**Dead Simple Python**

Idiomatic Python for the Impatient Programmer

by Jason C. McDonald

errata updated to print 1

<table>
<thead>
<tr>
<th>Page</th>
<th>Error</th>
<th>Correction</th>
<th>Print corrected</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Nuitka can be used to transpile Python code C and C++ . . .</td>
<td>Nuitka can be used to transpile Python code to C and C++ . . .</td>
<td>Print 2</td>
</tr>
<tr>
<td>5</td>
<td>On Fedora, RHEL, or CentOS, you can run this:</td>
<td>On Fedora, RHEL, or CentOS, you can run this:</td>
<td>Print 2</td>
</tr>
<tr>
<td></td>
<td><code>sudo dnf python3 python3-pip</code></td>
<td><code>sudo dnf install python3 python3-pip</code></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>If I ran the linter again, I'd only see the other two linter errors:</td>
<td>If I ran the linter again, I'd only see the other three linter errors:</td>
<td>Print 2</td>
</tr>
<tr>
<td>47</td>
<td><code>foo %= 51 # value is now 42.0 (144.0 % 51)</code></td>
<td><code>foo %= 51 # value is now 42.0 (144.0 % 51)</code></td>
<td>Print 2</td>
</tr>
<tr>
<td>52</td>
<td>The assignment expression is enclosed in parentheses for readability, although I technically could have omitted them.</td>
<td>The parentheses in the assignment expression is important, as it controls what part of the expression is stored as the value of eggs. If I omitted the parentheses, the value <code>True</code> would be stored instead of an integer.</td>
<td>Print 2</td>
</tr>
<tr>
<td>57</td>
<td>First, if you want to wrap an expression in literal curly braces, you must use two curly braces ({{ }}) for every one you want displayed:</td>
<td>First, if you want to wrap an expression in literal curly braces, you must use two curly braces ({{ }}) for every one you want displayed, plus an additional pair to enable substitution.</td>
<td>Print 2</td>
</tr>
</tbody>
</table>
Mild: ["Lettuce", "Tomato", "Cheese", "Beef"]  
Mild: ["Lettuce", "Tomato", "Beef"]  
Default: ["Lettuce", "Tomato", "Beef"] | Print 2 |
<p>| 162  | In this case, I assume this is some sort of string, which I run through the static method <code>_encode()</code> I defined earlier and then store in the list <code>self._secrets</code>. | In this case, I assume this is some sort of string, which I run through the class method <code>encrypt()</code> I defined earlier and then store in the list <code>self._secrets</code>. | Print 2 |</p>
<table>
<thead>
<tr>
<th>Page</th>
<th>Error</th>
<th>Correction</th>
<th>Print corrected</th>
</tr>
</thead>
<tbody>
<tr>
<td>162</td>
<td>You actually don't need to define a deleter if you have no need for special behavior when the decorator is deleted. Consider what you want to happen if del is called on your decorator, such as when you are deleting an associated attribute that the property controls, if you can't think of anything, skip writing the deleter.</td>
<td>You actually don't need to define a deleter if you have no need for special behavior when the property is deleted. Consider what you want to happen if del is called on your property, such as when you are deleting an associated attribute that the property controls, if you can't think of anything, skip writing the deleter.</td>
<td>Print corrected</td>
</tr>
<tr>
<td>184</td>
<td>If case exceptions . . .</td>
<td>In case exceptions . . .</td>
<td>Print 2</td>
</tr>
<tr>
<td>224</td>
<td>Insertion</td>
<td>Counter is designed specifically for counting hashable objects; the object is the key, and the count is an integer value. Other languages call this type of collection a multiset. Multisets are not the same as counters, but are sometimes used in place of them, as a side effect of how multisets work.</td>
<td>Print 2</td>
</tr>
<tr>
<td>318</td>
<td>Figure update</td>
<td></td>
<td>Print 2</td>
</tr>
<tr>
<td>325</td>
<td>path.touch() Creates an empty file at path. Normally, nothing happens if it already exists. If the optional exist_ok= argument is False and the file exists, a FileExistsError is raised.</td>
<td>path.touch() Creates an empty file at path. If one already exists, it updates the access timestamp on file, but does nothing else. If the optional exist_ok= argument is False and the file exists, a FileExistsError is raised.</td>
<td>Print 2</td>
</tr>
<tr>
<td>358</td>
<td>left = int.from_bytes(left, byteorder=byteorder) right = int.from_bytes(right, byteorder=byteorder)</td>
<td>left = int.from_bytes(left, byteorder, signed=False) right = int.from_bytes(right, byteorder, signed=False)</td>
<td>Print 2</td>
</tr>
<tr>
<td>359</td>
<td>result = left &amp; right return result.to_bytes(size, byteorder, signed=True)</td>
<td>result = left &amp; right return result.to_bytes(size, byteorder, signed=False)</td>
<td>Print 2</td>
</tr>
</tbody>
</table>

Listing 12-38: bitwise_via_int.py:3

I bind the result of the bitwise operation to result. Finally, I convert result back to a bytes object, using the size I determined earlier, the byteorder passed to my function, and signed=True to handle conversion of any possible negative integer values. I return the resulting bytes-like object.

Listing 12-38: bitwise_via_int.py:3

I bind the result of the bitwise operation to result. Finally, I convert result back to a bytes object, using the size I determined earlier, and the byteorder passed to my function. I can safely assume signed=False, as left and right can only ever be positive integers.
In this case, I’ll create **two** more versions of the function: one that works with a string argument and another that works with either an integer or a floating-point number argument:

```python
    @__eq__.register
    def _(self, other: str):
        return self.symbol == other

    @overload
    def _(self, other: float):
        ...

    @__eq__.register
    def _(self, other: int):
        return self.number == other
```

The first of these methods accepts a string argument. The first parameter, the one being switched on, is annotated with a type hint for the expected type, which is a string (`str`) in this first case.

The second method here accepts either an integer or a float, and it is made possible with the `@typing.overload` decorator. When type hinting, you can mark one or more function headings with `@overload`, to indicate that they overload an upcoming function or method with the same name. The `Ellipsis (...)` is used in place of the suite of the overloaded method, so it can instead share the suite of the method below it. The function or method not decorated with `@overload` must come immediately after all the overloaded versions thereof.

In this case, I’ll create **three** more versions of the function: one that works with a string argument, another that works with a floating-point number, and a third with an integer:

```python
    @__eq__.register
    def _(self, other: str):
        return self.symbol == other

    @__eq__.register
    def _(self, other: float):
        return self.number == other

    @__eq__.register
    def _(self, other: int):
        return self.number == other
```

The first of these methods accepts a string argument. The first parameter, the one being switched on, is annotated with a type hint for the expected type, which is a string (`str`) in this first case. **The second method here accepts a float, and the third an int.**

When type hinting, you can **ordinarily** mark one or more function headings with a special `@typing.overload` to indicate that they overload an upcoming function or method with the same name. The `Ellipsis (...)` is used in place of the suite of the overloaded method, so it can instead share the suite of the method below it. The function or method not decorated with `@overload` must come immediately after all the overloaded versions thereof. I first thought to use this here, since the second and third functions had the same body. Unfortunately, `@overload` does not work with other decorators, so I could not use this technique here.
<table>
<thead>
<tr>
<th>Page</th>
<th>Error</th>
<th>Correction</th>
<th>Print corrected</th>
</tr>
</thead>
</table>
| 453  | ```  def __str__(self):    
                s = ""    
                formula = self.components.copy()    
                # Hill system    
                if 'C' in formula.keys():    
                    s += f"C{formula['C']}"    
                    del formula['C']    
                if 1 in formula.keys():    
                    s += f"H{formula['H']}"    
                    del formula['H'] ``` | ```  def __str__(self):    
                s = ""    
                formula = self.components.copy()    
                # Hill system    
                if 'C' in formula.keys():    
                    s += f"C{formula['C']}"    
                    del formula['C']    
                if 'H' in formula.keys():    
                    s += f"H{formula['H']}"    
                    del formula['H'] ``` | Print 2 |
| 627  | It can also be used on a number of Raspberry Pi and Arduino microcontrollers, as well as hardware from many other brands. | It can also be used on a number of Raspberry Pi and Arduino microcontrollers, as well as hardware from many other brands. | Print 2 |