py\_comprehensions\_deep

In [167]:

%%html  
<style>  
  
/\*#These style overrides are for printing in b/w. The Jupyter style comes from  
 CodeMirror. The CodeMirror overrides all begin with .cm below.   
\*/  
div.input\_area {  
 border: 4px solid #cfcfcf;  
 border-radius: 4px;  
 background: white;  
 line-height: 1.21429em;  
}  
div.cell{color: black;}  
  
  
/\*  
These are all style overrides from the CodeMirror theme  
Good source of names:  
 https://github.com/draperjames/one-dark-notebook/blob/master/custom.css  
\*/  
  
/\*  
.cm-s-ipython span.cm-string {  
 color: black;  
}  
  
.cm-s-ipython span.cm-link {  
 color: black;  
}  
.cm-s-ipython span.cm-keyword {  
 color: black;  
}  
  
.cm-s-ipython .CodeMirror-linenumber {  
 color: black;  
 font-size: 10px;  
}  
  
.cm-s-ipython span.cm-comment {  
 color: black;  
}  
  
.cm-s-ipython span.cm-builtin {  
 color: black;  
}  
  
.cm-s-ipython span.cm-variable {  
 color: black;  
}  
  
.cm-s-ipython span.cm-variable-2 {  
 color: black;  
}  
  
.cm-s-ipython span.cm-variable-3 {  
 color: black;  
}  
  
  
.cm-s-default span.cm-bracket {  
 color: black;  
}  
  
.cm-s-ipython span.cm-operator {  
 color: black;  
}  
  
.cm-s-ipython span.cm-number {  
 color: black;  
}  
\*/  
  
</style>

In [155]:

import os  
from jupyter\_core.paths import jupyter\_config\_dir  
jupyter\_dir = jupyter\_config\_dir()  
print(jupyter\_dir)  
print(os.path.exists(jupyter\_dir))

C:\Users\pat\.jupyter  
False

A Deep Dive Into Comprehensions

[¶](#---------------------A-Deep-Dive-Into-Comprehensions-)

Patrick Barton barton.pj@gmail.com

</i>

Comprehensions provide an elegant and efficient means of producing collections in Python. They also allow on-the-fly application of filters.

Even if you don't use them yourself, it's worthwhile learning how to read them because you'll find plenty of them "in the wild", embedded in Python libraries you'll want to understand. And they can be might handy once you get the hang of them.

## Use Cases[¶](#Use-Cases)

Here are a couple situations where you may want to use comprehensions:

Case 1: You inherited a messy data table from your predecessor. It contains a bunch of values "Val\_1", "Val\_2", etc. each calculated for a different year. The column headings show up something like "Val\_1\_2025". You want to sort them and screen only for the year 2025, retaining the 'catch' in a new object.

With a comprehension, you could pull that off with a single, short line of code:

In [146]:

columns = ['Val\_3\_2020', 'Val\_1\_2025', 'Val\_3\_2023', 'Val\_3\_2024', 'Val\_3\_2024', 'Val\_3\_2023', 'Val\_3\_2025',   
 'Val\_2\_2023', 'Val\_1\_2023', 'Val\_2\_2025', 'Val\_1\_2022', 'Val\_3\_2023', 'Val\_2\_2024', 'Val\_3\_2021',   
 'Val\_1\_2023', 'Val\_3\_2024', 'Val\_3\_2021', 'Val\_1\_2020', 'Val\_3\_2024', 'Val\_1\_2022']  
   
[ col for col in sorted(columns) if '2025' in col]

Out[146]:

['Val\_1\_2025', 'Val\_2\_2025', 'Val\_3\_2025']

Case 2: You just need some sensible index names and placeholder values for a pandas **Series** object and want to do it efficiently. You could go:

In [107]:

import pandas as pd  
pd.Series(data = [val for val in range(5)], index = ["Year\_{}".format(yr) for yr in range(1905, 1910)])

Out[107]:

Year\_1905 0  
Year\_1906 1  
Year\_1907 2  
Year\_1908 3  
Year\_1909 4  
dtype: int64

A notable downside to comprehensions is that they can be difficult to decypher for those not familiar with the syntax. Don't believe me? Here's an example:

In [128]:

[y for level in three\_d\_array if level[2][2] %2 for x in level if x[1] > 3 for y in x if not y % 3 ]

A bit daunting, I'll admit. But fear not. When you've completed this unit, you'll be able to figure out what it produces with your pocket protector tied behind your back. Besides, there's nothing you can do with comprehensions that you can't do another way, so you don't really need to nail them. When you're done reading this unit, you'll be able to convert this into a (likely more easily-understood), verbose format and never have to look at again.

## Basics[¶](#Basics)

But let's begin at the beginning. Here's a simple example of a list comprehension:

In [108]:

iterable = "hey"  
[item for item in iterable]

Out[108]:

['h', 'e', 'y']

As you can see, we've done is created a new list object. It's identical to the following, more verbose, code (with a couple subtle differences):

In [109]:

as\_list = []  
for char in iterable:  
 as\_list.append(char)  
as\_list

Out[109]:

['h', 'e', 'y']

What are the differences? In the verbose mode we had to add the name 'as\_list' to the namespace and create an empty **list** instance. We also had to add the iterating variable 'item' to the project namespace.

Using the **list** comprehension, we had to do neither. The comprehension has its own "mini-namespace" which is created on the fly, and which goes out of context as soon as the comprehension has completed.

Now, lets build this up a bit at a time - it'll be easier to remember the steps this way.

[ ] #start with empty brackets; [ for item in iterable] #create a for loop (any valid expression will work); [ char for item in iterable] #put the thing you want to add to the list on the left.

The iterable object can be a **list**, **set**, **tuple**, generator expression, or just about anything with a **next**() method defined will do. Here's an example with a range object.

In [110]:

[i for i in range(10, 20, 2)]

Out[110]:

[10, 12, 14, 16, 18]

Now, since we've produced the iterating variable 'i', we can make any use of it to perform a "preprocessing operation" to produce the item added to the **list** at each iteration.

Let's say we wanted to find which printable character is mapped to a **range** of ordinal code points. Here's an easy way to manage it:

In [111]:

print([chr(code\_point) for code\_point in range(50, 65) ])

['2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<', '=', '>', '?', '@']

Good so far? Now, we can go upmarket a bit and add a filter to "screen in" a subset of the values produced by the iterating expression. The filter bit goes on the right.

Here's an example where we're looking only for the numbers evenly divisible by 3:

In [112]:

print([i for i in range(10, 20, 2) if not i % 3 ] )

[12, 18]

The filter can be any valid expression that can Python can evaluate as Boolean (**True**/**False**).

As an aside, Python is super-flexible in this regard. Any non-empty object and any non-zero number will evaluated as **True** when subjected to a Boolean test. The "%" is Python's modulo operator - it returns the remainder of a "floor" (rounded down) division e.g., 13 % 3 --> 1

## 2-D Objects (adding an extra internal variable)[¶](#2-D-Objects-(adding-an-extra-internal-variable))

Let's circle back to the notion of a comprehension's namespace. As mentioned earlier, the comprehension whistles up its own "mini-namespace" - the iterating variable 'item' is defined anew as the comprehension is calculated. To demonstrate:

In [113]:

item = "Snoopy the beagle" #setting 'item' in the main namespace  
[ item for item in {'x', 'y', 'z'} ] #re-using 'item' as an iterating variable  
print(item)

Snoopy the beagle

Introduction of a name to the internal namespace is done via the defining the iterating expression.

We've created an internal definition of 'item' and an internal definition of the **set** {'x', 'y', 'z'}. That's a seperate object from the object of the same name in the mainline code, and the value did not propogate.

If you need to create additional iterating variables, you need to create additional **for** statements.

Let's say you had an array-like structure, represented here as a **list** of **lists**, and you wanted to create a flattened version of it. You would want two iterating variables - one for the rows and the other for the columns. So you can go:

In [114]:

nested = [ [1,2,3],   
 [4,5,6],   
 [7,8, 9] ]  
flat = [y for x in nested for y in x]  
flat

Out[114]:

[1, 2, 3, 4, 5, 6, 7, 8, 9]

As you can see, even simple nested expressions can be hard on the eyes (not to mention your sanity). Here are a few tricks and tips for putting these together. Let's take the last expression apart a bit: outer loop inner loop flat = [ y for x in nested for y in x] We've defined two loops, an "outer loop" and an "inner loop". These are read starting from the left. The outer loop iterates through the rows of the array, each time producing a list like [1, 2, 3] and assigning it to the name 'x'. The(inner) loop processes each of these lists, producing one element at a time and assigning it to the name 'y'. The variable 'y' is then added to the output list. Still looks like gobbledy-gook? Yeah, I know. Here's another way to think about this: verbose version | list comprehension --------------- | ------------------ | flat\_verbose = [] #initialize | [ #initialize | y #add y here for x in nested: #outer loop | for x in nested #outer loop for y in x: #inner loop | for y in x #inner loop flat\_verbose.append(y) #add y here | | ] #complete the expression On the left side a verbose, nested pair of for statements generates a flattened version of the array one element at a time. Naturally, we had to initialize and empty list and use append() to tack each new element onto the end. On the right side the compression does exactly the same thing but the order of operations is a bit different because we add the new element at the beginning. The order of the iterating expressions is exactly the same, as is the syntax (sans the ":"). If it's easier, you can go ahead and write the comprehension in the 'indented format' - Python won't complain about the multiple lines because it assumes you'll close the [bracked expression] eventually. So this works:

In [159]:

[y  
 for x in nested  
 for y in x]

Out[159]:

[1, 2, 3, 4, 5, 6, 7, 8, 9]

## 3-D and Beyond[¶](#3-D-and-Beyond)

You can extend a nested **list** comprehension to an arbitrarily-large number of levels. Here we'll discuss the example presented at the beginning of this unit along with a concrete example.

In [116]:

#This is the array we'll flatten  
three\_d\_array = [[[11, 12, 13], [14, 15, 16], [17, 18, 19]], #top 'level'  
 [[21, 22, 23], [24, 25, 26], [27, 28, 29]], #middle 'level'  
 [[31, 32, 33], [34, 35, 36], [37, 38, 39]]] #bottom 'level'  
  
flat\_three\_d = [y for level in three\_d\_array if level[2][2] %2 for x in level if x[1] > 3 for y in x if not y % 3 ]  
  
flat\_three\_d

Out[116]:

[12, 15, 18, 21, 24, 27, 33, 36, 39]

Here's an annotated version of the comprehension in verbose form. You'll note that we have three iterating variables, 'level', 'x', and 'y' and a filter associated with each:

In [117]:

flat\_three\_d = []   
for level in three\_d\_array:  
 if level[2][2] %2: #is the last element of the last list in the 'level' odd?  
 for x in level:  
 if x[1] > 3: #is the second element of of each list larger than 3?  
 for y in x:  
 if not y %3: #is the individual element divisible by 3?  
 flat\_three\_d.append(y) #... if so, the number 'screens in' to the finished list  
   
flat\_three\_d

Out[117]:

[12, 15, 18, 21, 24, 27, 33, 36, 39]

Following the pattern in the 2-d example, we'll apply indentation to make it a bit more readable.

In [118]:

flat\_three\_d = [y #we'll add 'y' to the list here  
 for level in three\_d\_array #loop through the 'levels'  
 if level[2][2] %2 #...apply the 'level filter'  
 for x in level #loop thru the 'x' dimension  
 if x[1] > 3 #...apply the 'x' filter  
 for y in x #loop thru the 'y' dimension  
 if not y % 3 ] #...apply the 'y' filter  
flat\_three\_d

Out[118]:

[12, 15, 18, 21, 24, 27, 33, 36, 39]

The original should now make sense ... you just need to read it left to right, in chunks: \_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_ flat\_three\_d = [y for level in three\_d\_array if level[2][2] %2 for x in level if x[1] > 3 for y in x if not y % 3 ]

## Going the Other Way - Building Data Structures[¶](#Going-the-Other-Way---Building-Data-Structures)

Just as you can flatten high-dimensional data, you can also build data structures starting from a flat data source. Just for fun, let's try to make one of these:

In [119]:

target = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

So we're shooting for an array-like structure comprised of a **list** of **list** objects. Each of the internal **list** objects contains consecutive integers. Each new **list** picks off where the old one left off.

So, we're going to need a computational way to figure out the starting point of each internal **list**. Then populate the rest if it. One way to code this is:

In [129]:

#[ [start1 = 1, start1+1, start1+2 ], [start2 = 4, start2+1, start2\_2], ... ]  
  
outer = []  
for y in range(0, 10, 3): #number just before start 0, 3, 6, 9  
 inner = []  
 for x in range(1, 4): # numbers to add: 1, 2, 3  
 inner.append(x + y)  
 outer.append(inner)  
print(outer)   
print("Nailed it? {}.".format(outer==target))

[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]  
Nailed it? True.

Alternatively, we could use nested comprehensions where we use a outer 'y' loop to find the initial values; and the inner 'x' loop to populate individual **lists**. The comprehension should be structured thusly:

In [121]:

[ [x + y   
 for x in range (1, 4)]  
 for y in range(0, 10, 3)   
 ]

Out[121]:

[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

In [122]:

# more simply as a 'one-liner'  
  
#inner loop makes inner lists ...with every cycle of the outer loop  
[[x + y for x in range(1,4) ] for y in range(0, 10, 3)]

Out[122]:

[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

Creating a 3-d structure is just a little more complex. Here's a quick-and-dirty way to create a stack of array-like **'list-of-lists'** structure. This is handy code fragment to keep around for testing - you'll note that you can determine the x, y, and z coordinates based on the data values alone.

In [123]:

[ #new stack   
 [ #new row   
 [z\*100 + y\*10 + x for x in range(1,4)] #a list within a row ('inner loop')   
 for y in range(4) # for statement populates the list elements  
 ] #end of new row  
 for z in range(1,4) #for statement to add a new layer layer within the stack  
] #end of new stack

Out[123]:

[[[101, 102, 103], [111, 112, 113], [121, 122, 123], [131, 132, 133]],  
 [[201, 202, 203], [211, 212, 213], [221, 222, 223], [231, 232, 233]],  
 [[301, 302, 303], [311, 312, 313], [321, 322, 323], [331, 332, 333]]]

In [124]:

# ...as a 'one-liner'  
#<-----------------------------containg list ('the stack')-------------------------------------->  
# <------------------a row------------------------------------------->  
# <-------------an inner list----------->  
[ [ [z\*100 + y\*10 + x for x in range(1,4) ] for y in range(4)] for z in range(1,4) ]

Out[124]:

[[[101, 102, 103], [111, 112, 113], [121, 122, 123], [131, 132, 133]],  
 [[201, 202, 203], [211, 212, 213], [221, 222, 223], [231, 232, 233]],  
 [[301, 302, 303], [311, 312, 313], [321, 322, 323], [331, 332, 333]]]

## Other 'Flavors' of Comprehensions[¶](#Other-'Flavors'-of-Comprehensions)

Besides the **list** comprehension Python supports lots of others including comprehensions for **set** and **dict** objects, as well as one that creates a generator object. The good news is that they have about the same syntax. Here are some examples:

In [125]:

#Set comprehension - just use opposing {curly braces} instead of [square brackets]. Note that the set object deduplicates.  
{ value\*\*2 for value in range(-5, 5)}

Out[125]:

{0, 1, 4, 9, 16, 25}

In [126]:

#Dict comprehension - same as a set comprehension, but with a key:value pairing on the left.  
{char:ord(char) for char in ['A', '\*', 'x', '|']}

Out[126]:

{'A': 65, '\*': 42, 'x': 120, '|': 124}

In [127]:

#Generator comprehension - same as a list comprehension, but with (parens) instead of [square brackets].  
gen = (color for color in ['red', 'blue', 'green'])  
print (gen)  
  
print(next(gen))  
print(next(gen))  
print(next(gen))

<generator object <genexpr> at 0x00000283916FCFC0>  
red  
blue  
green

## Further Reading[¶](#Further-Reading)

If you're interested in learning more about comprehensions in general, or in exploring their history, application in other languages, etc. Wikipedia has an excellent article here:

<https://en.wikipedia.org/wiki/List_comprehension#History>