Python_Tips_Tricks_and_Hacks

2.4 Checking a Condition on Any or Every List Element

Say you want to check to see if *any* element in a list satisfies a condition (say, it's below 10). Before Python 2.5, you could do something like this:

#'s

```
1numbers = [1,10,100,1000,10000]
2if [number for number in numbers if number < 10]:
3    print 'At least one element is over 10'
4# Output: 'At least one element is over 10'
```

If none of the elements satisfy the condition, the list comprehension will create an empty list which evaluates as false. Otherwise, a non-empty list will be created, which evaluates as true. Strictly, you don't need to evaluate every item in the list; you could bail after the first item that satisfies the condition. The method above, then, is less efficient, but might be your only choice if you can't commit to only Python 2.5 and need to squeeze all of this logic in one expression.

With the new built-in `any` function introduced in Python 2.5, you can do the same thing cleanly and efficiently. *any* **is** actually smart enough to bail and return `True` after the first item that satisfies the condition. Here, I use a generator expression that returns a `True` or `False` value for each element, and pass it to `any`. The generator expression only computes these values as they are needed, and `any` only requests the values it needs [2]:

#'s

```
1numbers = [1,10,100,1000,10000]
2if any(number < 10 for number in numbers):
3    print 'Success'
4# Output: 'Success!'
```

Similarly, you can check if *every* element satisfies a condition. Without Python 2.5, you'll have to do something like this:

#'s

```
1numbers = [1,2,3,4,5,6,7,8,9]
2if len(numbers) == len([number for number in numbers if number < 10]):
3    print 'Success!'
4# Output: 'Success!'
```

Here we filter with a list comprehension and check to see if we still have as many elements. If we do, then all of the elements satisfied the condition. Again, this is less efficient than it could be, because there is no need to keep checking after the first element that doesn't satisfy the condition. Also again, without Python 2.5 it might be your only choice for fitting all the logic in one expression.

With Python 2.5, there's of course an easier way: the built-in `all` function. As you might expect, it's smart enough to bail after the first element that *doesn't* match, returning `False`. This method works just like the `any` method described above.

#'s

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

```
2if all(number < 10 for number in numbers):
3    print 'Success!'
4# Output: 'Success!'
```

3.1 Constructing Dictionaries with Keyword Arguments

When initially learning Python, I completely missed this alternate way to create dictionaries. Any keyword arguments you pass to the `dict` constructor are added to the newly created dictionary before returning. Of course, you are limited to the keys that can be made into keyword arguments: valid Python variable names. Here's an example:

#'s

```
1dict(a=1, b=2, c=3)
2# returns {'a': 1, 'b': 2, 'c': 3}
```

5.5 'Switch Statements' using Dictionaries of Functions

Ever miss the switch statement? As you probably know, Python doesn't really have a syntactical equivalent, unless you count repeated `elif`'s. What you might not know, though, is that you can replicate the behavior (if not the cleanliness) of the switch statement by creating a dictionary of functions keyed by the value you want to switch on.

For example, say you're handling keystrokes and you need to call a different function for each keystroke. Also say you've already defined these three functions:

#'s

```
 1def key_1_pressed():
 2    print 'Key 1 Pressed'
 3
 4def key_2_pressed():
 5    print 'Key 2 Pressed'
 6
 7def key_3_pressed():
 8    print 'Key 3 Pressed'
 9
10def unknown_key_pressed():
11    print 'Unknown Key Pressed'
```

In Python, you would typically use `elif`'s to choose a function:

#'s

```
 1keycode = 2
 2if keycode == 1:
 3   key_1_pressed()
 4elif keycode == 2:
 5   key_2_pressed()
 6elif number == 3:
 7   key_3_pressed()
 8else:
 9   unknown_key_pressed()
10# prints 'Key 2 Pressed'
```

But you could also throw all the functions in a dictionary, and key them to the value you're switching

on. You could even check see if the key exists and run some code if it doesn't:

#'s

```
1keycode = 2
2functions = {1: key_1_pressed, 2: key_2_pressed, 3: key_3_pressed}
3functions.get(keycode, unknown_key_pressed)()
```

6.3 Modifying Classes After Creation

You can add, modify, or delete a class property or method long after the class has been created, and even after it has been instantiated. Just access the property or method as `Class.attribute`. No matter when they were created, instances of the class will respect these changes:

#'s

```
 1class Class:
 2   def method(self):
 3         print 'Hey a method'
 4
 5instance = Class()
 6instance.method()
 7# prints 'Hey a method'
 8
 9def new_method(self):
10    print 'New method wins!'
11
12Class.method = new_method
13instance.method()
14# prints 'New method wins!'
```

Pretty awesome. But don't get carried away with modifying preexisting methods, it's bad form and can confuse the crap out of any objects using that class. On the other hand, adding methods is a lot less (but still somewhat) dangerous.