Chapter 3: Indentation

Section 3.1: Simple example

For Python, Guido van Rossum based the grouping of statements on indentation. The reasons for this are explained in the first section of the "Design and History Python FAQ". Colons, `:`, are used to declare an indented code block, such as the following example:

```
class ExampleClass:
    #Every function belonging to a class must be indented equally
    def __init__(self):
        name = "example"

    def someFunction(self, a):
        #Notice everything belonging to a function must be indented
        if a > 5:
            return True
        else:
            return False

#If a function is not indented to the same level it will not be considers as part of the parent class
def separateFunction(b):
    for i in b:
    #Loops are also indented and nested conditions start a new indentation
        if i == 1:
            return True
    return False

separateFunction([2,3,5,6,1])
```

Spaces or Tabs?

The recommended indentation is 4 spaces but tabs or spaces can be used so long as they are consistent. ***Do not mix tabs and spaces in Python*** as this will cause an error in Python 3 and can causes errors in Python 2.

Section 3.2: How Indentation is Parsed

Whitespace is handled by the lexical analyzer before being parsed.

The lexical analyzer uses a stack to store indentation levels. At the beginning, the stack contains just the value 0, which is the leftmost position. Whenever a nested block begins, the new indentation level is pushed on the stack, and an "INDENT" token is inserted into the token stream which is passed to the parser. There can never be more than one "INDENT" token in a row (`IndentationError`).

When a line is encountered with a smaller indentation level, values are popped from the stack until a value is on top which is equal to the new indentation level (if none is found, a syntax error occurs). For each value popped, a "DEDENT" token is generated. Obviously, there can be multiple "DEDENT" tokens in a row.

The lexical analyzer skips empty lines (those containing only whitespace and possibly comments), and will never generate either "INDENT" or "DEDENT" tokens for them.

At the end of the source code, "DEDENT" tokens are generated for each indentation level left on the stack, until just the 0 is left.

For example:

```
if foo:
    if bar:
        x = 42
else:
    print foo
```

is analyzed as:

```
<if> <foo> <:>                  [0]
<INDENT> <if> <bar> <:>         [0, 4]
<INDENT> <x> <=> <42>           [0, 4, 8]
<DEDENT> <DEDENT> <else> <:>    [0]
<INDENT> <print> <foo>          [0, 2]
<DEDENT>
```

The parser than handles the "INDENT" and "DEDENT" tokens as block delimiters.

Section 3.3: Indentation Errors

The spacing should be even and uniform throughout. Improper indentation can cause an `IndentationError` or cause the program to do something unexpected. The following example raises an `IndentationError`:

```
a = 7
if a > 5:
  print "foo"
else:
  print "bar"
 print "done"
```

Or if the line following a colon is not indented, an `IndentationError` will also be raised:

```
if True:
print "true"
```

If you add indentation where it doesn't belong, an `IndentationError` will be raised:

```
if  True:
    a = 6
        b = 5
```

If you forget to un-indent functionality could be lost. In this example `None` is returned instead of the expected `False`:

```
def isEven(a):
    if a%2 ==0:
        return True
        #this next line should be even with the if
        return False
print isEven(7)
```