Inline `if` Statements

- Shorter, more concise format than normal '`if`' statement
- Useful in cases were only a single statement is required

```
x = y if condition else z     # conditional assignment
```

Example:

```
# Set flag if update is needed
update = true if version != current_version else false
```

© 2014 Cisco and/or its affiliates. All rights reserved. Cisco Confidential

Nested `if` Statements

- `if` statements can be nested
- Be careful to match up your '`else`'s with correct '`if`'s

```
if condition-1:
   # code for when condition-1 is true

   if condition-2:
      # code for when condition-2 is true

else:
   # which 'if' is this the 'else' for?
   # indentation provides the answer
```

© 2014 Cisco and/or its affiliates. All rights reserved. Cisco Confidential

Nested `if` Statements Alternative

- If possible, use '`and`' and '`or`' to collapse nested if statements
- Easier to write, easier to read, less prone to unexpected behavior
- Can be include any number of conditions

```
if condition-1 and condition-2:
   # code for when both conditions are true

else:
   # code for when either condition is false
```

Precedence

- Comparison operations (==, !=, <, >, etc.) have equal precedence
- However `'and'` is higher precedence than `'or'`, and `'not'` is higher than both. Bitwise `OR`, `XOR`, and `AND` are higher still
- Use parenthesis '`(...)`' to avoid confusion and force desired order:

```
# Which condition is evaluated first?
if condition-1 and condition-2 or condition-3:

# Use parenthesis to avoid confusion
if condition-1 and (condition-2 or condition-3):
```

cisco
© 2014 Cisco and/or its affiliates. All rights reserved. Cisco Confidential

Inline if Statements

There may be situations in which the code blocks for your `if` statements are only one statement. In such cases, it can be useful to condense your code into an inline `if` statement. In these situations, the code begins with an assignment statement, followed by the `if` condition, followed by the `else` clause. The format in the longer case is:

```
if condition:
   x = y
else:
   x = z
```

The inline form is:

```
x = y if condition else z    # conditional assignment
```

As you can see, this example is a more compact version of the standard `if-else` statement. Once the structure is understood, the inline `if` statement provides a simple way to condense four Python statements into one. A real-world example is:

```
# Set flag if update is needed
update = true if version != current_version else false
```

The example is testing to see if the software version of a device is equal to the current version; if not, the 'update' flag is set to true, so that a report of non-compliant devices can be generated.

Nested if Statements

Sometimes it is necessary to test for multiple conditions, in order to make a decision. To support these use cases, Python allows for the nesting of if statements.

However, it is important to understand the dangers in creating code that does this type of nesting—or sometimes, even more nesting. The issue has to do with correctly creating the `else` statements with the `if` statements to which they correspond. When you consider that some `if` statements can have no else statements, as in the example below, it can get confusing to untangle the logic of such constructs.

```
   if condition-1:
      # code for when condition-1 is true
      if condition-2:
         # code for when condition-2 is true
   else:
      # which 'if' is this the 'else' for?
      # indentation provides the answer
```

Some languages rely on curly braces or `begin-end` or just `end` to identify specific code blocks. Python does not use these constructs, but relies on indentation which makes it easier to visually see the beginning and end of code blocks to insure that `if-else` statements are correctly structured.

```
if version.major == current_major_version:
    if version.minor == current_minor_version:
        print(dev_info.name + ': version is up-to-date')
    else:
        print(dev_info.name + ': minor version is not current!')
else:
    print(dev_info.name + ': major version is not current!')
```

You should still take care when creating nested `if` statements, however, as they can be difficult to logically follow, even with enforced indentation as in Python. Sometimes it will be preferable to use multiple conditions on the same `if` statement.

Nested if Statement Alternative

As an alternative to nested `if` statements, consider putting multiple conditions in the initial `if` statement, where appropriate.

Python provides the Boolean operations `and`, `or`, and `not`. The `not` operation works on a single item, but `and` and `or` are useful for combining comparison values on a single `if` statement.

```
   if condition-1 and condition-2:
      # code for when both conditions are true
   else:
      # code for when either condition is false
```

The nested version check example that was shown previously could be written as follows.

```
if (version.major == current_major_version) and (version.minor == current_minor_ve
    print(dev_info.name + ': version is up-to-date')
else:
    print(dev_info.name + ': version is not current!')
```

If it was desired to highlight whether the major or minor version was out-of-date, the following code could be used:

```
if (version.major == current_major_version) and (version.minor == current_minor_ve
    print(dev_info.name + ': version is up-to-date')
elif version.major != current_major_version):
    print(dev_info.name + ': major version is not current!')
elif version.minor != current_minor_version:
    print(dev_info.name + ': only the minor version is not current!')
```

There are multiple ways to construct this logic; however, the purpose of this simple example is to show that grouping conditions together in the same `if` statement can make the code more readable, easier to understand, and debug.

Precedence

You can combine as many conditionals as you desire, using `and` and `or`. In order to organize the evaluation of your conditional, you will need to understand precedence rules, or explicitly specify precedence using parenthesis.

Precedence refers to the order in which Python will evaluate sets of comparisons in a conditional statement.

A complete table of precedence can be found in many places online; a simplified version is as follows

- Arithmetic comparison operators have equal precedence, and are evaluated left-to-right.
- Boolean operator precedence is, in order: `not`, `and`, `or`.
- Bitwise operators (`OR`, XOR, `AND`) are higher precedence than Boolean operators.

The good news is you do not need to memorize the table of precedence, because parentheses can be used to explicitly instruct Python what is your preferred order of evaluation. Parentheses are safest because they override any implicit precedence rules.

Calculate the number of routes that egress an interface:

Output of “show ip route” is directed to file labelled “ip-routes”

Terminal - cisco@cisco-python: /var/local/PyNE/labs/sections/section09

```
from pprint import pprint
import re

# Create regular expression to match Gigabit interface names
gig_pattern = re.compile('(GigabitEthernet)([0-9]\/[0-9]\/[0-9]\/[0-9])')

routes = {} # Create dictionary to hold number of routes per interface

# Read all lines of IP routing information
file = open('ip-routes','r')
for line in file:

    match = gig_pattern.search( line ) # Match for Gigabit Ethernet

    # Check to see if we matched the Gig Ethernet string
    if match:
        intf = match.group(2) # get the interface from the match
        routes[intf] = routes[intf]+1 if intf in routes else 1
    else:
        continue

print ''
print 'Number of routes per interface'
print '------------------------------'
pprint(routes)
print '' # Print final blank line
```