Python

Geordan

March 15, 2023

This notebook will be used to note down the things I learn in python. Originally meant for dictionary syntax but we will see.

```
if key in some_dict:
    value = some_dict[key]
else:
    value = default_value

value = some_dict.get(key, default_value)
```

The first if else statement does the same thing as the .get method. Basically, if a key is in the dictionary it returns the value of that key, else it returns a default value

[15]:
```
words = ["barista", "apple", "bat", "bar", "atom", "book", "apple"]

by_letter = {}

for word in words:
     letter = word[0]
     if letter not in by_letter:
         by_letter[letter] = [word]
     else:
         by_letter[letter].append(word)

by_letter
```

[15]:
```
{'b': ['barista', 'bat', 'bar', 'book'], 'a': ['apple', 'atom', 'apple']}
```

Basically this selects the first letter in each word and adds it to a list. This list becomes the value of the the first letter as a key.

[16]:
```
by_letter = {}
for word in words:
    letter = word[0]
    by_letter.setdefault(letter, []).append(word)
by_letter
```

[16]:
```
{'b': ['barista', 'bat', 'bar', 'book'], 'a': ['apple', 'atom', 'apple']}
```

I noticed that this doesnt check if the a particular word is in the dictionary, only if the starting letter is.

This is a faster way to do it. With the set default. Here, if the word with the letter is found, it appends it to a list, else it doesnt.

There's a collections module which makes this even easier.

[17]:
```
from collections import defaultdict

by_letter = defaultdict(list)

for word in words:
    by_letter[word[0]].append(word)

dict(by_letter)
```

[17]:
```
{'b': ['barista', 'bat', 'bar', 'book'], 'a': ['apple', 'atom', 'apple']}
```

You can only use immutable objects as a key, i.e strings, tuples, but not lists.

Oh and apparently there are sets too. Which support things like union, intersection, etc

The enumerate function. It's actually so useful. Here's what it does and how to use it

[18]:
```
collection = list(range(3,30))
index = 0
for value in collection:
   # do something with value
   index += 1
print(index)

a = []
for index, value in enumerate(collection):
   # do something with value
    a.append(value + index)

print(a)
```

```
27
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43,
45, 47, 49, 51, 53, 55]
```

Reversed

Reversed can be used to reverse a list. It iterates over the the list in reversed order.

[19]:
```
list(reversed(range(10)))
```

[19]:
```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

List comprehension

Eliminates the need to go through a complicated for loop and enables a simple one liner.

```
[expr for value in collection if condition]
is the same as:
result = []
for value in collection:
    if condition:
        result.append(expr)
```

[20]:
```
strings = ["a", "as", "bat", "car", "dove", "python"]

[x.upper() for x in strings if len(x) > 2]
```

[20]:
```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Dictionary comprehensions are similar but a bit more natural.

```
Dictionary comprehension
dict_comp = {key-expr: value-expr for value in collection
             if condition}
Set comprehension
set_comp = {expr for value in collection if condition}
```

A set is an unordered collection of unique elements. Using it on data reveals unique occurances of elements in the data.

[21]:
```
# We can also use the map function.
set(map(len, strings))
```

[21]:
```
{1, 2, 3, 4, 6}
```

This one is actually pretty interesting.

[22]:
```
loc_mapping = {value: index for index, value in enumerate(strings)}

loc_mapping
```

[22]:
```
{'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

[23]:
```
all_data = [["John", "Emily", "Michael", "Mary", "Steven"], ["Maria", "Juan",␣
 ↪"Javier", "Natalia", "Pilar"]]
## Names with 2 or more a's
names_of_interest = []

for names in all_data:
    enough_as = [name for name in names if name.count("a") >= 2]
    names_of_interest.extend(enough_as)

print(names_of_interest)
```

```
## Single list comprehension
names_of_interest_d = [name for names in all_data for name in names if name.
↪count("a") >= 2]
print(names_of_interest_d)
```

```
['Maria', 'Natalia']
['Maria', 'Natalia']
```

Here's a nested list comprehension. Basically, the for loops follow the same order as in a normal nested for loop. The expression is just placed first and then the loops and finally the condition. You can also do a list comprehension inside a list comprehension.

[24]:
```
some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
print("some tuples:", some_tuples)
flattened = tuple([x for tup in some_tuples for x in tup])
print("A flattened tuple:", flattened)

flattened = []

for tup in some_tuples:
    for x in tup:
        flattened.append(x)
print("Flattened using a for loop:", flattened)

list_flattened = [[x for x in tup] for tup in some_tuples]
print("List of lists:", list_flattened)
```

```
some tuples: [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
A flattened tuple: (1, 2, 3, 4, 5, 6, 7, 8, 9)
Flattened using a for loop: [1, 2, 3, 4, 5, 6, 7, 8, 9]
List of lists: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Just when I thought there was nothing else to learn in functions. I find this beautiful cleaning module.

[25]:
```
states = ["   Alabama ", "Georgia!", "Georgia", "georgia", "FlOrIda", "south ␣
↪carolina##", "West virginia?"]

import re

def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub("[!#?]", "", value)
        value = value.title()
        result.append(value)
    return result
```

```
print(clean_strings(states))
```

```
['Alabama', 'Georgia', 'Georgia', 'Georgia', 'Florida', 'South   Carolina',
'West Virginia']
```

[39]:
```
# You can also define a list of functions to use on value.
# First, define a function for re.sub
def remove_punctuation(value):
    return re.sub("[!#?]", "", value)


# Create the list of functions
clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
    result = []
    for value in strings:
        for func in ops:
            value = func(value)
        result.append(value)
    return result


cleaned = clean_strings(states, clean_ops)
sc = list(cleaned[5])
sc = sc[0:6] + sc[8:]
sc = "".join(map(str, sc))
cleaned[5] = sc
print(cleaned)
# It is now reusable and it's easy to see which functions are being applied
# The spaces inside South Carolina were annoying me
```

```
['Alabama', 'Georgia', 'Georgia', 'Georgia', 'Florida', 'South Carolina', 'West
Virginia']
```