# We know this is in our dict
>>> 'Adam' in names_and_countries
True

# Adam's value is in the dict... right?
>>> names_and_countries['Adam']
'Argentina'
>>> 'Argentina' in names_and_countries
False
```

We expect INCORRECT_NAME not to be in our dict and Adam to be in it. But why does 'Argentina' return False? We've just seen that it's the value of Adam?!

Remember at the start of the article that I said dictionaries are maps? They map from something we know (the key) to something we don't (the value). So when we ask if something is in our dictionary, we are asking if it is a key. We're not asking if it's a value.

Which is more natural when thinking of a real-life dictionary:

1. Is the word 'facetious' in this dictionary?
2. Is the word meaning 'lacking serious intent; concerned with something nonessential, amusing or frivolous' in this dictionary?

Clearly the first one is the winner and this is the default behaviour for Python.

```
>>> 'something' in my_dict
```

We are checking if 'something' is a *key* in my_dict.

But fear not, if you want to check whether a specific value is in a dictionary, that is possible! We simply have to use some methods.

Python Dictionary Methods - Keys, Values and Items

There are 3 methods to look at. All can be used to check membership or for iterating over specific parts of a dictionary. Each returns an iterable.

- .keys() - iterate over the dictionary's keys
- .values() - iterate over the dictionary's values
- .items() - iterate over both the dictionary's keys and values

Note: we've changed Ethan's country back to Estonia for readability

```
>>> names_and_countries.keys()
dict_keys(['Adam', 'Beth', 'Charlie', 'Dani', 'Ethan', 'Fred'])

>>> names_and_countries.values()
dict_values(['Argentina', 'Bulgaria', 'Colombia', 'Denmark', 'Estonia', 'France'])

>>> names_and_countries.items()
```

```
dict_items([('Adam', 'Argentina'),
            ('Beth', 'Bulgaria'),
            ('Charlie', 'Colombia'),
            ('Dani', 'Denmark'),
            ('Ethan', 'Estonia'),
            ('Fred', 'France')])
```

We can now check membership in keys and values

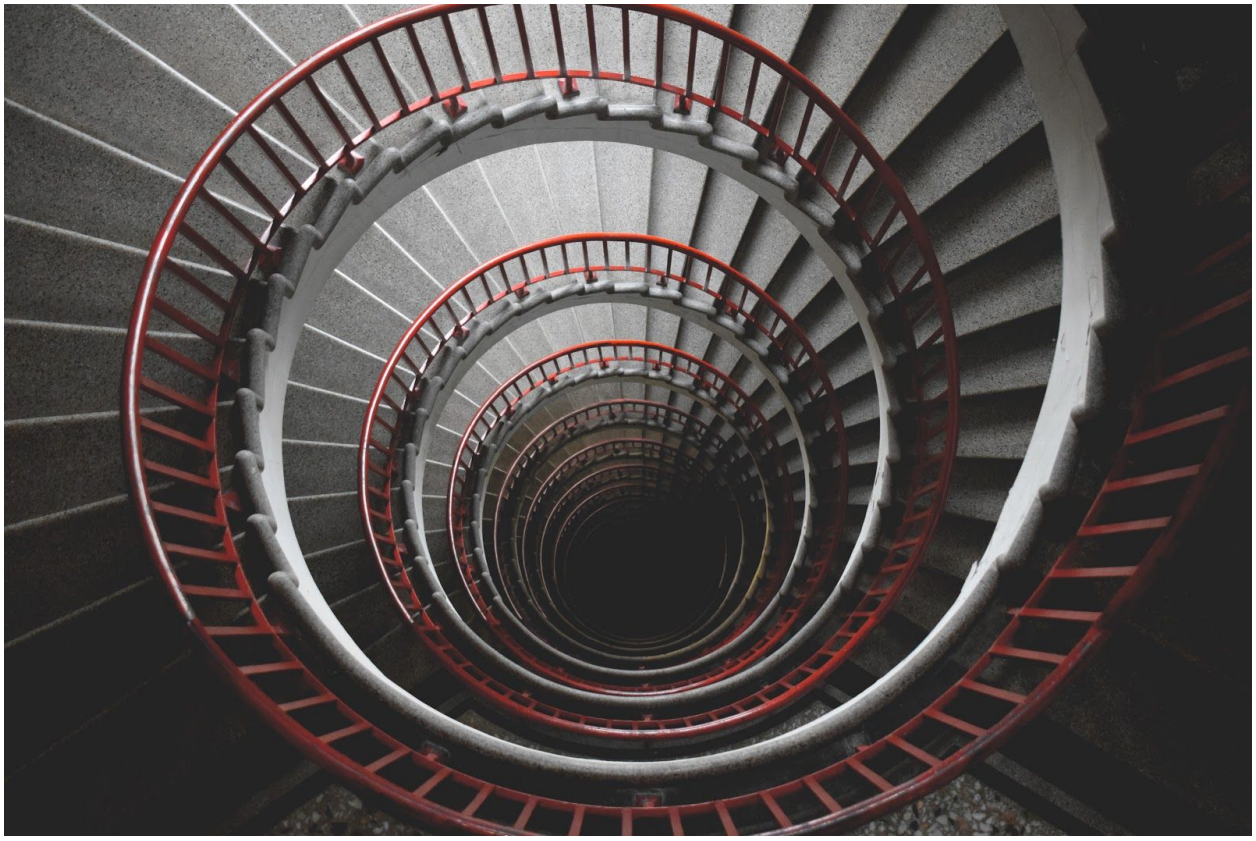
```
# Check membership in dict's keys
>>> 'Adam' in names_and_countries
True
>>> 'Adam' in names_and_countries.keys()
True

# Check membership in the dict's values
>>> 'Argentina' in names_and_countries.values()
True

# Check membership in either keys or values???
>>> 'Denmark' in names_and_countries.items()
False
```

You cannot check in the keys and values at the same time. This is because items() returns an iterable of tuples. As 'Denmark' is not a tuple, it will return False.

```
>>> for thing in names_and_countries.items():
        print(thing)
('Adam', 'Argentina')
('Beth', 'Bulgaria')
('Charlie', 'Colombia')
('Dani', 'Denmark')
('Ethan', 'Estonia')

# True because it's a tuple containing a key-value pair
>>> ('Dani', 'Denmark') in names_and_countries.items()
True
```

Python Loop Through Dictionary - An Overview

To iterate over any part of the dictionary we can use a for loop



Let's start looping
Photo by Tine Ivanič on Unsplash

```
>>> for name in names_and_countries.keys():
        print(name)
Adam
Beth
Charlie
Dani
Ethan
Fred

>>> for country in names_and_countries.values():
        print(f'{country} is wonderful!')
Argentina is wonderful!
Bulgaria is wonderful!
Colombia is wonderful!
Denmark is wonderful!
Estonia is wonderful!
France is wonderful!

>>> for name, country in names_and_countries.items():
        print(f'{name} is from {country}.')
Adam is from Argentina.
Beth is from Bulgaria.
```

```
Charlie is from Colombia.
Dani is from Denmark.
Ethan is from Estonia.
Fred is from France.
```

It's best practice to use descriptive names for the objects you iterate over. Code is meant to be read and understood by humans! Thus we chose 'name' and 'country' rather than 'key' and 'value'.

```
# Best practice
>>> for descriptive_key, descriptive_value in my_dict.items():
        # do something

# Bad practice (but you will see it 'in the wild'!)
>>> for key, value in my_dict.items():
        # do something
```

If your key-value pairs don't follow a specific pattern, it's ok to use 'key' and 'value' as your iterable variables, or even 'k' and 'v'.

```
# Iterating over the dict is the same as dict.keys()
>>> for thing in names_and_countries:
        print(thing)
Adam
Beth
Charlie
Dani
Ethan
Fred
```

A Note On Reusability

```
# Works with general Python types
>>> for key in object:
        # do something

# Works only with dictionaries
>>> for key in object.keys():
        # do something
```

Do not specify keys() if your code needs to work with other objects like lists and sets. Use the keys() method if your code is only meant for dictionaries. This prevents future users inputting incorrect objects.

Python dict has_key

The method has_key() is exclusive to Python 2. It returns True if the key is in the dictionary and False if not.

Python 3 removed this functionality in favour of the following syntax

```
>>> if key in d:
        # do something
```

This keeps dictionary syntax in line with that of other data types such as sets and lists. This aids readability and reusability.

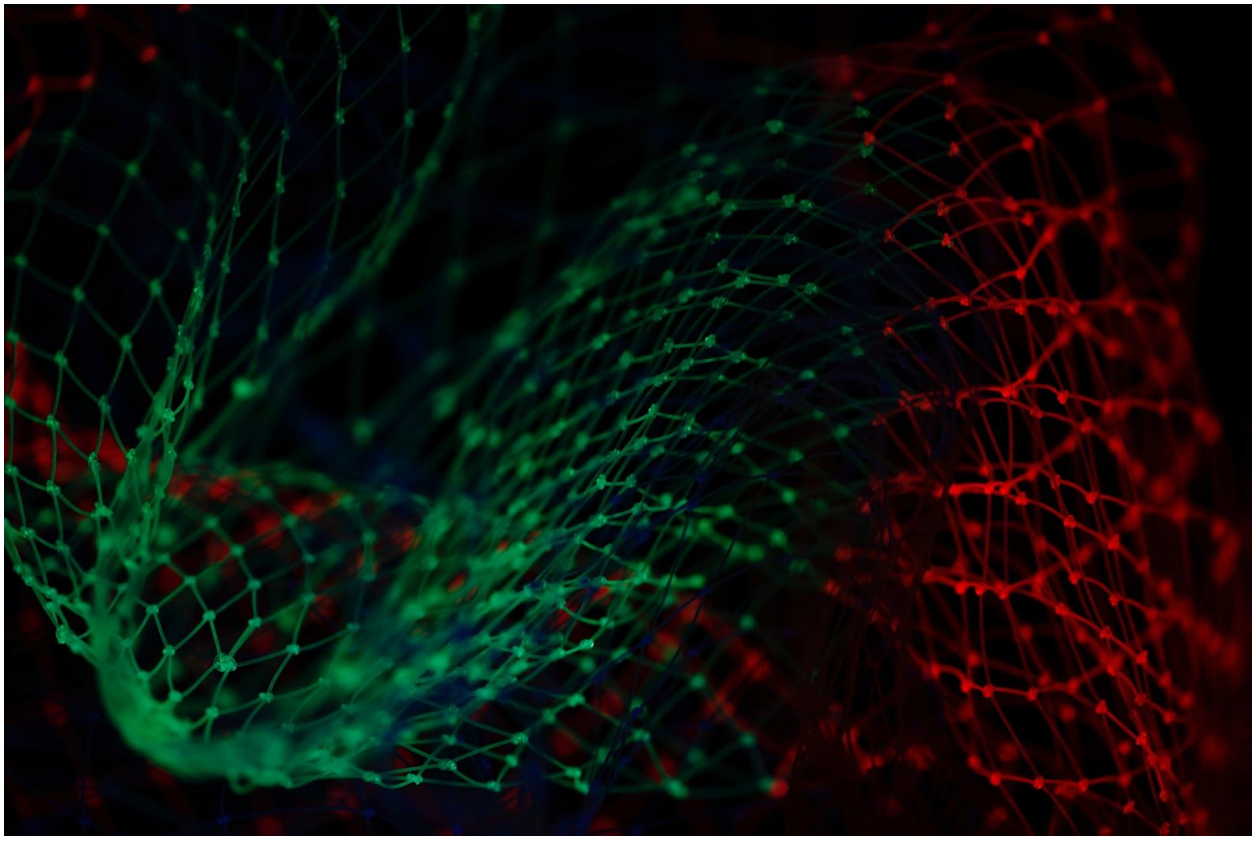
Pretty Printing Dictionaries Using pprint()

The built-in module pprint contains the function pprint. This will 'pretty print' your dictionary. It sorts the keys alphabetically and prints each key-value pair on a newline.

```
>>> from pprint import pprint
>>> messy_dict = dict(z='Here is a really long key that spans a lot of text', a='here is another long key that is really too long', j='this is the final key in this dictionary')

>>> pprint(messy_dict)
{'a': 'here is another long key that is really too long',
'j': 'this is the final key in this dictionary',
'z': 'Here is a really long key that spans a lot of text'}
```

It does not change the dictionary at all. It's just much more readable now.

Python Dictionaries and JSON Files



We need to encode and decode all this data
Photo by Pietro Jeng on Unsplash

A common filetype you will interact with is a JSON file. It stands for Javascript Object Notation. They are used to structure and send data in web applications.

They work almost exactly the same way as dictionaries and you can easily turn one into the other very easily.

Python Dict to JSON

```
>>> import json
>>> my_dict = dict(a=1, b=2, c=3, d=4)

>>> with open('my_json.json', 'w') as f:
        json.dump(my_dict, f)
```

The above code takes my_dict and writes it to the file my_json.json in the current directory.

You can get more complex than this by setting character encodings and spaces. For more detail, we direct the reader to the docs.

Python JSON to Dict

We have the file my_json.json in our current working directory.

```
>>> import json
>>> with open('my_json.json', 'r') as f:
        new_dict = json.load(f)

>>> new_dict
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Note: the key-value pairs in JSON are always converted to strings when encoded in Python. It is easy to change any object into a string and it leads to fewer errors when encoding and decoding files. But it means that sometimes the file you load and the file you started with are not identical.

Python Dictionary Methods

Here's a quick overview:

1. dict.clear() - remove all key-value pairs from a dict
2. dict.update() - merge two dictionaries together
3. dict.pop() - remove a key and return its value
4. dict.popitem() - remove a random key-value pair and return it as a tuple

We'll use letters A and B for our dictionaries as they are easier to read than descriptive names. Plus we have kept the examples simple to aid understanding.

dict.clear() - remove all key-value pairs from a dict

```
>>> A = dict(a=1, b=2)
>>> A.clear()
>>> A
{}
```

Calling this on a dict removes all key-value pairs in place. The dict is now empty.

dict.update() - merge two dictionaries together

```
>>> A = dict(a=1, b=2)
>>> B = dict(c=3, d=4)
>>> A.update(B)
>>> A
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> B
{'c': 3, 'd': 4}
```

We have just updated A. Thus all the key-value pairs from B have been added to A. B has not changed.

If A and B some keys, B's value will replace A's. This is because A is updated by B and so takes all of B's values (not the other way around).

```
>>> A = dict(a=1, b=2)
>>> B = dict(b=100)
>>> A.update(B)

# A now contains B's values
>>> A
{'a': 1, 'b': 100}

# B is unchanged
>>> B
{'b': 100}
```

You can also pass a sequence of tuples or keyword arguments to update(), like you would with the dict() constructor.

```
>>> A = dict(a=1, b=2)
# Sequence of tuples
>>> B = [('c', 3), ('d', 4)]
>>> A.update(B)
>>> A
{'a': 1, 'b': 2, 'c': 3, 'd': 4}

>>> A = dict(a=1, b=2)
# Pass key-value pairs as keyword arguments
>>> A.update(c=3, d=4)
>>> A
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

dict.pop() - remove a key and return its value

```
>>> A = dict(a=1, b=2)
>>> A.pop('a')
1
>>> A
```

```
{'b': 2}
```

If you try call dict.pop() with a key that is not in the dictionary, Python raises a KeyError.

```
>>> A.pop('non_existent_key')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'non_existent_key'
```

Like the get() method, you can specify an optional second argument. This is returned if the key is not in the dictionary and so avoids KeyErrors.

```
>>> A.pop('non_existent_key', 'not here')
'not here'
```

dict.popitem() - remove a random key-value pair and return it as a tuple

```
>>> A = dict(a=1, b=2, c=3)
# Your results will probably differ
>>> A.popitem()
('c', 3)
>>> A
{'a': 1, 'b': 2}
>>> A.popitem()
('b', 2)
>>> A
{'a': 1}
```

If the dictionary is empty, Python raises a KeyError.

```
>>> A = dict()
>>> A.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

Python Loop Through Dictionary - In Detail

There are several common situations you will encounter when iterating over dictionaries. Python has developed several methods to help you work more efficiently.

But before we head any further, please remember the following:

NEVER EVER use bracket notation when iterating over a dictionary. If there are any errors, the whole iteration will break and you will not be happy.

The standard Python notation for incrementing numbers or appending to lists is

```
# Counting
my_num = 0
for thing in other_thing:
    my_num += 1

# Appending to lists
my_list = []
for thing in other_thing:
    my_list.append(thing)
```

This follows the standard pattern:

1. Initialise 'empty' object
2. Begin for loop
3. Add things to that object

When iterating over a dictionary, our values can be numbers or list-like. Thus we can add or we can append to values. It would be great if our code followed the above pattern. But...

```
>>> my_dict = {}
>>> for thing in other_thing:
        my_dict['numerical_key'] += 1
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'numerical_key'

>>> for thing in other_thing:
        my_dict['list_key'].append(thing)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'list_key'
```

Unfortunately, both raise a KeyError. Python tells us the key do not exist and so we cannot increment its value. Thus we must first create a key-value pair before we do anything with it.

We'll now show 4 ways to solve this problem:

1. Manually initialise a key if it does not exist
2. The get() method
3. The setdefault() method

4. The defaultdict()

We'll explain this through some examples, so let's go to the setup.

Three friends - Adam, Bella and Cara, have gone out for a meal on Adam's birthday. They have stored their starter, main and drinks orders in one list. The price of each item is in another list. We will use this data to construct different dictionaries.

```
people = ['Adam', 'Bella', 'Cara',
          'Adam', 'Bella', 'Cara',
          'Adam', 'Bella', 'Cara',]

food = ['soup', 'bruschetta', 'calamari',   # starter
        'burger', 'calzone', 'pizza',       # main
        'coca-cola', 'fanta', 'water']      # drink

# Cost of each item in £
prices = [3.20, 4.50, 3.89,
          12.50, 15.00, 13.15,
          3.10, 2.95, 1.86]

# Zip data together to allow iteration
# We only need info about the person and the price
meal_data = zip(people, prices)
```

Our three friends are very strict with their money. They want to pay exactly the amount they ordered. So we will create a dictionary containing the total cost for each person. This is a numerical incrementation problem.

Manually Initialise a Key

```
# Initialise empty dict
total = {}

# Iterate using descriptive object names
for (person, price) in meal_data:

    # Create new key and set value to 0 if key doesn't yet exist
    if person not in total:
        total[person] = 0

    # Increment the value by the price of each item purchased.
    total[person] += price

>>> total
{'Adam': 18.8, 'Bella': 22.45, 'Cara': 18.9}
```

We write an if statement which checks if the key is already in the dictionary. If it isn't, we set the value to 0. If it is, Python does not execute the if statement. We then increment using the expected syntax.

This works well but requires quite a few lines of code. Surely we can do better?

Python Dict get() Method When Iterating

```
# Reinitialise meal_data as we have already iterated over it
meal_data = zip(people, prices)

total = {}
for (person, price) in meal_data:

    # get method returns 0 the first time we call it
    # and returns the current value subsequent times
    total[person] = total.get(person, 0) + price

>>> total
{'Adam': 18.8, 'Bella': 22.45, 'Cara': 18.9}
```

We've got it down to one line!

We pass get() a second value which is returned if the key is not in the dictionary. In this case, we choose 0 like the above example. The first time we call get() it returns 0. We have just initialised a key-value pair! In the same line, we add on 'price'. The next time we call get(), it returns the current value and we can add on 'price' again.

This method does not work for appending. You need some extra lines of code. We will look at the setdefault() method instead.

Python Dict setdefault() Method

The syntax of this method makes it an excellent choice for modifying a key's value via the append() method.

First we will show why it's not a great choice to use if you are incrementing with numbers.

```
meal_data = zip(people, prices)
total = {}
for (person, price) in meal_data:

    # Set the initial value of person to 0
    total.setdefault(person, 0)

    # Increment by price
```

```
    total[person] += price

0
0
0
3.2
4.5
3.89
15.7
19.5
17.04
>>> total
{'Adam': 18.8, 'Bella': 22.45, 'Cara': 18.9}
```

It works but requires more lines of code than get() and prints lots of numbers to the screen. Why is this?

The setdefault() method takes two arguments:

1. The key you wish to set a default value for
2. What you want the default value to be

So setdefault(person, 0) sets the default value of person to be 0.

It always returns one of two things:

1. The current value of the key
2. If the key does not exist, it returns the default value provided

This is why the numbers are printed to the screen. They are the values of 'person' at each iteration.

Clearly this is not the most convenient method for our current problem. If we do 100k iterations, we don't want 100k numbers printed to the screen.

So we recommend using the get() method for numerical calculations.

Let's see it in action with lists and sets. In this dictionary, each person's name is a key. Each value is a list containing the price of each item they ordered (starter, main, dessert).

```
meal_data = zip(people, prices)
individual_bill = {}

for (person, price) in meal_data:
```

```
    # Set default to empty list and append in one line!
    individual_bill.setdefault(person, []).append(price)

>>> individual_bill
{'Adam': [3.2, 12.5, 3.1],
'Bella': [4.5, 15.0, 2.95],
'Cara': [3.89, 13.15, 1.86]}
```

Now we see the true power of setdefault()! Like the get method in our numerical example, we initialise a default value and modify it in one line!

Note: setdefault() calculates the default value every time it is called. This may be an issue if your default value is expensive to compute. Get() only calculates the default value if the key does not exist. Thus get() is a better choice if your default value is expensive. Since most default values are 'zeros' such as 0, [ ] and { }, this is not an issue for most cases.

We've seen three solutions to the problem now. We've got the code down to 1 line. But the syntax for each has been different to what we want. Now let's see something that solves the problem exactly as we'd expect: introducing defaultdict!

Python defaultdict()

Let's solve our numerical incrementation problem:

```
# Import from collections module
from collections import defaultdict

meal_data = zip(people, prices)

# Initialise with int to do numerical incrementation
total = defaultdict(int)

# Increment exactly as we want to!
for (person, price) in meal_data:
    total[person] += price

>>> total
defaultdict(<class 'int'>, {'Adam': 18.8, 'Bella': 22.45, 'Cara': 18.9})
```

Success!! But what about our list problem?

```
from collections import defaultdict

meal_data = zip(people, prices)

# Initialise with list to let us append
```

```
individual_bill = defaultdict(list)

for (person, price) in meal_data:
    individual_bill[person].append(price)

>>> individual_bill
defaultdict(<class 'list'>, {'Adam': [3.2, 12.5, 3.1],
                                        'Bella': [4.5, 15.0, 2.95],
                                        'Cara': [3.89, 13.15, 1.86]})
```

The defaultdict is part of the built-in collections module. So before we use it, we must first import it.

Defaultdict is the same as a normal Python dictionary except:

1. It takes a callable data type as an argument
2. When it meets a key for the first time, the default value is set as the 'zero' for that data type. For int it is 0, for list it's an empty list [ ] etc..

Thus you will never get a KeyError! Plus and initialising default values is taken care of automatically!

We have now solved the problem using the same syntax for lists and numbers!

Now let's go over some special cases for defaultdict.

Python defaultdict() Special Cases

Above we said it's not possible to get a KeyError when using defaultdict. This is only true if you correctly initialise your dict.

```
# Initialise without an argument
>>> bad_dict = defaultdict()
>>> bad_dict['key']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'key'

# Initialise with None
>>> another_bad_dict = defaultdict(None)
>>> another_bad_dict['another_key']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'another_key'
```

Let's say you initialise defaultdict without any arguments. Then Python raises a KeyError if you call a key not in the dictionary. This is the same as initialising with None and defeats the whole purpose of defaultdict.

The issue is that None is not callable. Yet you can get defaultdict to return None by using a lambda function

```
>>> none_dict = defaultdict(lambda: None)
>>> none_dict['key']
>>>
```

Note that you cannot increment or append to None. Make sure you choose your default value to match the problem you are solving!

Whilst we're here, let's take a look at some more dictionaries in the collections module.

OrderedDict

Photo by Brooke Lark on Unsplash

Earlier we said that dictionaries preserve their order from Python 3.7 onwards. So why do we need something called OrderedDict?

As the name suggests, OrderedDict preserves the order elements are added. But two OrderedDicts are the same if and only if their elements are in the same order. This is not the case with normal dicts.

```
>>> from collections import OrderedDict

# Normal dicts preserve order but don't use it for comparison
>>> normal1 = dict(a=1, b=2)
>>> normal2 = dict(b=2, a=1)
>>> normal1 == normal2
True

# OrderedDicts preserve order and use it for comparison
>>> ordered1 = OrderedDict(a=1, b=2)
>>> ordered2 = OrderedDict(b=2, a=1)
>>> ordered1 == ordered2
False
```

Other than that, OrderedDict has all the same properties as a regular dictionary. If your elements must be in a particular order, then use OrderedDict!

Counter()

Let's say we want to count how many times each word appears in a piece of text (a common thing to do in NLP). We'll use The Zen of Python for our example. If you don't know what it is, run

```
>>> import this
```

I've stored it in the list zen_words where each element is a single word.

We can manually count each word using defaultdict. But printing it out with the most frequent words occurring first is a bit tricky.

```
>>> from collections import defaultdict
>>> word_count = defaultdict(int)
>>> for word in zen_words:
        word_count[word] += 1

# Define function to return the second value of a tuple
>>> def select_second(tup):
        return tup[1]

# Reverse=True - we want the most common first
```

```
# word_count.items() - we want keys and values
# sorted() returns a list, so wrap in dict() to return a dict

>>> dict(sorted(word_count.items(), reverse=True, key=select_second))
{'is': 10,
'better': 8,
'than': 8,
'to': 5,
...}
```

As counting is quite a common process, the Counter() dict subclass was created. It is complex enough that we could write a whole article about it.

For brevity, we will include the most basic use cases and let the reader peruse the docs themselves.

```
>>> from collections import Counter
>>> word_count = Counter(zen_words)
>>> word_count
Counter({'is': 10, 'better': 8, 'than': 8, 'to': 5, ...})
```

You can pass any iterable or dictionary to Counter(). It returns a dictionary in descending order of counts

```
>>> letters = Counter(['a', 'b', 'c', 'c', 'c', 'c', 'd', 'd', 'a'])
>>> letters
Counter({'c': 4, 'a': 2, 'd': 2, 'b': 1})

# Count of a missing key is 0
>>> letters['z']
0
```

Reversed()

In Python 3.8 they introduced the reversed() function for dictionaries! It returns an iterator. It iterates over the dictionary in the opposite order to how the key-value pairs were added. If the key-value pairs have no order, reversed() will not give them any further ordering. If you want to sort the keys alphabetically for example, use sorted().

```
# Python 3.8

# Reverses the order key-value pairs were added to the dict
>>> ordered_dict = dict(a=1, b=2, c=3)
>>> for key, value in reversed(ordered_dict.items()):
        print(key, value)
```

```
c 3
b 2
a 1

# Does not insert order where there is none.
>>> unordered_dict = dict(c=3, a=1, b=2)
>>> for key, value in reversed(unordered_dict.items()):
        print(key, value)
b 2
a 1
c 3

# Order unordered_dict alphabetically using sorted()
>>> dict(sorted(unordered_dict.items()))
{'a': 1, 'b': 2, 'c': 3}
```

Since it's an iterator, remember to use the keys(), values() and items() methods to select the elements you want. If you don't specify anything, you'll iterate over the keys.

Dictionary Comprehensions

A wonderful feature of dictionaries, and Python in general, is the comprehension. This lets you create dictionaries in a clean, easy to understand and Pythonic manner. You must use curly braces {} to do so (not dict()).

We've already seen that if you have two lists, you can create a dictionary from them using dict(zip()).

```
names = ['Adam', 'Beth', 'Charlie', 'Dani', 'Ethan']
countries = ['Argentina', 'Bulgaria', 'Colombia', 'Denmark', 'Estonia']

dict_zip = dict(zip(names, countries))

>>> dict_zip
{'Adam': 'Argentina',
'Beth': 'Bulgaria',
'Charlie': 'Colombia',
'Dani': 'Denmark',
'Ethan': 'Estonia'}
```

We can also do this using a for loop

```
>>> new_dict = {}
>>> for name, country in zip(names, countries):
        new_dict[name] = country
```

```
>>> new_dict
{'Adam': 'Argentina',
'Beth': 'Bulgaria',
'Charlie': 'Colombia',
'Dani': 'Denmark',
'Ethan': 'Estonia'}
```

We initialise our dict and iterator variables with descriptive names. To iterate over both lists at the same time we zip them together. Finally, we add key-value pairs as desired. This takes 3 lines.

Using a comprehension turns this into one line.

```
dict_comp = {name: country for name, country in zip(names, countries)}

>>> dict_comp
{'Adam': 'Argentina',
'Beth': 'Bulgaria',
'Charlie': 'Colombia',
'Dani': 'Denmark',
'Ethan': 'Estonia'}
```

They are a bit like for loops in reverse. First, we state what we want our key-value pairs to be. Then we use the same for loop as we did above. Finally, we wrap everything in curly braces.

Note that every comprehension can be written as a for loop. If you ever get results you don't expect, try it as a for loop to see what is happening.

Here's a common mistake

```
dict_comp_bad = {name: country
                 for name in names
                 for country in countries}

>>> dict_comp_bad
{'Adam': 'Estonia',
'Beth': 'Estonia',
'Charlie': 'Estonia',
'Dani': 'Estonia',
'Ethan': 'Estonia'}
```

What's going on? Let's write it as a for loop to see. First, we'll write it out to make sure we are getting the same, undesired, result.

```
bad_dict = {}
for name in names:
```

```
    for country in countries:
        bad_dict[name] = country

>>> bad_dict
{'Adam': 'Estonia',
'Beth': 'Estonia',
'Charlie': 'Estonia',
'Dani': 'Estonia',
'Ethan': 'Estonia'}
```

Now we'll use the bug-finder's best friend: the print statement!

```
# Don't initialise dict to just check for loop logic
for name in names:
    for country in countries:
        print(name, country)
Adam Argentina
Adam Bulgaria
Adam Colombia
Adam Denmark
Adam Estonia
Beth Argentina
Beth Bulgaria
Beth Colombia
...
Ethan Colombia
Ethan Denmark
Ethan Estonia
```

Here we remove the dictionary to check what is actually happening in the loop. Now we see the problem! The issue is we have nested for loops. The loop says: for each name pair it with every country. Since dictionary keys can only appear, the value gets overwritten on each iteration. So each key's value is the final one that appears in the loop - 'Estonia'.

The solution is to remove the nested for loops and use zip() instead.

Python Nested Dictionaries with Dictionary Comprehensions

```
nums = [0, 1, 2, 3, 4, 5]

dict_nums = {n: {'even': n % 2 == 0,
                 'square': n**2,
                 'cube': n**3,
                 'square_root': n**0.5}
             for n in nums}

# Pretty print for ease of reading
>>> pprint(dict_nums)
```

```
{0: {'cube': 0, 'even': True, 'square': 0, 'square_root': 0.0},
1: {'cube': 1, 'even': False, 'square': 1, 'square_root': 1.0},
2: {'cube': 8, 'even': True, 'square': 4, 'square_root': 1.4142135623730951},
3: {'cube': 27, 'even': False, 'square': 9, 'square_root': 1.7320508075688772},
4: {'cube': 64, 'even': True, 'square': 16, 'square_root': 2.0},
5: {'cube': 125, 'even': False, 'square': 25, 'square_root': 2.23606797749979}}
```

This is where comprehensions become powerful. We define a dictionary within a dictionary to create lots of information in a few lines of code. The syntax is exactly the same as above but our value is more complex than the first example.

Remember that our key value pairs must be unique and so we cannot create a dictionary like the following

```
>>> nums = [0, 1, 2, 3, 4, 5]
>>> wrong_dict = {'number': num, 'square': num ** 2 for num in nums}
  File "<stdin>", line 1
    wrong_dict = {'number': num, 'square': num ** 2 for num in nums}
                                                    ^
SyntaxError: invalid syntax
```

We can only define one pattern for key-value pairs in a comprehension. But if you could define more, it wouldn't be very helpful. We would overwrite our key-value pairs on each iteration as keys must be unique.

If-Elif-Else Statements

```
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Just the even numbers
even_squares = {n: n ** 2 for n in nums
                if n % 2 == 0}

# Just the odd numbers
odd_squares = {n: n ** 2 for n in nums
               if n % 2 == 1}

>>> even_dict
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64, 10: 100}

>>> odd_dict
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

We can apply if conditions after the for statement. This affects all the values you are iterating over.

You can also apply them to your key and value definitions. We'll now create different key-value pairs based on whether a number is odd or even.

```
# Use parenthesis to aid readability
different_vals = {n: ('even' if n % 2 == 0 else 'odd')
                  for n in range(5)}

>>> different_vals
{0: 'even', 1: 'odd', 2: 'even', 3: 'odd', 4: 'even'}
```

We can get really complex and use if/else statements in both the key-value definitions and after the for loop!

```
# Change each key using an f-string
{(f'{n}_cubed' if n % 2 == 1 else f'{n}_squared'):

# Cube odd numbers, square even numbers
 (n ** 3 if n % 2 == 1 else n ** 2)

# The numbers 0-10 inclusive
 for n in range(11)

# If they are not multiples of 3
 if n % 3 != 0}

{'1_cubed': 1, '2_squared': 4, '4_squared': 16, '5_cubed': 125, '7_cubed': 343,
'8_squared': 64, '10_squared': 100}
```

It is relatively simple to do this using comprehensions. Trying to do so with a for loop or dict() constructor would be much harder.

Merging Two Dictionaries

Let's say we have two dictionaries A and B. We want to create a dictionary, C, that contains all the key-value pairs of A and B. How do we do this?

```
>>> A = dict(a=1, b=2)
>>> B = dict(c=3, d=4)

# Update method does not create a new dictionary
>>> C = A.update(B)
>>> C
>>> type(C)
<class 'NoneType'>
```

```
>>> A
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Using merge doesn't work. It modifies A in place and so doesn't return anything.

Before Python 3.5, you had to write a function to do this. In Python 3.5 they introduced this wonderful bit of syntax.

```
# Python >= 3.5
>>> A = dict(a=1, b=2)
>>> B = dict(c=3, d=4)
>>> C = {**A, **B}
>>> C
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

We use ** before each dictionary to 'unpack' all the key-value pairs.

The syntax is very simple: a comma-separated list of dictionaries wrapped in curly braces. You can do this for an arbitrary number of dictionaries.

```
A = dict(a=1, b=2)
B = dict(c=3, d=4)
C = dict(e=5, f=6)
D = dict(g=7, h=8)
>>> all_the_dicts = {**A, **B, **C, **D}
>>> all_the_dicts
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8}
```

Finally, what happens if the dicts share key-value pairs?

```
>>> A = dict(a=1, b=2)
>>> B = dict(a=999)
>>> B_second = {**A, **B}
>>> A_second = {**B, **A}

# Value of 'a' taken from B
>>> B_second
{'a': 999, 'b': 2}

# Value of 'a' taken from A
>>> A_second
{'a': 1, 'b': 2}
```

As is always the case with Python dictionaries, a key's value is dictated by its last assignment. The dict B_second first takes A's values then take's B's. Thus any shared keys between A and B will be overwritten with B's values. The opposite is true for A_second.

Note: if a key's value is overridden, the position of that key in the dict does not change.

```
>>> D = dict(g=7, h=8)
>>> A = dict(a=1, g=999)
>>> {**D, **A}

# 'g' is still in the first position despite being overridden with A's value
{'g': 999, 'h': 8, 'a': 1}
```

Conclusion

You now know almost everything you'll ever need to know to use Python Dictionaries. Well done! Please bookmark this page and refer to it as often as you need!

If you have any questions post them in the comments and we'll get back to you as quickly as possible.

If you love Python and want to become a freelancer, there is no better course out there than this one. I bought it myself and it is why you are reading these words today.

References

1. https://www.dictionary.com/
2. https://tinyurl.com/yg6kgy9h
3. https://stackoverflow.com/questions/7886355/defaultdictnone
4. https://www.datacamp.com/community/tutorials/python-dictionary-tutorial
5. https://docs.python.org/3.8/tutorial/datastructures.html#dictionaries
6. https://stackoverflow.com/questions/526125/why-is-python-ordering-my-dictionary-like-so
7. https://stackoverflow.com/a/378987/11829398
8. https://en.wikipedia.org/wiki/Hash_function
9. https://docs.python.org/2/library/collections.html#collections.OrderedDict
10. https://www.quora.com/What-are-hashable-types-in-Python
11. https://hg.python.org/cpython/file/default/Objects/dictobject.c
12. https://www.dictionary.com/browse/facetious?s=t
13. https://thispointer.com/python-how-to-copy-a-dictionary-shallow-copy-vs-deep-copy/
14. https://docs.python.org/3.8/library/collections.html#collections.Counter
15. https://stackoverflow.com/questions/12309269/how-do-i-write-json-data-to-a-file
16. https://realpython.com/python-dicts/#built-in-dictionary-methods

17. https://stackoverflow.com/questions/38987/how-do-i-merge-two-dictionaries-in-a-single-expression