ciscoconfparse Documentation

Release 1.2.47

David Michael Pennington

May 15, 2017

Contents

Contents:

CHAPTER 1

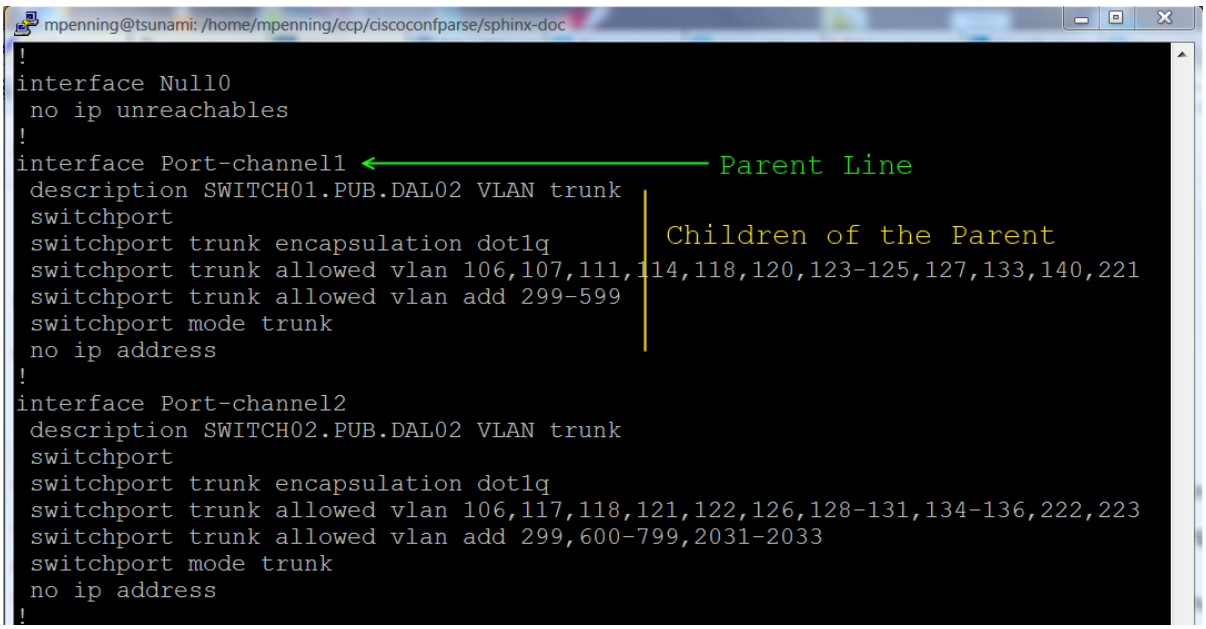
Introduction

Overview

ciscoconfparse is a Python library, which parses through Cisco IOS-style configurations. It can:

- Audit existing router / switch / firewall / wlc configurations
- Retrieve portions of the configuration
- Modify existing configurations
- Build new configurations

The library examines an IOS-style config and breaks it into a set of linked parent / child relationships; each configuration line is stored in a different *IOSCfgLine* object.



Then you issue queries against these relationships using a familiar

Fig. 1.1: Figure 1, An Example of Parent-line / Child-line relationships

family syntax model. Queries can either be in the form of a simple string, or you can use regular expressions. The API provides powerful query tools, including the ability to find all parents that have or do not have children matching a certain criteria.

The package also provides a set of methods to query and manipulate the *IOSCfgLine* objects themselves. This gives you a flexible mechanism to build your own custom queries, because the *IOSCfgLine* objects store all the parent / child hierarchy in them.

What is ciscoconfparse good for?

After several network evolutions, you may have a tangled mess of conflicting or misconfigured Cisco devices. Misconfigurations of proxy-arp, static routes, FHRP timers, routing protocols, duplicated subnets, cdp, console passwords, or aaa schemes have a measurable affect on up time and beg for a tool to audit them. However, manually scrubbing configurations is a long and error-prone process.

Audits aren't the only use for ciscoconfparse. Let's suppose you are working on a design and need a list of dot1q trunks on a switch with more than 400 interfaces. You can't grep for them because you need the interface names of layer2 trunks; the interface name is stored on one line, and the trunk configuration is stored somewhere below the interface name. With ciscoconfparse, it's really this easy...

```
>>> from ciscoconfparse import CiscoConfParse
>>> parse = CiscoConfParse('/tftpboot/largeConfig.conf')
>>> trunks = parse.find_parents_w_child("^interface", "switchport trunk")
>>> for intf in trunks:
...     print intf
interface GigabitEthernet 1/7
interface GigabitEthernet 1/23
interface GigabitEthernet 1/24
interface GigabitEthernet 1/30
interface GigabitEthernet 3/2
interface GigabitEthernet 5/10
<and so on...>
```

So you may be saying, that all sounds great, but I have no idea what you did with that code up there. If so, don't worry... There is a tutorial following this intro. For more depth, I highly recommend Dive into Python and Dive into Python3.

We don't have Ciscos

Don't let that stop you. CiscoConfParse parses anything that has a Cisco IOS style of configuration, which includes:

- Cisco IOS, Cisco Nexus, Cisco IOS-XR, Cisco IOS-XE, Aironet OS, Cisco ASA, Cisco CatOS
- Arista EOS
- Brocade
- HP Switches
- Force 10 Switches

- Dell PowerConnect Switches
- Extreme Networks
- Enterasys

As of CiscoConfParse 1.2.4, you can parse brace-delimited configurations into a Cisco IOS style (see Github Issue #17), which means that CiscoConfParse understands these configurations too:

- Juniper Networks Junos, and Screenos
- F5 Networks configurations

Quotes

These are a few selected public mentions about CiscoConfParse; I usually try not to share private emails without asking, thus the quotes aren't long at this time.

What's new in version 1.0.0

I wrote `ciscoconfparse` seven years ago as literally my first Python project; through the years, my understanding of Python improved, and I also found many missing features along the way. Some of these features, like changing a configuration after it was parsed, required non-trivial changes to the whole project.

Starting in version 0.9, I initiated a major rewrite; several important changes were made:

- Python3 compatibility; Python2.4 deprecation
- Major improvement in config parsing speed
- Much better unit-test coverage
- Too many bug fixes to count
- New feature - `ciscoconfparse` inserts, deletes and appends config lines
- Rearchitected the library, with an eye towards more future improvements
- Revisions in scripting flow. All users are encouraged to use `IOSCfgLine()` objects whenever possible. Typically, you'll start by matching them with `find_objects()`. Working directly with `IOSCfgLine()` objects makes your scripts less complicated and it also makes them faster than using legacy `ciscoconfparse` syntax.

CHAPTER 2

History and Python Apologetic

`CiscoConfParse()` was born from audit requirements. When I first built the module, I was contracting for a company with hundreds of devices; PCI compliance obligated us to perform security audits on the configs and management wanted it done quarterly. Our company was supposed to have an automated tool, but nobody could get it to work. I offered to build an audit and diff script instead of our entire team spending hundreds of man-hours on a manual task each quarter.

At first, I tried using a canned Perl config-parsing library; it was great when it worked, but the library suffered from mysterious crashes on certain configs. I tried auditing the troublesome configs manually, but dealing with the crashes put me behind schedule. I reached a point where I realized the audit results were going to be late if something didn't change, so I wrote the author for help, but he literally said that he wasn't really sure how the library works.[1]

With the deadline approaching, I wound up spending a full weekend of my own time writing my first endeavor in Python. It worked so well, I found myself building similar tools for other accounts that weren't even mine. After more work, I ultimately I published this as open-source software. ciscoconfparse is available to anyone who wants to invest a little effort on the front-end. Many companies in the US and Europe are already using it to audit their configs; I only ask that you drop me a line[2] and let me know if you like it and how I can improve the library.

[1] *This is not so much a slam on the module or author; it's part of Perl syntax. After six months, most people have a hard time remembering the meaning of those quirky idioms that make their code tick. Perl's syntax, and its convoluted error messages are why I left ten previous years of Perl experience behind me, and started fresh with Python in 2007.*

[2] *mike [~at~] pennington [~dot~] net*

CHAPTER 3

CiscoConfParse Installation and Python Basics

A note about Python

If you are coming from Perl or another language (many people do), you may not be familiar with Python's interpreter interface. To access the interpreter, just issue `python` at the Windows or unix command-line; this drops you into an interactive interpreter, where you can issue python commands. Use `quit()` to leave the interpreter.

When you see >>> preceding python statements, that means the example is run from within the python interpreter.

```
>>>
>>> print "Hello world"
```

If you don't see >>> preceding python statements, that means the example is run from a file saved to disk.

Using the Python in Unix

This is a "Hello World" example from within a unix Python interpreter.

```
[mpenning@mpenning-S10 ~]$ which python
/usr/local/bin/python
[mpenning@mpenning-S10 ~]$ python
Python 2.5.2 (r252:60911, Dec  5 2008, 11:57:32)
[GCC 3.4.6 [FreeBSD] 20060305] on freebsd6
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> print "Hello world"
Hello world
>>> quit()
[mpenning@mpenning-S10 ~]$
```

The same commands could be used in an executable script (mode 755) saved to disk... and run from the unix shell.

```
#!/usr/bin/env python

print "Hello world"
```

Using the Python in Windows

Please see the official Python on Windows documentation.

Using ciscoconfparse

Once you know how to find and use python on your system, it's time to ensure you have a copy of *ciscoconfparse*. Many of the examples assume you have imported *CiscoConfParse* at the interpreter before you start...

```
>>> from ciscoconfparse import CiscoConfParse
```

Try importing *CiscoConfParse* in the python interpreter now. If it doesn't work, then you'll need to install ciscoconfparse.

Installing ciscoconfparse

ciscoconfparse needs Python versions 2.6, 2.7 or 3.2+; the OS should not matter. If you want to run it under a Python virtualenv, it's been heavily tested in that environment as well.

You can check your python version with the `-V` switch...

```
[mpenning@Mudslide ~]$ python -V
Python 2.7.3
[mpenning@Mudslide ~]$
```

The best way to get ciscoconfparse is with pip or setuptools.

Install with pip

If you already have pip, you can install as usual:

Alternatively you can install with pip:

```
pip install --upgrade ciscoconfparse
```

If you have a specific version of ciscoconfparse in mind, you can specify that at the command-line

```
pip install ciscoconfparse==1.2.39
```

Install with setuptools

If you don't have pip, you can use setuptools...

```
# Substitute whatever ciscoconfparse version you like...
easy_install -U ciscoconfparse
```

If you have a specific version of ciscoconfparse in mind, you can specify that at the command-line

```
easy_install -U ciscoconfparse==1.2.39
```

Install from the source

If you don't have either pip or setuptools, you can download the ciscoconfparse compressed tarball, extract it and run the `setup.py` script in the tarball:

```
python setup.py install
```

Github and Bitbucket

If you're interested in the source, you can always pull from the github repo or bitbucket repo:

- From bitbucket (this also assumes you have mercurial):

```
hg init
hg clone https://bitbucket.org/mpenning/ciscoconfparse
```

- From github:

```
git clone git://github.com//mpenning/ciscoconfparse
```

CHAPTER 4

CiscoConfParse Tutorial

This is a brief tutorial which will cover the features that most *CiscoConfParse* users care about. We make a couple of assumptions throughout this tutorial...

- You already know a scripting language like Python or Perl
- You (naturally) have a basic understanding of Cisco IOS

Contents:

CiscoConfParse Fundamentals

IOS Parent-child relationships

CiscoConfParse() reads an IOS configuration and breaks it into a list of parent-child relationships. Used correctly, these relationships can reveal a lot of useful information. The concept of IOS parent and child is pretty intuitive, but we'll go through a simple example for clarity.

Note: CiscoConfParse assumes the configuration is in the *exact format* rendered by Cisco IOS devices when you use `show runn` or `show start`.

Line 1 is a parent:

```
policy-map QOS_1
 class GOLD
  priority percent 10
 class SILVER
  bandwidth 30
  random-detect
 class default
!
```

Child lines are indented more than parent lines; thus, lines 2, 4 and 7 are children of line 1:

```
policy-map QOS_1
 class GOLD
  priority percent 10
 class SILVER
  bandwidth 30
  random-detect
 class default
!
```

Furthermore, line 3 (highlighted) is a child of line 2:

```
policy-map QOS_1
 class GOLD
  priority percent 10
 class SILVER
  bandwidth 30
  random-detect
 class default
!
```

In short:

- Line 1 is a parent, and its children are lines 2, 4, and 7.
- Line 2 is also a parent, and it only has one child: line 3.

CiscoConfParse() uses these parent-child relationships to build queries. For instance, you can find a list of all parents with or without a child; or you can find all the configuration elements that are required to reconfigure a certain class-map.

IOSCfgLine objects

When *CiscoConfParse()* reads a configuration, it stores parent-child relationships as a special *IOSCfgLine* object. These objects are very powerful.

IOSCfgLine objects remember:

- The original IOS configuration line
- The parent configuration line
- All child configuration lines

IOSCfgLine objects also know about child indentation, and they keep special configuration query methods in the object itself. For instance, if you found an *IOSCfgLine* object with children, you can search the children directly from the parent by using *re_search_children()*.

Example: Retrieving text from an IOSCfgLine object

This example:

- Parses through a configuration
- Finds an *IOSCfgLine* object with *find_objects()*
- Retrieves the configuration text from that object (highlighted in yellow)

```
>>> from ciscoconfparse import CiscoConfParse
>>> parse = CiscoConfParse([
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.5 255.255.255.252'
...     ])
>>> for obj in parse.find_objects(r"interface"):
...     print "Object:", obj
...     print "Config text:", obj.text
...
Object: <IOSCfgLine # 1 'interface Serial1/0'>
Config text: interface Serial1/0
>>>
>>> quit()
[mpenning@tsunami ~]$
```

In the example, `obj.text` refers to the *IOSCfgLine* `text` attribute, which retrieves the text of the original IOS configuration statement.

Baseline configuration for these examples

This tutorial will run all the queries against a sample configuration, which is shown below.

```
! Filename: /tftpboot/bucksnort.conf
!
policy-map QOS_1
 class GOLD
  priority percent 10
 class SILVER
  bandwidth 30
  random-detect
 class default
!
interface Ethernet0/0
 ip address 1.1.2.1 255.255.255.0
 no cdp enable
!
interface Serial1/0
 encapsulation ppp
 ip address 1.1.1.1 255.255.255.252
!
interface Serial1/1
 encapsulation ppp
 ip address 1.1.1.5 255.255.255.252
 service-policy output QOS_1
!
interface Serial1/2
 encapsulation hdlc
 ip address 1.1.1.9 255.255.255.252
!
class-map GOLD
 match access-group 102
class-map SILVER
 match protocol tcp
!
```

Example Usage: Finding interface names that match a substring

The following script will load a configuration file from `/tftpboot/bucksnort.conf` and use *`find_objects()`* to find the Serial interfaces.

Note that the `^` symbol at the beginning of the search string is a regular expression; `^interface Serial` tells python to limit the search to lines that *begin* with `interface Serial`.

```
>>> from ciscoconfparse import CiscoConfParse
>>> parse = CiscoConfParse("/tftpboot/bucksnort.conf")
>>> serial_objs = parse.find_objects("^interface Serial")
```

The assuming we use the configuration in the example above, *`find_objects()`* scans the configuration for matching config objects and stores a list of *`IOSCfgLine`* objects in `serial_objs`.

```
>>> serial_objs
[<IOSCfgLine # 14 'interface Serial1/0'>,
<IOSCfgLine # 18 'interface Serial1/1'>,
<IOSCfgLine # 23 'interface Serial1/2'>]
```

As you can see, the config statements are stored inside *`IOSCfgLine`* objects. If you want to access the text inside the *`IOSCfgLine`* objects, just call their `text` attribute. For example...

```
>>> for obj in serial_objs:
...     print obj.text
...
interface Serial1/0
interface Serial1/1
interface Serial1/2
```

Going forward, I will assume that you know how to use regular expressions; if you would like to know more about regular expressions, O'Reilly's Mastering Regular Expressions book is very good.

Example Usage: Finding parents with a specific child

Suppose we need to find interfaces with the `QOS_1` service-policy applied outbound...

Method 1: for-loop to iterate over objects and search children

```
>>> parse = CiscoConfParse("/tftpboot/bucksnort.conf")
>>> all_intfs = parse.find_objects(r"^interf")
>>> qos_intfs = list()
>>> for obj in all_intfs:
...     if obj.re_search_children(r"service-policy\soutput\sQOS_1"):
...         qos_intfs.append(obj)
...
>>> qos_intfs
[<IOSCfgLine # 18 'interface Serial1/1'>]
```

This script iterates over the interface objects, and searches the children for the qos policy. It's worth mentioning that Python also has something called a list-comprehension, which makes the script for this task a little more compact...

Method 2: list-comprehension to iterate over objects and search children

```
>>> parse = CiscoConfParse("/tftpboot/bucksnort.conf")
>>> qos_intfs = [obj for obj in parse.find_objects(r"^interf") \
...     if obj.re_search_children(r"service-policy\soutput\sQOS_1")]
...
>>> qos_intfs
[<IOSCfgLine # 18 'interface Serial1/1'>]
```

Method 3: `find_objects_w_child()`

```
>>> parse = CiscoConfParse("/tftpboot/bucksnort.conf")
>>> qos_intfs = parse.find_objects_w_child(parentspec=r"^interf", \
...     childspec=r"service-policy\soutput\sQOS_1")
...
>>> qos_intfs
[<IOSCfgLine # 18 'interface Serial1/1'>]
```

You can choose any of these methods to accomplish your task... some might question why we cover the first two methods when *find_objects_w_child()* solves the problem completely. In this case, they have a point; however, *find_objects_w_child()* is much slower when you have more than one child line to inspect per interface, because *find_objects_w_child()* performs a line-by-line search of the whole configuration line each time it is called. By contrast, Method 1 is more efficient because you could simply call *re_search_children()* multiple times for each interface object. *re_search_children()* only searches the child lines of that *IOSCfgLine()* interface object.

Example Usage: Finding parents *without* a specific child

Let's suppose you wanted a list of all interfaces that have CDP enabled; this implies a couple of things:

1. CDP has not been disabled globally with `no cdp run`
2. The interfaces in question are not configured with `no cdp enable`

find_objects_wo_child() is a function to find parents without a specific child; it requires arguments similar to *find_objects_w_child()*:

- The first argument is a regular expression to match the parents
- The second argument is a regular expression to match the child's *exclusion*

Since we need to find parents that do not have `no cdp enable`, we will use *find_objects_wo_child()* for this query. Note that the script below makes use of a special property of python lists... empty lists test False in Python; thus, we can use `if not bool(parse.find_objects(r'no cdp run'))` to ensure that CDP is running globally on this device.

```
>>> parse = CiscoConfParse("/tftpboot/bucksnort.conf")
>>> if not bool(parse.find_objects(r'no cdp run')):
...     cdp_intfs = parse.find_objects_wo_child(r'^interface',
...         r'no cdp enable')
```

Results:

```
>>> cdp_intfs
[<IOSCfgLine # 14 'interface Serial1/0'>, <IOSCfgLine # 18 'interface Serial1/1'>,
↪<IOSCfgLine # 23 'interface Serial1/2'>]
```

Example Usage: A Contrived Configuration Audit

Suppose you have a large switched network and need to run audits on your configurations; assume you need to build configurations which conform to the following criteria:

- Access switchports *must* be configured with `storm-control`
- Trunk ports *must not* have port-security
- Timestamps must be enabled on logging and debug messages

You should follow the following steps.

Assume that you start with the following Cisco IOS configuration saved as `short.conf` (All the interfaces need to be changed, to conform with audit requirements):

```
!
interface FastEthernet0/1
 switchport mode access
 switchport access vlan 532
!
interface FastEthernet0/2
 switchport mode trunk
 switchport trunk allowed 300,532
 switchport nonegotiate
 switchport port-security maximum 2
 switchport port-security violation restrict
 switchport port-security
!
interface FastEthernet0/3
 switchport mode access
 switchport access vlan 300
!
end
```

Next, we build this script to read and change the config:

```
from ciscoconfparse import CiscoConfParse

def standardize_intfs(parse):

    ## Search all switch interfaces and modify them
    #
    # r'^interface.+?thernet' is a regular expression, for ethernet intfs
    for intf in parse.find_objects(r'^interface.+?thernet'):

        has_stormcontrol = intf.has_child_with(r' storm-control broadcast')
        is_switchport_access = intf.has_child_with(r'switchport mode access')
        is_switchport_trunk = intf.has_child_with(r'switchport mode trunk')

        ## Add missing features
        if is_switchport_access and (not has_stormcontrol):
            intf.append_to_family(' storm-control action trap')
            intf.append_to_family(' storm-control broadcast level 0.4 0.3')

        ## Remove dot1q trunk misconfiguration...
        elif is_switchport_trunk:
            intf.delete_children_matching('port-security')
```

```
## Parse the config
parse = CiscoConfParse('short.conf')

## Add a new switchport at the bottom of the config...
parse.append_line('interface FastEthernet0/4')
parse.append_line(' switchport')
parse.append_line(' switchport mode access')
parse.append_line('!')
parse.commit()      # commit() **must** be called before searching again

## Search and standardize the interfaces...
standardize_intfs(parse)
parse.commit()      # commit() **must** be called before searching again

## I'm illustrating regular expression usage in has_line_with()
if not parse.has_line_with(r'^service\stimestamp'):
    ## prepend_line() adds a line at the top of the configuration
    parse.prepend_line('service timestamps debug datetime msec localtime show-timezone
↪')
    parse.prepend_line('service timestamps log datetime msec localtime show-timezone')

## Write the new configuration
parse.save_as('short.conf.new')
```

Normally, regular expressions should be used in `.has_child_with()`; however, you can technically get away with the bare strings that I used in `standardize_intfs()` in some cases. That said, regular expressions are more powerful, and reliable when searching text. Usage of the `has_line_with()` and `find_objects()` methods illustrate regular expression syntax.

After the script runs, the new configuration (`short.conf.new`) looks like this:

```
service timestamps log datetime msec localtime show-timezone
service timestamps debug datetime msec localtime show-timezone
!
interface FastEthernet0/1
 switchport mode access
 switchport access vlan 532
 storm-control broadcast level 0.4 0.3
 storm-control action trap
!
interface FastEthernet0/2
 switchport mode trunk
 switchport trunk allowed 300,532
 switchport nonegotiate
!
interface FastEthernet0/3
 switchport mode access
 switchport access vlan 300
 storm-control broadcast level 0.4 0.3
 storm-control action trap
!
interface FastEthernet0/4
 switchport
 switchport mode access
 storm-control broadcast level 0.4 0.3
 storm-control action trap
!
end
```

The script:

- *Added* an access switchport: `interface FastEthernet0/4`
- *Added* `storm-control` to Fa0/1, Fa0/3, and Fa0/4
- *Removed* `port-security` from Fa0/2
- *Added* `timestamps` to logs and debug messages

Example Usage: Build configuration diffs

Let's suppose we need to find all serial interfaces in a certain address range and configure them for the MPLS LDP protocol. We will assume that all serial interfaces in 1.1.1.0/24 need to be configured with LDP.

Baseline Configuration

This tutorial will run all the queries against a sample configuration, which is shown below.

```
! Filename: /tftpboot/bucksnort.conf
!
policy-map QOS_1
 class GOLD
  priority percent 10
 class SILVER
  bandwidth 30
  random-detect
 class default
!
interface Ethernet0/0
 ip address 1.1.2.1 255.255.255.0
 no cdp enable
!
interface Serial1/0
 encapsulation ppp
 ip address 1.1.1.1 255.255.255.252
!
interface Serial1/1
 encapsulation ppp
 ip address 1.1.1.5 255.255.255.252
 service-policy output QOS_1
!
interface Serial1/2
 encapsulation hdlc
 ip address 1.1.1.9 255.255.255.252
!
class-map GOLD
 match access-group 102
class-map SILVER
 match protocol tcp
!
access-list 101 deny tcp any any eq 25 log
access-list 101 permit ip any any
!
access-list 102 permit tcp any host 1.5.2.12 eq 443
access-list 102 deny ip any any
!
```

Diff Script

The script below will build a list of serial interfaces, check to see whether they are in the correct address range. If so, the script will build a diff to enable LDP.

```
from ciscoconfparse import CiscoConfParse

# Parse the original configuration
parse = CiscoConfParse('/tftpboot/bucksnort.conf')

# Build a blank configuration for diffs
cfgdiffs = CiscoConfParse([])

# Iterate over :class:`~IOSCfgLine` objects
for intf in parse.find_objects("^interface Serial"):

   ## Search children of the interface for 1.1.1
   if (intf.re_search_children(r"ip\saddress\s1\.1\.1")):
      cfgdiffs.append_line("!")
      cfgdiffs.append_line(intf.text)  # Add the interface text
      cfgdiffs.append_line(" mpls ip")
```

Result:

```
>>> cfgdiffs.ioscfg
['interface Serial1/0', ' mpls ip', 'interface Serial1/1', ' mpls ip', 'interface␣
↪Serial1/2', ' mpls ip']
>>> for line in cfgdiffs.ioscfg:
...     print line
...
!
interface Serial1/0
 mpls ip
!
interface Serial1/1
 mpls ip
!
interface Serial1/2
 mpls ip
>>>
```

CHAPTER 5

CiscoConfParse Legacy Syntax

This section will cover the legacy *CiscoConfParse()* syntax; these were the original methods before version 1.0.0; legacy methods always returned text strings. This makes them easier to learn, but harder to write complex scripts with. There is nothing wrong with continuing to use these methods; however, you will probably find that your scripts are more efficient if you use the newer methods that manipulate *IOSCfgLine()* objects, which were introduced in version 1.0.0.

Baseline configuration

This tutorial will run all the queries against a sample configuration, which is shown below.

```
! Filename: /tftpboot/bucksnort.conf
!
policy-map QOS_1
 class GOLD
  priority percent 10
 class SILVER
  bandwidth 30
  random-detect
 class default
!
interface Ethernet0/0
 ip address 1.1.2.1 255.255.255.0
 no cdp enable
!
interface Serial1/0
 encapsulation ppp
 ip address 1.1.1.1 255.255.255.252
!
interface Serial1/1
 encapsulation ppp
 ip address 1.1.1.5 255.255.255.252
 service-policy output QOS_1
```

```
!
interface Serial1/2
 encapsulation hdlc
 ip address 1.1.1.9 255.255.255.252
!
class-map GOLD
 match access-group 102
class-map SILVER
 match protocol tcp
!
access-list 101 deny tcp any any eq 25 log
access-list 101 permit ip any any
!
access-list 102 permit tcp any host 1.5.2.12 eq 443
access-list 102 deny ip any any
!
logging 1.2.1.10
logging 1.2.1.11
logging 1.2.1.12
```

Finding interface names that match a substring

The following script will load a configuration file from `/tftpboot/bucksnort.conf` and use *`find_lines()`* to find the Serial interfaces.

Note that the `^` symbol at the beginning of the search string is a regular expression; `^interface Serial` tells python to limit the search to lines that *begin* with `interface Serial`.

To find matching interface statements, use this code...

```
>>> from ciscoconfparse import CiscoConfParse
>>> parse = CiscoConfParse("/tftpboot/bucksnort.conf")
>>> serial_lines = parse.find_lines("^interface Serial")
>>> serial_lines
['interface Serial1/0', 'interface Serial1/1', 'interface Serial1/2']
```

Going forward, I will assume that you know how to use regular expressions; if you would like to know more about regular expressions, O'Reilly's Mastering Regular Expressions book is very good.

Finding parents with a specific child

The last example was a nice start, but if this was all *`CiscoConfParse`* could do, then it's easier to use `grep`.

Let's suppose you need to find all interfaces that are configured with `service-policy QOS_1` in the output direction. We will use *`find_parents_w_child()`* to search the config.

`find_parents_w_child()` requires at least two different arguments:

- The first argument is a regular expression to match the parents
- The second argument is a regular expression to match the child

If the arguments above match both the parent and child respectively, then *`find_parents_w_child()`* will add the parent's line to a list. This list is returned after *`find_parents_w_child()`* finishes analyzing the configuration.

In this case, we need to find parents that begin with `^interface` and have a child matching `service-policy output QOS_1`. One might wonder why we chose to put a caret (`^`) in front of the parent's regex, but not in

front of the child's regex. We did this because of the way IOS indents commands in the configuration. Interface commands always show up at the top of the heirarchy in the configuration; interfaces do not get indented. On the other hand, the commands applied to the interface, such as a service-policy *are* indented. If we put a caret in front of `service-policy output QOS_1`, it would not match anything because we would be forcing a beginning-of-the-line match. The search and result is shown below.

```
>>> parse = CiscoConfParse("/tftpboot/bucksnort.conf")
>>> qos_intfs = parse.find_parents_w_child( "^interf", "service-policy output QOS_1" )
```

Results:

```
>>> qos_intfs
['interface Serial1/1']
```

Finding parents *without* a specific child

Let's suppose you wanted a list of all interfaces that have CDP enabled; this implies a couple of things:

1. CDP has not been disabled globally with `no cdp run`
2. The interfaces in question are not configured with `no cdp enable`

`find_parents_wo_child()` is a function to find parents without a specific child; it requires arguments similar to *`find_parents_w_child()`*:

- The first argument is a regular expression to match the parents
- The second argument is a regular expression to match the child's *exclusion*

Since we need to find parents that do not have `no cdp enable`, we will use *`find_parents_wo_child()`* for this query. Note that the script below makes use of a special property of python lists... empty lists test False in Python; thus, we can use `if not bool(parse.find_lines('no cdp run'))` to ensure that CDP is running globally on this device.

```
>>> if not bool(parse.find_lines('no cdp run')):
...     cdp_intfs = parse.find_parents_wo_child('^interface', 'no cdp enable')
```

Results:

```
>>> cdp_intfs
['interface Serial1/0', 'interface Serial1/1', 'interface Serial1/2']
```

Finding children

Let's suppose you needed to look at the children of a particular parent, but you didn't want the children's children. *`find_children()`* was made for this purpose.

```
>>> children = parse.find_children('policy-map QOS_1')
```

Results:

```
>>> children
['policy-map QOS_1', ' class GOLD', ' class SILVER', ' class default']
```

If you *do* want the children (recursively), then use *`find_all_children()`*.

```
>>> all_children = parse.find_all_children('policy-map QOS_1')
```

```
>>> all_children
['policy-map QOS_1', ' class GOLD', '  priority percent 10', ' class SILVER', ' ␣
↪bandwidth 30', '  random-detect', ' class default']
```

CiscoConfParse options

Several of *CiscoConfParse*'s functions support one of these options:

- exactmatch
- ignore_ws

`exactmatch` - This can either be `True` or `False` (the default). When `exactmatch` is set `True`, CiscoConfParse requires an exact match of the whole string (instead of a sub-string match, which is the default).

`ignore_ws` - This can either be `True` or `False` (the default). When `ignore_ws` is set `True`, CiscoConfParse will ignore differences in whitespace between the query string and the IOS configuration.

Not all functions support the options above; please consult the API documentation for specifics.

Checking Passwords

Sometimes you find yourself wishing you could decrypt vty or console passwords to ensure that they conform to the corporate standard. *CiscoConfParse* comes with a `CiscoPassword` class that can decrypt some Cisco IOS type 7 passwords.

Note: Cisco IOS Type 7 passwords were never meant to be secure; these passwords only protect against shoulder-surfing. When you add users and enable passwords to your router, be sure to use Cisco IOS Type 5 passwords; these are much more secure and cannot be decrypted.

Warning: `CiscoPassword` also cannot decrypt all Type 7 passwords. If the passwords exceed a certain length, the algorithm I have ceases to work. An error is printed to the console when this happens. In a future version of the script I will raise a python error when this happens.

Simple example... let's suppose you have this configuration...

```
line con 0
 login
 password 107D3D232342041E3A
 exec-timeout 15 0
```

We need to ensure that the password on the console is correct. This is easy with the `CiscoPassword` class

```
>>> from ciscoconfparse import CiscoPassword
>>> dp = CiscoPassword()
>>> decrypted_passwd = dp.decrypt('107D3D232342041E3A')
```

Result:

```
>>> decrypted_passwd
'STZF5vuV'
```

CHAPTER 6

License and Copyright

CHAPTER 7

API

This part of the documentation covers all the significant Python classes and methods used in `CiscoConfParse()`.

Contents:

CiscoConfParse Object

class ciscoconfparse.**CiscoConfParse**(*config=''*, *comment='!'*, *debug=False*, *factory=False*, *linesplit_rgx='\r*\n+'*, *ignore_blank_lines=True*, *syntax='ios'*)

Parses Cisco IOS configurations and answers queries about the configs

Initialize CiscoConfParse.

Kwargs:

- config (list or str): A list of configuration statements, or a configuration file path to be parsed
- comment (str): A comment delimiter. This should only be changed when parsing non-Cisco IOS configurations, which do not use a ! as the comment delimiter. `comment` defaults to '!'. This value can hold multiple characters in case the config uses multiple characters for comment delimiters; however, the comment delimiters are always assumed to be one character wide
- debug (bool): `debug` defaults to False, and should be kept that way unless you're working on a very tricky config parsing problem. Debug output is not particularly friendly
- factory (bool): `factory` defaults to False; if set `True`, it enables a beta-quality configuration line classifier.
- linesplit_rgx (str): `linesplit_rgx` is used when parsing configuration files to find where new configuration lines are. It is best to leave this as the default, unless you're working on a system that uses unusual line terminations (for instance something besides Unix, OSX, or Windows)
- ignore_blank_lines (bool): `ignore_blank_lines` defaults to True; when this is set True, ciscoconfparse ignores blank configuration lines. You might want to set `ignore_blank_lines` to False if you intentionally use blank lines in your configuration (ref: Github Issue #2), or you are parsing configurations which naturally have blank lines (such as Cisco Nexus configurations).

- syntax (str): `syntax` defaults to 'ios'; You can choose from the following values: ios, nxos, asa

Attributes:

- comment_delimiter (str): A string containing the comment-delimiter
- ConfigObjs (*IOSConfigList*) : A custom list, which contains all parsed *IOSCfgLine* instances.
- all_parents (list) : A list of all parent *IOSCfgLine* instances.
- last_index (int) : An integer with the last index in `ConfigObjs`

Returns:

- An instance of a *CiscoConfParse* object

This example illustrates how to parse a simple Cisco IOS configuration with *CiscoConfParse* into a variable called `parse`. This example also illustrates what the `ConfigObjs` and `ioscfg` attributes contain.

```
>>> config = [
...     'logging trap debugging',
...     'logging 172.28.26.15',
...     ]
>>> parse = CiscoConfParse(config)
>>> parse
<CiscoConfParse: 2 lines / syntax: ios / comment delimiter: '!' / factory: False>
>>> parse.ConfigObjs
<IOSConfigList, comment='!', conf=[<IOSCfgLine # 0 'logging trap debugging'>,
↪<IOSCfgLine # 1 'logging 172.28.26.15'>]>
>>> parse.ioscfg
['logging trap debugging', 'logging 172.28.26.15']
>>>
```

append_line(*linespec*)

Unconditionally insert `linespec` (a text line) at the end of the configuration

Args:

- linespec (str): Text IOS configuration line

Returns:

- The parsed *IOSCfgLine* instance

atomic()

Call *atomic()* to manually fix up `ConfigObjs` relationships after modifying a parsed configuration. This method is slow; try to batch calls to *atomic()* if possible.

> **Warning:** If you modify a configuration after parsing it with *CiscoConfParse*, you *must* call *commit()* or *atomic()* before searching the configuration again with methods such as *find_objects()* or *find_lines()*. Failure to call *commit()* or *atomic()* on config modifications could lead to unexpected search results.

commit()

Alias for calling the *atomic()* method. This method is slow; try to batch calls to *commit()* if possible.

> **Warning:** If you modify a configuration after parsing it with *CiscoConfParse*, you *must* call *commit()* or *atomic()* before searching the configuration again with methods such as

> *find_objects()* or *find_lines()*. Failure to call *commit()* or *atomic()* on config modifications could lead to unexpected search results.

convert_braces_to_ios (*input_list*, *stop_width=4*)

delete_lines (*linespec*, *exactmatch=False*, *ignore_ws=False*)
Find all *IOSCfgLine* objects whose text matches linespec, and delete the object

find_all_children (*linespec*, *exactmatch=False*, *ignore_ws=False*)
Returns the parents matching the linespec, and all their children. This method is different than `find_children()`, because `find_all_children()` finds children of children. `find_children()` only finds immediate children.

Args:

- linespec (str): Text regular expression for the line to be matched

Kwargs:

- exactmatch (bool): boolean that controls whether partial matches are valid
- ignore_ws (bool): boolean that controls whether whitespace is ignored

Returns:

- list. A list of matching configuration lines

Suppose you are interested in finding all *archive* statements in the following configuration...

```
username ddclient password 7 107D3D232342041E3A
archive
 log config
  logging enable
  hidekeys
 path ftp://ns.foo.com//tftpboot/Foo-archive
!
```

Using the config above, we expect to find the following config lines...

```
archive
 log config
  logging enable
  hidekeys
 path ftp://ns.foo.com//tftpboot/Foo-archive
```

We would accomplish this by querying *find_all_children('^archive')*...

```
>>> from ciscoconfparse import CiscoConfParse
>>> config = ['username ddclient password 7 107D3D232342041E3A',
...           'archive',
...           ' log config',
...           '  logging enable',
...           '  hidekeys',
...           ' path ftp://ns.foo.com//tftpboot/Foo-archive',
...           '!',
...     ]
>>> p = CiscoConfParse(config)
>>> p.find_all_children('^archive')
```

```
['archive', ' log config', '  logging enable', '  hidekeys', ' path ftp://ns.
↪foo.com//tftpboot/Foo-archive']
>>>
```

find_blocks(*linespec*, *exactmatch=False*, *ignore_ws=False*)

Find all siblings matching the linespec, then find all parents of those siblings. Return a list of config lines sorted by line number, lowest first. Note: any children of the siblings should NOT be returned.

Args:

- linespec (str): Text regular expression for the line to be matched

Kwargs:

- exactmatch (bool): boolean that controls whether partial matches are valid
- ignore_ws (bool): boolean that controls whether whitespace is ignored

Returns:

- list. A list of matching configuration lines

This example finds *bandwidth percent* statements in following config, the siblings of those *bandwidth percent* statements, as well as the parent configuration statements required to access them.

```
!
policy-map EXTERNAL_CBWFQ
 class IP_PREC_HIGH
  priority percent 10
  police cir percent 10
    conform-action transmit
    exceed-action drop
 class IP_PREC_MEDIUM
  bandwidth percent 50
  queue-limit 100
 class class-default
  bandwidth percent 40
  queue-limit 100
policy-map SHAPE_HEIR
 class ALL
  shape average 630000
  service-policy EXTERNAL_CBWFQ
!
```

The following config lines should be returned:

```
policy-map EXTERNAL_CBWFQ
 class IP_PREC_MEDIUM
  bandwidth percent 50
  queue-limit 100
 class class-default
  bandwidth percent 40
  queue-limit 100
```

We do this by quering *find_blocks('bandwidth percent')*...

```
>>> from ciscoconfparse import CiscoConfParse
>>> config = ['!',
...           'policy-map EXTERNAL_CBWFQ',
...           ' class IP_PREC_HIGH',
```

```
...             '  priority percent 10',
...             '  police cir percent 10',
...             '    conform-action transmit',
...             '    exceed-action drop',
...             ' class IP_PREC_MEDIUM',
...             '  bandwidth percent 50',
...             '  queue-limit 100',
...             ' class class-default',
...             '  bandwidth percent 40',
...             '  queue-limit 100',
...             'policy-map SHAPE_HEIR',
...             ' class ALL',
...             '  shape average 630000',
...             '  service-policy EXTERNAL_CBWFQ',
...             '!',
...         ]
>>> p = CiscoConfParse(config)
>>> p.find_blocks('bandwidth percent')
['policy-map EXTERNAL_CBWFQ', ' class IP_PREC_MEDIUM', '  bandwidth percent 50
↪', '  queue-limit 100', ' class class-default', '  bandwidth percent 40', '␣
↪ queue-limit 100']
>>>
>>> p.find_blocks(' class class-default')
['policy-map EXTERNAL_CBWFQ', ' class IP_PREC_HIGH', ' class IP_PREC_MEDIUM',
↪' class class-default']
>>>
```

find_children(*linespec*, *exactmatch=False*, *ignore_ws=False*)

Returns the parents matching the linespec, and their immediate children. This method is different than `find_all_children()`, because `find_all_children()` finds children of children. `find_children()` only finds immediate children.

Args:

- linespec (str): Text regular expression for the line to be matched

Kwargs:

- exactmatch (bool): boolean that controls whether partial matches are valid
- ignore_ws (bool): boolean that controls whether whitespace is ignored

Returns:

- list. A list of matching configuration lines

Suppose you are interested in finding all immediate children of the *archive* statements in the following configuration...

```
username ddclient password 7 107D3D232342041E3A
archive
 log config
  logging enable
  hidekeys
 path ftp://ns.foo.com//tftpboot/Foo-archive
!
```

Using the config above, we expect to find the following config lines...

```
archive
 log config
 path ftp://ns.foo.com//tftpboot/Foo-archive
```

We would accomplish this by querying *find_children('^archive')*...

```
>>> from ciscoconfparse import CiscoConfParse
>>> config = ['username ddclient password 7 107D3D232342041E3A',
...           'archive',
...           ' log config',
...           '  logging enable',
...           '  hidekeys',
...           ' path ftp://ns.foo.com//tftpboot/Foo-archive',
...           '!',
...      ]
>>> p = CiscoConfParse(config)
>>> p.find_children('^archive')
['archive', ' log config', ' path ftp://ns.foo.com//tftpboot/Foo-archive']
>>>
```

find_children_w_parents (*parentspec*, *childspec*, *ignore_ws=False*)

Parse through the children of all parents matching parentspec, and return a list of children that matched the childspec.

Args:

- parentspec (str): Text regular expression for the line to be matched; this must match the parent's line
- childspec (str): Text regular expression for the line to be matched; this must match the child's line

Kwargs:

- ignore_ws (bool): boolean that controls whether whitespace is ignored

Returns:

- list. A list of matching child configuration lines

This example finds the port-security lines on FastEthernet0/1 in following config...

```
!
interface FastEthernet0/1
 switchport access vlan 532
 switchport port-security
 switchport port-security violation protect
 switchport port-security aging time 5
 switchport port-security aging type inactivity
 spanning-tree portfast
 spanning-tree bpduguard enable
!
interface FastEthernet0/2
 switchport access vlan 300
 spanning-tree portfast
 spanning-tree bpduguard enable
!
interface FastEthernet0/2
 duplex full
 speed 100
 switchport access vlan 300
 spanning-tree portfast
```

```
 spanning-tree bpduguard enable
!
```

The following lines should be returned:

```
switchport port-security
switchport port-security violation protect
switchport port-security aging time 5
switchport port-security aging type inactivity
```

We do this by quering *find_children_w_parents()*; we set our parent as *^interface* and set the child as *switchport port-security*.

```
>>> config = ['!',
...            'interface FastEthernet0/1',
...            ' switchport access vlan 532',
...            ' switchport port-security',
...            ' switchport port-security violation protect',
...            ' switchport port-security aging time 5',
...            ' switchport port-security aging type inactivity',
...            ' spanning-tree portfast',
...            ' spanning-tree bpduguard enable',
...            '!',
...            'interface FastEthernet0/2',
...            ' switchport access vlan 300',
...            ' spanning-tree portfast',
...            ' spanning-tree bpduguard enable',
...            '!',
...            'interface FastEthernet0/3',
...            ' duplex full',
...            ' speed 100',
...            ' switchport access vlan 300',
...            ' spanning-tree portfast',
...            ' spanning-tree bpduguard enable',
...            '!',
...      ]
>>> p = CiscoConfParse(config)
>>> p.find_children_w_parents('^interface\sFastEthernet0/1',            'port-
↪security')
[' switchport port-security', ' switchport port-security violation protect',
↪' switchport port-security aging time 5', ' switchport port-security aging␣
↪type inactivity']
>>>
```

find_interface_objects(*intfspec*, *exactmatch=True*)

Find all *IOSCfgLine* objects whose text is an abbreviation for `intfspec` and return the *IOSIntfLine* objects in a python list.

Note: The configuration *must* be parsed with `factory=True` to use this method

---

Args:

- intfspec (str): A string which is the abbreviation (or full name) of the interface

Kwargs:

- exactmatch (bool): Defaults to True; when True, this option requires `intfspec` match the whole interface name and number.

Returns:

- list. A list of matching `IOSIntfLine` objects

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial1/1',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config, factory=True)
>>>
>>> parse.find_interface_objects('Se 1/0')
[<IOSIntfLine # 1 'Serial1/0' info: '1.1.1.1/30'>]
>>>
```

find_lineage(*linespec*, *exactmatch=False*)

Iterate through to the oldest ancestor of this object, and return a list of all ancestors / children in the direct line. Cousins or aunts / uncles are *not* returned. Note, all children of this object are returned.

find_lines(*linespec*, *exactmatch=False*, *ignore_ws=False*)

This method is the equivalent of a simple configuration grep (Case-sensitive).

Args:

- linespec (str): Text regular expression for the line to be matched

Kwargs:

- exactmatch (bool): Defaults to False. When set True, this option requires `linespec` match the whole configuration line, instead of a portion of the configuration line.
- ignore_ws (bool): boolean that controls whether whitespace is ignored. Default is False.

Returns:

- list. A list of matching configuration lines

find_objects(*linespec*, *exactmatch=False*, *ignore_ws=False*)

Find all *IOSCfgLine* objects whose text matches `linespec` and return the *IOSCfgLine* objects in a python list. *find_objects()* is similar to *find_lines()*; however, the former returns a list of *IOSCfgLine* objects, while the latter returns a list of text configuration statements. Going forward, I strongly encourage people to start using *find_objects()* instead of *find_lines()*.

Args:

- linespec (str): A string or python regular expression, which should be matched

Kwargs:

- exactmatch (bool): Defaults to False. When set True, this option requires `linespec` match the whole configuration line, instead of a portion of the configuration line.
- ignore_ws (bool): boolean that controls whether whitespace is ignored. Default is False.

Returns:

- list. A list of matching `IOSCfgLine` objects

This example illustrates the difference between *find_objects()* and *find_lines()*.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial1/1',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> parse.find_objects(r'^interface')
[<IOSCfgLine # 1 'interface Serial1/0'>, <IOSCfgLine # 4 'interface Serial1/1
↪'>]
>>>
>>> parse.find_lines(r'^interface')
['interface Serial1/0', 'interface Serial1/1']
>>>
```

find_objects_dna(*dnaspec*, *exactmatch=False*)

Find all *IOSCfgLine* objects whose text matches `dnaspec` and return the *IOSCfgLine* objects in a python list.

Note: *find_objects_dna()* requires the configuration to be parsed with factory=True

---

Args:

- dnaspec (str): A string or python regular expression, which should be matched. This argument will be used to match dna attribute of the object

Kwargs:

- exactmatch (bool): Defaults to False. When set True, this option requires `dnaspec` match the whole configuration line, instead of a portion of the configuration line.

Returns:

- list. A list of matching `IOSCfgLine` objects

```
>>> config = [
...     '!',
...     'hostname MyRouterHostname',
...     '!',
...     ]
>>> parse = CiscoConfParse(config, factory=True, syntax='ios')
>>>
>>> obj_list = parse.find_objects_dna(r'Hostname')
>>> obj_list
[<IOSHostnameLine # 1 'MyRouterHostname'>]
>>>
>>> # The IOSHostnameLine object has a hostname attribute
>>> obj_list[0].hostname
'MyRouterHostname'
>>>
```

find_objects_w_all_children(*parentspec*, *childspec*, *ignore_ws=False*)

Return a list of parent *IOSCfgLine* objects, which matched the `parentspec` and whose children match all elements in `childspec`. Only the parent *IOSCfgLine* objects will be returned.

Args:

- parentspec (str): Text regular expression for the *IOSCfgLine* object to be matched; this must match the parent's line
- childspec (str): A list of text regular expressions to be matched among the children

Kwargs:

- ignore_ws (bool): boolean that controls whether whitespace is ignored

Returns:

- list. A list of matching parent *IOSCfgLine* objects

This example uses `find_objects_w_child()` to find all ports that are members of access vlan 300 in following config...

```
!
interface FastEthernet0/1
 switchport access vlan 532
 spanning-tree vlan 532 cost 3
!
interface FastEthernet0/2
 switchport access vlan 300
 spanning-tree portfast
!
interface FastEthernet0/2
 duplex full
 speed 100
 switchport access vlan 300
 spanning-tree portfast
!
```

The following interfaces should be returned:

```
interface FastEthernet0/2
interface FastEthernet0/3
```

We do this by quering *find_objects_w_all_children()*; we set our parent as ^*interface* and set the childspec as ['switchport access vlan 300', 'spanning-tree portfast'].

```
>>> config = ['!',
...           'interface FastEthernet0/1',
...           ' switchport access vlan 532',
...           ' spanning-tree vlan 532 cost 3',
...           '!',
...           'interface FastEthernet0/2',
...           ' switchport access vlan 300',
...           ' spanning-tree portfast',
...           '!',
...           'interface FastEthernet0/3',
...           ' duplex full',
...           ' speed 100',
...           ' switchport access vlan 300',
...           ' spanning-tree portfast',
...           '!',
```

```
...     ]
>>> p = CiscoConfParse(config)
>>> p.find_objects_w_all_children('^interface',
...     ['switchport access vlan 300', 'spanning-tree portfast'])
...
[<IOSCfgLine # 5 'interface FastEthernet0/2'>, <IOSCfgLine # 9 'interface␣
↪FastEthernet0/3'>]
>>>
```

find_objects_w_child(*parentspec*, *childspec*, *ignore_ws=False*)

Return a list of parent *IOSCfgLine* objects, which matched the `parentspec` and whose children match `childspec`. Only the parent *IOSCfgLine* objects will be returned.

Args:

- parentspec (str): Text regular expression for the *IOSCfgLine* object to be matched; this must match the parent's line
- childspec (str): Text regular expression for the line to be matched; this must match the child's line

Kwargs:

- ignore_ws (bool): boolean that controls whether whitespace is ignored

Returns:

- list. A list of matching parent *IOSCfgLine* objects

This example uses `find_objects_w_child()` to find all ports that are members of access vlan 300 in following config...

```
!
interface FastEthernet0/1
 switchport access vlan 532
 spanning-tree vlan 532 cost 3
!
interface FastEthernet0/2
 switchport access vlan 300
 spanning-tree portfast
!
interface FastEthernet0/2
 duplex full
 speed 100
 switchport access vlan 300
 spanning-tree portfast
!
```

The following interfaces should be returned:

```
interface FastEthernet0/2
interface FastEthernet0/3
```

We do this by quering *find_objects_w_child()*; we set our parent as *^interface* and set the child as *switchport access vlan 300*.

```
>>> config = ['!',
...           'interface FastEthernet0/1',
...           ' switchport access vlan 532',
...           ' spanning-tree vlan 532 cost 3',
...           '!',
```

```
...             'interface FastEthernet0/2',
...             ' switchport access vlan 300',
...             ' spanning-tree portfast',
...             '!',
...             'interface FastEthernet0/3',
...             ' duplex full',
...             ' speed 100',
...             ' switchport access vlan 300',
...             ' spanning-tree portfast',
...             '!',
...         ]
>>> p = CiscoConfParse(config)
>>> p.find_objects_w_child('^interface',
...     'switchport access vlan 300')
...
[<IOSCfgLine # 5 'interface FastEthernet0/2'>, <IOSCfgLine # 9 'interface␣
↪FastEthernet0/3'>]
>>>
```

find_objects_w_missing_children(*parentspec*, *childspec*, *ignore_ws=False*)

find_objects_w_parents(*parentspec*, *childspec*, *ignore_ws=False*)

Parse through the children of all parents matching parentspec, and return a list of child objects, which matched the childspec.

Args:

- parentspec (str): Text regular expression for the line to be matched; this must match the parent's line
- childspec (str): Text regular expression for the line to be matched; this must match the child's line

Kwargs:

- ignore_ws (bool): boolean that controls whether whitespace is ignored

Returns:

- list. A list of matching child objects

This example finds the object for "ge-0/0/0" under "interfaces" in the following config...

```
interfaces
    ge-0/0/0
        unit 0
            family ethernet-switching
                port-mode access
                vlan
                    members VLAN_FOO
    ge-0/0/1
        unit 0
            family ethernet-switching
                port-mode trunk
                vlan
                    members all
                native-vlan-id 1
    vlan
        unit 0
            family inet
                address 172.16.15.5/22
```

The following object should be returned:

```
<IOSCfgLine # 7 '    ge-0/0/1' (parent is # 0)>
```

We do this by quering *find_childobj_w_parents()*; we set our parent as *^s*interface* and set the child as *^s+ge-0/0/1*.

```
>>> config = ['interfaces',
...            '    ge-0/0/0',
...            '        unit 0',
...            '            family ethernet-switching',
...            '                port-mode access',
...            '                vlan',
...            '                    members VLAN_FOO',
...            '    ge-0/0/1',
...            '        unit 0',
...            '            family ethernet-switching',
...            '                port-mode trunk',
...            '                vlan',
...            '                    members all',
...            '                native-vlan-id 1',
...            '    vlan',
...            '        unit 0',
...            '            family inet',
...            '                address 172.16.15.5/22',
...      ]
>>> p = CiscoConfParse(config)
>>> p.find_objects_w_parents('^\s*interfaces',            r'\s+ge-0/0/1')
[<IOSCfgLine # 7 '    ge-0/0/1' (parent is # 0)>]
>>>
```

find_objects_wo_child(*parentspec*, *childspec*, *ignore_ws=False*)

Return a list of parent *IOSCfgLine* objects, which matched the `parentspec` and whose children did not match `childspec`. Only the parent *IOSCfgLine* objects will be returned. For simplicity, this method only finds oldest_ancestors without immediate children that match.

Args:

- parentspec (str): Text regular expression for the *IOSCfgLine* object to be matched; this must match the parent's line
- childspec (str): Text regular expression for the line to be matched; this must match the child's line

Kwargs:

- ignore_ws (bool): boolean that controls whether whitespace is ignored

Returns:

- list. A list of matching parent configuration lines

This example finds all ports that are autonegotiating in the following config...

```
!
interface FastEthernet0/1
 switchport access vlan 532
 spanning-tree vlan 532 cost 3
!
interface FastEthernet0/2
 switchport access vlan 300
 spanning-tree portfast
```

```
!
interface FastEthernet0/2
 duplex full
 speed 100
 switchport access vlan 300
 spanning-tree portfast
!
```

The following interfaces should be returned:

```
interface FastEthernet0/1
interface FastEthernet0/2
```

We do this by quering *find_objects_wo_child()*; we set our parent as *^interface* and set the child as *speedsd+* (a regular-expression which matches the word 'speed' followed by an integer).

```
>>> config = ['!',
...            'interface FastEthernet0/1',
...            ' switchport access vlan 532',
...            ' spanning-tree vlan 532 cost 3',
...            '!',
...            'interface FastEthernet0/2',
...            ' switchport access vlan 300',
...            ' spanning-tree portfast',
...            '!',
...            'interface FastEthernet0/3',
...            ' duplex full',
...            ' speed 100',
...            ' switchport access vlan 300',
...            ' spanning-tree portfast',
...            '!',
...        ]
>>> p = CiscoConfParse(config)
>>> p.find_objects_wo_child(r'^interface', r'speed\s\d+')
[<IOSCfgLine # 1 'interface FastEthernet0/1'>, <IOSCfgLine # 5 'interface␣
↪FastEthernet0/2'>]
>>>
```

find_parents_w_child(*parentspec*, *childspec*, *ignore_ws=False*)

Parse through all children matching childspec, and return a list of parents that matched the parentspec. Only the parent lines will be returned.

Args:

- parentspec (str): Text regular expression for the line to be matched; this must match the parent's line
- childspec (str): Text regular expression for the line to be matched; this must match the child's line

Kwargs:

- ignore_ws (bool): boolean that controls whether whitespace is ignored

Returns:

- list. A list of matching parent configuration lines

This example finds all ports that are members of access vlan 300 in following config...

```
!
interface FastEthernet0/1
 switchport access vlan 532
 spanning-tree vlan 532 cost 3
!
interface FastEthernet0/2
 switchport access vlan 300
 spanning-tree portfast
!
interface FastEthernet0/2
 duplex full
 speed 100
 switchport access vlan 300
 spanning-tree portfast
!
```

The following interfaces should be returned:

```
interface FastEthernet0/2
interface FastEthernet0/3
```

We do this by quering *find_parents_w_child()*; we set our parent as *^interface* and set the child as *switchport access vlan 300*.

```
>>> config = ['!',
...           'interface FastEthernet0/1',
...           ' switchport access vlan 532',
...           ' spanning-tree vlan 532 cost 3',
...           '!',
...           'interface FastEthernet0/2',
...           ' switchport access vlan 300',
...           ' spanning-tree portfast',
...           '!',
...           'interface FastEthernet0/3',
...           ' duplex full',
...           ' speed 100',
...           ' switchport access vlan 300',
...           ' spanning-tree portfast',
...           '!',
...      ]
>>> p = CiscoConfParse(config)
>>> p.find_parents_w_child('^interface', 'switchport access vlan 300')
['interface FastEthernet0/2', 'interface FastEthernet0/3']
>>>
```

find_parents_wo_child(*parentspec*, *childspec*, *ignore_ws=False*)

Parse through all parents matching parentspec, and return a list of parents that did NOT have children match the childspec. For simplicity, this method only finds oldest_ancestors without immediate children that match.

Args:

- parentspec (str): Text regular expression for the line to be matched; this must match the parent's line
- childspec (str): Text regular expression for the line to be matched; this must match the child's line

Kwargs:

- ignore_ws (bool): boolean that controls whether whitespace is ignored

Returns:

- list. A list of matching parent configuration lines

This example finds all ports that are autonegotiating in the following config...

```
!
interface FastEthernet0/1
 switchport access vlan 532
 spanning-tree vlan 532 cost 3
!
interface FastEthernet0/2
 switchport access vlan 300
 spanning-tree portfast
!
interface FastEthernet0/2
 duplex full
 speed 100
 switchport access vlan 300
 spanning-tree portfast
!
```

The following interfaces should be returned:

```
interface FastEthernet0/1
interface FastEthernet0/2
```

We do this by quering *find_parents_wo_child()*; we set our parent as ^*interface* and set the child as *speedsd+* (a regular-expression which matches the word 'speed' followed by an integer).

```
>>> config = ['!',
...           'interface FastEthernet0/1',
...           ' switchport access vlan 532',
...           ' spanning-tree vlan 532 cost 3',
...           '!',
...           'interface FastEthernet0/2',
...           ' switchport access vlan 300',
...           ' spanning-tree portfast',
...           '!',
...           'interface FastEthernet0/3',
...           ' duplex full',
...           ' speed 100',
...           ' switchport access vlan 300',
...           ' spanning-tree portfast',
...           '!',
...     ]
>>> p = CiscoConfParse(config)
>>> p.find_parents_wo_child('^interface', 'speed\s\d+')
['interface FastEthernet0/1', 'interface FastEthernet0/2']
>>>
```

has_line_with(*linespec*)

insert_after(*linespec*, *insertstr=''*, *exactmatch=False*, *ignore_ws=False*, *atomic=False*)

Find all *IOSCfgLine* objects whose text matches `linespec`, and insert `insertstr` after those line objects

insert_after_child(*parentspec*, *childspec*, *insertstr=''*, *exactmatch=False*, *excludespec=None*, *ignore_ws=False*, *atomic=False*)
Find all *IOSCfgLine* objects whose text matches `linespec` and have a child matching `childspec`, and insert an *IOSCfgLine* object for `insertstr` after those child objects.

insert_before(*linespec*, *insertstr=''*, *exactmatch=False*, *ignore_ws=False*, *atomic=False*)
Find all objects whose text matches linespec, and insert 'insertstr' before those line objects

ioscfg
A list containing all text configuration statements

objs
An alias to the `ConfigObjs` attribute

prepend_line(*linespec*)
Unconditionally insert an *IOSCfgLine* object for `linespec` (a text line) at the top of the configuration

replace_all_children(*parentspec*, *childspec*, *replacestr*, *excludespec=None*, *exactmatch=False*, *atomic=False*)
Replace lines matching *childspec* within all children (recursive) of lines whilch match *parentspec*

replace_children(*parentspec*, *childspec*, *replacestr*, *excludespec=None*, *exactmatch=False*, *atomic=False*)
Replace lines matching *childspec* within the *parentspec*'s immediate children.

Args:

- parentspec (str): Text IOS configuration line
- childspec (str): Text IOS configuration line, or regular expression
- replacestr (str): Text IOS configuration, which should replace text matching `childspec`.

Kwargs:

- excludespec (str): A regular expression, which indicates `childspec` lines which *must* be skipped. If `excludespec` is None, no lines will be excluded.
- exactmatch (bool): Defaults to False. When set True, this option requires `linespec` match the whole configuration line, instead of a portion of the configuration line.

Returns:

- list. A list of changed *IOSCfgLine* instances.

replace_children() just searches through a parent's child lines and replaces anything matching *childspec* with *replacestr*. This method is one of my favorites for quick and dirty standardization efforts if you *know* the commands are already there (just set inconsistently).

One very common use case is rewriting all vlan access numbers in a configuration. The following example sets *storm-control broadcast level 0.5* on all GigabitEthernet ports.

```
>>> from ciscoconfparse import CiscoConfParse
>>> config = ['!',
...           'interface GigabitEthernet1/1',
...           ' description {I have a broken storm-control config}',
...           ' switchport',
...           ' switchport mode access',
...           ' switchport access vlan 50',
...           ' switchport nonegotiate',
...           ' storm-control broadcast level 0.2',
...           '!'
...     ]
>>> p = CiscoConfParse(config)
```

```
>>> p.replace_children(r'^interface\sGigabit', r'broadcast\slevel\s\S+',
↪'broadcast level 0.5')
[' storm-control broadcast level 0.5']
>>>
```

One thing to remember about the last example, you *cannot* use a regular expression in *replacestr*; just use a normal python string.

replace_lines(*linespec*, *replacestr*, *excludespec=None*, *exactmatch=False*, *atomic=False*)

This method is a text search and replace (Case-sensitive). You can optionally exclude lines from replacement by including a string (or compiled regular expression) in *excludespec*.

Args:

- linespec (str): Text regular expression for the line to be matched
- replacestr (str): Text used to replace strings matching linespec

Kwargs:

- excludespec (str): Text regular expression used to reject lines, which would otherwise be replaced. Default value of `excludespec` is None, which means nothing is excluded
- exactmatch (bool): boolean that controls whether partial matches are valid
- atomic (bool): boolean that controls whether the config is reparsed after replacement (default True)

Returns:

- list. A list of changed configuration lines

This example finds statements with *EXTERNAL_CBWFQ* in following config, and replaces all matching lines (in-place) with *EXTERNAL_QOS*. For the purposes of this example, let's assume that we do *not* want to make changes to any descriptions on the policy.

```
!
policy-map EXTERNAL_CBWFQ
 description implement an EXTERNAL_CBWFQ policy
 class IP_PREC_HIGH
  priority percent 10
  police cir percent 10
    conform-action transmit
    exceed-action drop
 class IP_PREC_MEDIUM
  bandwidth percent 50
  queue-limit 100
 class class-default
  bandwidth percent 40
  queue-limit 100
policy-map SHAPE_HEIR
 class ALL
  shape average 630000
  service-policy EXTERNAL_CBWFQ
!
```

We do this by calling *replace_lines(linespec='EXTERNAL_CBWFQ', replacestr='EXTERNAL_QOS', excludespec='description')...*

```
>>> from ciscoconfparse import CiscoConfParse
>>> config = ['!',
```

```
...             'policy-map EXTERNAL_CBWFQ',
...             ' description implement an EXTERNAL_CBWFQ policy',
...             ' class IP_PREC_HIGH',
...             '  priority percent 10',
...             '  police cir percent 10',
...             '    conform-action transmit',
...             '    exceed-action drop',
...             ' class IP_PREC_MEDIUM',
...             '  bandwidth percent 50',
...             '  queue-limit 100',
...             ' class class-default',
...             '  bandwidth percent 40',
...             '  queue-limit 100',
...             'policy-map SHAPE_HEIR',
...             ' class ALL',
...             '  shape average 630000',
...             '  service-policy EXTERNAL_CBWFQ',
...             '!',
...         ]
>>> p = CiscoConfParse(config)
>>> p.replace_lines('EXTERNAL_CBWFQ', 'EXTERNAL_QOS', 'description')
['policy-map EXTERNAL_QOS', '  service-policy EXTERNAL_QOS']
>>>
```

Now when we call *p.find_blocks('policy-map EXTERNAL_QOS')*, we get the changed configuration, which has the replacements except on the policy-map's description.

```
>>> p.find_blocks('EXTERNAL_QOS')
['policy-map EXTERNAL_QOS', ' description implement an EXTERNAL_CBWFQ policy',
 ↪ ' class IP_PREC_HIGH', ' class IP_PREC_MEDIUM', ' class class-default',
 ↪'policy-map SHAPE_HEIR', ' class ALL', '  shape average 630000', '  service-
 ↪policy EXTERNAL_QOS']
>>>
```

req_cfgspec_all_diff(*cfgspec*, *ignore_ws=False*)

req_cfgspec_all_diff takes a list of required configuration lines, parses through the configuration, and ensures that none of cfgspec's lines are missing from the configuration. req_cfgspec_all_diff returns a list of missing lines from the config.

One example use of this method is when you need to enforce routing protocol standards, or standards against interface configurations.

Example

```
>>> config = [
...     'logging trap debugging',
...     'logging 172.28.26.15',
...     ]
>>> p = CiscoConfParse(config)
>>> required_lines = [
...     "logging 172.28.26.15",
...     "logging 172.16.1.5",
...     ]
>>> diffs = p.req_cfgspec_all_diff(required_lines)
>>> diffs
['logging 172.16.1.5']
>>>
```

req_cfgspec_excl_diff(*linespec*, *uncfgspec*, *cfgspec*)

req_cfgspec_excl_diff accepts a linespec, an unconfig spec, and a list of required configuration elements. Return a list of configuration diffs to make the configuration comply. **All** other config lines matching the linespec that are *not* listed in the cfgspec will be removed with the uncfgspec regex.

Uses for this method include the need to enforce syslog, acl, or aaa standards.

Example

```
>>> config = [
...     'logging trap debugging',
...     'logging 172.28.26.15',
...     ]
>>> p = CiscoConfParse(config)
>>> required_lines = [
...     "logging 172.16.1.5",
...     "logging 1.10.20.30",
...     "logging 192.168.1.1",
...     ]
>>> linespec = "logging\s+\d+\.\d+\.\d+\.\d+"
>>> unconfspec = linespec
>>> diffs = p.req_cfgspec_excl_diff(linespec, unconfspec,
...     required_lines)
>>> diffs
['no logging 172.28.26.15', 'logging 172.16.1.5', 'logging 1.10.20.30',
↪'logging 192.168.1.1']
>>>
```

save_as(*filepath*)

Save a text copy of the configuration at `filepath`; this method uses the OperatingSystem's native line separators (such as `\r\n` in Windows).

sync_diff(*cfgspec*, *linespec*, *uncfgspec=None*, *ignore_order=True*, *remove_lines=True*, *debug=False*)

`sync_diff()` accepts a list of required configuration elements, a linespec, and an unconfig spec. This method return a list of configuration diffs to make the configuration comply with cfgspec.

Args:

- cfgspec (list): A list of required configuration lines
- linespec (str): A regular expression, which filters lines to be diff'd

Kwargs:

- uncfgspec (str): A regular expression, which is used to unconfigure lines. When ciscoconfparse removes a line, it takes the entire portion of the line that matches `uncfgspec`, and prepends "no" to it.
- ignore_order (bool): Indicates whether the configuration should be reordered to minimize the number of diffs. Default: True (usually it's a good idea to leave `ignore_order` True, except for ACL comparisions)
- remove_lines (bool): Indicates whether the lines which are *not* in `cfgspec` should be removed. Default: True. When `remove_lines` is True, all other config lines matching the linespec that are *not* listed in the cfgspec will be removed with the uncfgspec regex.
- debug (bool): Miscellaneous debugging; Default: False

Returns:

- list. A list of string configuration diffs

Uses for this method include the need to enforce syslog, acl, or aaa standards.

Example

```
>>> config = [
...     'logging trap debugging',
...     'logging 172.28.26.15',
...     ]
>>> p = CiscoConfParse(config)
>>> required_lines = [
...     "logging 172.16.1.5",
...     "logging 1.10.20.30",
...     "logging 192.168.1.1",
...     ]
>>> linespec = "logging\s+\d+\.\d+\.\d+\.\d+"
>>> unconfspec = linespec
>>> diffs = p.sync_diff(required_lines, linespec, unconfspec)
>>> diffs
['no logging 172.28.26.15', 'logging 172.16.1.5', 'logging 1.10.20.30',
↪'logging 192.168.1.1']
>>>
```

IOSConfigList Object

class ciscoconfparse.**IOSConfigList**(*data=None*, *comment_delimiter='!'*, *debug=False*, *factory=False*, *ignore_blank_lines=True*, *syntax='ios'*, *CiscoConfParse=None*)

A custom list to hold *IOSCfgLine* objects. Most people will never need to use this class directly.

Initialize the class.

Kwargs:

- data (list): A list of parsed *IOSCfgLine* objects
- comment (str): A comment delimiter. This should only be changed when parsing non-Cisco IOS configurations, which do not use a ! as the comment delimiter. `comment` defaults to '!'
- debug (bool): `debug` defaults to False, and should be kept that way unless you're working on a very tricky config parsing problem. Debug output is not particularly friendly
- ignore_blank_lines (bool): `ignore_blank_lines` defaults to True; when this is set True, ciscoconfparse ignores blank configuration lines. You might want to set `ignore_blank_lines` to False if you intentionally use blank lines in your configuration (ref: Github Issue #2).

Returns:

- An instance of an *IOSConfigList* object.

config_heirarchy()

Walk this configuration and return the following tuple at each parent 'level':

(list_of_parent_sibling_objs, list_of_nonparent_sibling_objs)

count(*value*) → integer – return number of occurrences of value

extend(*values*)

S.extend(iterable) – extend sequence by appending elements from the iterable

index(*value*) → integer – return first index of value.

Raises ValueError if the value is not present.

pop([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse *IN PLACE*

IOSCfgLine Object

class models_cisco.**IOSCfgLine**(**args*, ***kwargs*)
An object for a parsed IOS-style configuration line. *IOSCfgLine* objects contain references to other parent and child *IOSCfgLine* objects.

Note: Originally, *IOSCfgLine* objects were only intended for advanced ciscoconfparse users. As of ciscoconfparse version 0.9.10, *all users* are strongly encouraged to prefer the methods directly on *IOSCfgLine* objects. Ultimately, if you write scripts which call methods on *IOSCfgLine* objects, your scripts will be much more efficient than if you stick strictly to the classic *CiscoConfParse* methods.

Args:

- text (str): A string containing a text copy of the IOS configuration line. *CiscoConfParse* will automatically identify the parent and children (if any) when it parses the configuration.
- comment_delimiter (str): A string which is considered a comment for the configuration format. Since this is for Cisco IOS-style configurations, it defaults to `!`.

Attributes:

- text (str): A string containing the parsed IOS configuration statement
- linenum (int): The line number of this configuration statement in the original config; default is -1 when first initialized.
- parent (*IOSCfgLine()*): The parent of this object; defaults to `self`.
- children (list): A list of `IOSCfgLine()` objects which are children of this object.
- child_indent (int): An integer with the indentation of this object's children
- indent (int): An integer with the indentation of this object's `text` oldest_ancestor (bool): A boolean indicating whether this is the oldest ancestor in a family
- is_comment (bool): A boolean indicating whether this is a comment

Returns:

- An instance of *IOSCfgLine*.

Accept an IOS line number and initialize family relationship attributes

add_child(*childobj*)
Add references to childobj, on this object

add_parent(*parentobj*)
Add a reference to parentobj, on this object

add_uncfgtext(*unconftext*)
unconftext is defined during special method calls. Do not assume it is automatically populated.

append_to_family(*insertstr*, *indent=-1*, *auto_indent_width=1*, *auto_indent=False*)
Append an *IOSCfgLine* object with `insertstr` as a child at the bottom of the current configuration family.

Args:

- insertstr (str): A string which contains the text configuration to be apppended.
- indent (int): The amount of indentation to use for the child line; by default, the number of left spaces provided with `insertstr` are respected. However, you can manually set the indent level when `indent`>0. This option will be ignored, if `auto_indent` is True.
- auto_indent_width (int): Amount of whitespace to automatically indent
- auto_indent (bool): Automatically indent the child to `auto_indent_width`

Returns:

- str. The text matched by the regular expression group; if there is no match, None is returned.

This example illustrates how you can use `append_to_family()` to add a `carrier-delay` to each interface.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial1/1',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> for obj in parse.find_objects(r'^interface'):
...     obj.append_to_family(' carrier-delay msec 500')
>>>
>>> for line in parse.ioscfg:
...     print line
...
!
interface Serial1/0
 ip address 1.1.1.1 255.255.255.252
 carrier-delay msec 500
!
interface Serial1/1
 ip address 1.1.1.5 255.255.255.252
 carrier-delay msec 500
!
>>>
```

delete(*recurse=True*)
Delete this object. By default, if a parent object is deleted, the child objects are also deleted; this happens because `recurse` defaults True.

delete_children_matching(*linespec*)
Delete any child *IOSCfgLine* objects which match `linespec`.

Args:

- linespec (str): A string or python regular expression, which should be matched.

Returns:

- list. A list of *IOSCfgLine* objects which were deleted.

This example illustrates how you can use `delete_children_matching()` to delete any description on an interface.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' description Some lame description',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial1/1',
...     ' description Another lame description',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> for obj in parse.find_objects(r'^interface'):
...     obj.delete_children_matching(r'description')
>>>
>>> for line in parse.ioscfg:
...     print line
...
!
interface Serial1/0
 ip address 1.1.1.1 255.255.255.252
!
interface Serial1/1
 ip address 1.1.1.5 255.255.255.252
!
>>>
```

geneology
Iterate through to the oldest ancestor of this object, and return a list of all ancestors in the direct line as well as this obj. Cousins or aunts / uncles are *not* returned. Note: children of this object are *not* returned.

geneology_text
Iterate through to the oldest ancestor of this object, and return a list of all ancestors in the direct line as well as this obj. Cousins or aunts / uncles are *not* returned. Note: children of this object are *not* returned.

hash_children
Return a unique hash of all children (if the number of children > 0)

in_portchannel
Return a boolean indicating whether this port is configured in a port-channel

insert_after()

insert_before()

ioscfg
Return a list with this the text of this object, and with all children in the direct line.

is_config_line
Return a boolean for whether this is a config statement; returns False if this object is a blank line, or a comment

is_ethernet_intf

Returns a boolean (True or False) to answer whether this *IOSCfgLine* is an ethernet interface. Any ethernet interface (10M through 10G) is considered an ethernet interface.

Returns:

- bool.

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface FastEthernet1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>> obj = parse.find_objects('^interface\sFast')[0]
>>> obj.is_ethernet_intf
True
>>> obj = parse.find_objects('^interface\sATM')[0]
>>> obj.is_ethernet_intf
False
>>>
```

is_intf

Returns a boolean (True or False) to answer whether this *IOSCfgLine* is an interface; subinterfaces also return True.

Returns:

- bool.

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>> obj = parse.find_objects('^interface\sSerial')[0]
>>> obj.is_intf
```

```
True
>>> obj = parse.find_objects('^interface\sATM')[0]
>>> obj.is_intf
True
>>>
```

is_loopback_intf

Returns a boolean (True or False) to answer whether this *IOSCfgLine* is a loopback interface.

Returns:

- bool.

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface FastEthernet1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Loopback0',
...     ' ip address 1.1.1.5 255.255.255.255',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>> obj = parse.find_objects('^interface\sFast')[0]
>>> obj.is_loopback_intf
False
>>> obj = parse.find_objects('^interface\sLoop')[0]
>>> obj.is_loopback_intf
True
>>>
```

is_portchannel

Return a boolean indicating whether this port is a port-channel intf

is_subintf

Returns a boolean (True or False) to answer whether this *IOSCfgLine* is a subinterface.

Returns:

- bool.

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
```

```
>>> obj = parse.find_objects('^interface\sSerial')[0]
>>> obj.is_subintf
False
>>> obj = parse.find_objects('^interface\sATM')[0]
>>> obj.is_subintf
True
>>>
```

lineage
Iterate through to the oldest ancestor of this object, and return a list of all ancestors / children in the direct line. Cousins or aunts / uncles are *not* returned. Note: all children of this object are returned.

portchannel_number
Return an integer for the port-channel which it's configured in. Return -1 if it's not configured in a port-channel

re_match(*regex*, *group=1*, *default=''*)
Use `regex` to search the *IOSCfgLine* text and return the regular expression group, at the integer index.

Args:

- regex (str): A string or python regular expression, which should be matched. This regular expression should contain parenthesis, which bound a match group.

Kwargs:

- group (int): An integer which specifies the desired regex group to be returned. `group` defaults to 1.
- default (str): The default value to be returned, if there is no match. By default an empty string is returned if there is no match.

Returns:

- str. The text matched by the regular expression group; if there is no match, `default` is returned.

This example illustrates how you can use `re_match()` to store the mask of the interface which owns "1.1.1.5" in a variable called `netmask`.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial1/1',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> for obj in parse.find_objects(r'ip\saddress'):
...     netmask = obj.re_match(r'1\.1\.1\.5\s(\S+)')
>>>
>>> print "The netmask is", netmask
The netmask is 255.255.255.252
>>>
```

re_match_iter_typed(*regex*, *group=1*, *result_type=<type 'str'>*, *default=''*)
Use `regex` to search the children of *IOSCfgLine* text and return the contents of the regular expression group, at the integer `group` index, cast as `result_type`; if there is no match, `default` is returned.

Args:

- regex (str): A string or python compiled regular expression, which should be matched. This regular expression should contain parenthesis, which bound a match group.

Kwargs:

- group (int): An integer which specifies the desired regex group to be returned. `group` defaults to 1.
- result_type (type): A type (typically one of: `str`, `int`, `float`, or `IPv4Obj`). All returned values are cast as `result_type`, which defaults to `str`.
- default (any): The default value to be returned, if there is no match.

Returns:

- `result_type`. The text matched by the regular expression group; if there is no match, `default` is returned. All values are cast as `result_type`.

This example illustrates how you can use *re_match_iter_typed()* to build an `IPv4Obj()` address object for each interface.

```
>>> from ciscoconfparse import CiscoConfParse
>>> from ciscoconfparse.ccp_util import IPv4Obj
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial2/0',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>> INTF_RE = re.compile(r'interface\s\S+')
>>> ADDR_RE = re.compile(r'ip\saddress\s(\S+\s+\S+)')
>>> for obj in parse.find_objects(INTF_RE):
...     print obj.text, obj.re_match_iter_typed(ADDR_RE, result_type=IPv4Obj)
interface Serial1/0 <IPv4Obj 1.1.1.1/30>
interface Serial2/0 <IPv4Obj 1.1.1.5/30>
>>>
```

re_match_typed(*regex*, *group=1*, *result_type=<type 'str'>*, *default=''*)

Use `regex` to search the *IOSCfgLine* text and return the contents of the regular expression group, at the integer `group` index, cast as `result_type`; if there is no match, `default` is returned.

Args:

- regex (str): A string or python regular expression, which should be matched. This regular expression should contain parenthesis, which bound a match group.

Kwargs:

- group (int): An integer which specifies the desired regex group to be returned. `group` defaults to 1.
- result_type (type): A type (typically one of: `str`, `int`, `float`, or `IPv4Obj`). All returned values are cast as `result_type`, which defaults to `str`.
- default (any): The default value to be returned, if there is no match.

Returns:

- `result_type`. The text matched by the regular expression group; if there is no match, `default` is returned. All values are cast as `result_type`.

This example illustrates how you can use *`re_match_typed()`* to build an association between an interface name, and its numerical slot value. The name will be cast as `str()`, and the slot will be cast as `int()`.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial2/0',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> slots = dict()
>>> for obj in parse.find_objects(r'^interface'):
...     name = obj.re_match_typed(regex=r'^interface\s(\S+)',
...         default='UNKNOWN')
...     slot = obj.re_match_typed(regex=r'Serial(\d+)',
...         result_type=int,
...         default=-1)
...     print "Interface {0} is in slot {1}".format(name, slot)
...
Interface Serial1/0 is in slot 1
Interface Serial2/0 is in slot 2
>>>
```

`re_search` (*regex*, *default=''*)
Use `regex` to search this *`IOSCfgLine`*'s text.

Args:

- regex (str): A string or python regular expression, which should be matched.

Kwargs:

- default (str): A value which is returned if `re_search()` doesn't find a match while looking for `regex`.

Returns:

- str. The *`IOSCfgLine`* text which matched. If there is no match, `default` is returned.

`re_search_children` (*regex*)
Use `regex` to search the text contained in the children of this *`IOSCfgLine`*.

Args:

- regex (str): A string or python regular expression, which should be matched.

Returns:

- list. A list of matching *`IOSCfgLine`* objects which matched. If there is no match, an empty `list()` is returned.

`re_sub` (*regex*, *replacergx*, *ignore_rgx=None*)
Replace all strings matching `linespec` with `replacestr` in the *`IOSCfgLine`* object; however, if the *`IOSCfgLine`* text matches `ignore_rgx`, then the text is *not* replaced.

Args:

- linespec (str): A string or python regular expression, which should be matched.
- replacestr (str): A string or python regular expression, which should replace the text matched by `linespec`.

Kwargs:

- ignore_rgx (str): A string or python regular expression; the replacement is skipped if *IOSCfgLine* text matches `ignore_rgx`. `ignore_rgx` defaults to None, which means no lines matching `linespec` are skipped.

Returns:

- str. The new text after replacement

This example illustrates how you can use *re_sub()* to replace `Serial1` with `Serial0` in a configuration...

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial1/1',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> for obj in parse.find_objects('Serial'):
...     print "OLD", obj.text
...     obj.re_sub(r'Serial1', r'Serial0')
...     print "  NEW", obj.text
OLD interface Serial1/0
  NEW interface Serial0/0
OLD interface Serial1/1
  NEW interface Serial0/1
>>>
```

replace (*linespec*, *replacestr*, *ignore_rgx=None*)

Replace all strings matching `linespec` with `replacestr` in the *IOSCfgLine* object; however, if the *IOSCfgLine* text matches `ignore_rgx`, then the text is *not* replaced. The `replace()` method is simply an alias to the `re_sub()` method.

Args:

- linespec (str): A string or python regular expression, which should be matched
- replacestr (str): A string or python regular expression, which should replace the text matched by `linespec`.

Kwargs:

- ignore_rgx (str): A string or python regular expression; the replacement is skipped if *IOSCfgLine* text matches `ignore_rgx`. `ignore_rgx` defaults to None, which means no lines matching `linespec` are skipped.

Returns:

- str. The new text after replacement

This example illustrates how you can use *replace()* to replace `Serial1` with `Serial0` in a configuration...

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial1/1',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> for obj in parse.find_objects('Serial'):
...     print "OLD", obj.text
...     obj.replace(r'Serial1', r'Serial0')
...     print "  NEW", obj.text
OLD interface Serial1/0
  NEW interface Serial0/0
OLD interface Serial1/1
  NEW interface Serial0/1
>>>
```

IOSIntfLine Object

class `models_cisco.`**IOSIntfLine**(**args*, ***kwargs*)

Accept an IOS line number and initialize family relationship attributes

> **Warning:** All *IOSIntfLine* methods are still considered beta-quality, until this notice is removed. The behavior of APIs on this object could change at any time.

abbvs

A python set of valid abbreviations (lowercased) for the interface

access_vlan

Return an integer with the access vlan number. Return 1, if the switchport has no explicit vlan configured; return 0 if the port isn't a switchport

add_child(*childobj*)

Add references to childobj, on this object

add_parent(*parentobj*)

Add a reference to parentobj, on this object

add_uncfgtext(*unconftext*)

unconftext is defined during special method calls. Do not assume it is automatically populated.

append_to_family(*insertstr*, *indent=-1*, *auto_indent_width=1*, *auto_indent=False*)

Append an *IOSCfgLine* object with `insertstr` as a child at the bottom of the current configuration family.

Args:

- insertstr (str): A string which contains the text configuration to be apppended.

- indent (int): The amount of indentation to use for the child line; by default, the number of left spaces provided with `insertstr` are respected. However, you can manually set the indent level when `indent`>0. This option will be ignored, if `auto_indent` is True.
- auto_indent_width (int): Amount of whitespace to automatically indent
- auto_indent (bool): Automatically indent the child to `auto_indent_width`

Returns:

- str. The text matched by the regular expression group; if there is no match, None is returned.

This example illustrates how you can use `append_to_family()` to add a `carrier-delay` to each interface.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial1/1',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> for obj in parse.find_objects(r'^interface'):
...     obj.append_to_family(' carrier-delay msec 500')
>>>
>>> for line in parse.ioscfg:
...     print line
...
!
interface Serial1/0
 ip address 1.1.1.1 255.255.255.252
 carrier-delay msec 500
!
interface Serial1/1
 ip address 1.1.1.5 255.255.255.252
 carrier-delay msec 500
!
>>>
```

delete(*recurse=True*)

Delete this object. By default, if a parent object is deleted, the child objects are also deleted; this happens because `recurse` defaults True.

delete_children_matching(*linespec*)

Delete any child *IOSCfgLine* objects which match `linespec`.

Args:

- linespec (str): A string or python regular expression, which should be matched.

Returns:

- list. A list of *IOSCfgLine* objects which were deleted.

This example illustrates how you can use `delete_children_matching()` to delete any description on an interface.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' description Some lame description',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial1/1',
...     ' description Another lame description',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> for obj in parse.find_objects(r'^interface'):
...     obj.delete_children_matching(r'description')
>>>
>>> for line in parse.ioscfg:
...     print line
...
!
interface Serial1/0
 ip address 1.1.1.1 255.255.255.252
!
interface Serial1/1
 ip address 1.1.1.5 255.255.255.252
!
>>>
```

description
Return the current interface description string.

geneology
Iterate through to the oldest ancestor of this object, and return a list of all ancestors in the direct line as well as this obj. Cousins or aunts / uncles are *not* returned. Note: children of this object are *not* returned.

geneology_text
Iterate through to the oldest ancestor of this object, and return a list of all ancestors in the direct line as well as this obj. Cousins or aunts / uncles are *not* returned. Note: children of this object are *not* returned.

has_manual_carrierdelay
Return a python boolean for whether carrier delay is manually configured on the interface

has_no_ip_proxyarp
Return a boolean for whether no ip proxy-arp is configured on the interface.

Returns:

- bool.

This example illustrates use of the method.

```
>>> from ciscoconfparse.ccp_util import IPv4Obj
>>> from ciscoconfparse import CiscoConfParse
>>> config = [
...     '!',
...     'interface FastEthernet1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     ' no ip proxy-arp',
...     '!',
...     ]
```

```
>>> parse = CiscoConfParse(config, factory=True)
>>> obj = parse.find_objects('^interface\sFast')[0]
>>> obj.has_no_ip_proxyarp
True
>>>
```

hash_children
Return a unique hash of all children (if the number of children > 0)

in_ipv4_subnet (*ipv4network=<IPv4Obj 0.0.0.0/32>*)
Accept an argument for the `IPv4Obj` to be considered, and return a boolean for whether this interface is within the requested `IPv4Obj`.

Kwargs:

- ipv4network (`IPv4Obj`): An object to compare against IP addresses configured on this *IOSIntfLine* object.

Returns:

- bool if there is an ip address, or None if there is no ip address.

This example illustrates use of the method.

```
>>> from ciscoconfparse.ccp_util import IPv4Obj
>>> from ciscoconfparse import CiscoConfParse
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config, factory=True)
>>> obj = parse.find_objects('^interface\sSerial')[0]
>>> obj
<IOSIntfLine # 1 'Serial1/0' info: '1.1.1.1/30'>
>>> obj.in_ipv4_subnet(IPv4Obj('1.1.1.0/24', strict=False))
True
>>> obj.in_ipv4_subnet(IPv4Obj('2.1.1.0/24', strict=False))
False
>>>
```

in_ipv4_subnets (*subnets=None*)
Accept a set or list of ccp_util.IPv4Obj objects, and return a boolean for whether this interface is within the requested subnets.

in_portchannel
Return a boolean indicating whether this port is configured in a port-channel

insert_after()

insert_before()

interface_number

Return a string representing the card, slot, port for this interface. If you call interface_number on GigabitEthernet2/25.100, you'll get this python string: '2/25'. If you call interface_number on GigabitEthernet2/0/25.100 you'll get this python string '2/0/25'. This method strips all subinterface information in the returned value.

Returns:

- string.

> **Warning:** interface_number should silently fail (returning an empty python string) if the interface doesn't parse correctly

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface FastEthernet1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config, factory=True)
>>> obj = parse.find_objects('^interface\sFast')[0]
>>> obj.interface_number
'1/0'
>>> obj = parse.find_objects('^interface\sATM')[-1]
>>> obj.interface_number
'2/0'
>>>
```

ioscfg

Return a list with this the text of this object, and with all children in the direct line.

ipv4_addr

Return a string with the interface's IPv4 address, or '' if there is none

ipv4_addr_object

Return a ccp_util.IPv4Obj object representing the address on this interface; if there is no address, return IPv4Obj('127.0.0.1/32')

ipv4_masklength

Return an integer with the interface's IPv4 mask length, or 0 if there is no IP address on the interace

ipv4_netmask

Return a string with the interface's IPv4 netmask, or '' if there is none

ipv4_network_object

Return an ccp_util.IPv4Obj object representing the subnet on this interface; if there is no address, return ccp_util.IPv4Obj('127.0.0.1/32')

is_abbreviated_as(*val*)
Test whether *val* is a good abbreviation for the interface

is_config_line
Return a boolean for whether this is a config statement; returns False if this object is a blank line, or a comment

is_ethernet_intf
Returns a boolean (True or False) to answer whether this *IOSCfgLine* is an ethernet interface. Any ethernet interface (10M through 10G) is considered an ethernet interface.

Returns:

- bool.

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface FastEthernet1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>> obj = parse.find_objects('^interface\sFast')[0]
>>> obj.is_ethernet_intf
True
>>> obj = parse.find_objects('^interface\sATM')[0]
>>> obj.is_ethernet_intf
False
>>>
```

is_intf
Returns a boolean (True or False) to answer whether this *IOSCfgLine* is an interface; subinterfaces also return True.

Returns:

- bool.

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
```

```
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>> obj = parse.find_objects('^interface\sSerial')[0]
>>> obj.is_intf
True
>>> obj = parse.find_objects('^interface\sATM')[0]
>>> obj.is_intf
True
>>>
```

is_loopback_intf

Returns a boolean (True or False) to answer whether this *IOSCfgLine* is a loopback interface.

Returns:

- bool.

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface FastEthernet1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Loopback0',
...     ' ip address 1.1.1.5 255.255.255.255',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>> obj = parse.find_objects('^interface\sFast')[0]
>>> obj.is_loopback_intf
False
>>> obj = parse.find_objects('^interface\sLoop')[0]
>>> obj.is_loopback_intf
True
>>>
```

is_portchannel

Return a boolean indicating whether this port is a port-channel intf

is_subintf

Returns a boolean (True or False) to answer whether this *IOSCfgLine* is a subinterface.

Returns:

- bool.

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
```

```
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>> obj = parse.find_objects('^interface\sSerial')[0]
>>> obj.is_subintf
False
>>> obj = parse.find_objects('^interface\sATM')[0]
>>> obj.is_subintf
True
>>>
```

lineage
Iterate through to the oldest ancestor of this object, and return a list of all ancestors / children in the direct line. Cousins or aunts / uncles are *not* returned. Note: all children of this object are returned.

manual_arp_timeout
Return an integer with the current interface ARP timeout, if there isn't one set, return 0. If there is no IP address, return -1

manual_carrierdelay
Return the manual carrier delay (in seconds) of the interface as a python float. If there is no explicit carrier delay, return 0.0

manual_clock_rate
Return the clock rate of the interface as a python integer. If there is no explicit clock rate, return 0

manual_holdqueue_in
Return the current hold-queue in depth, if default return 0

manual_holdqueue_out
Return the current hold-queue out depth, if default return 0

manual_mtu
Returns a integer value for the manual MTU configured on an *IOSIntfLine* object. Interfaces without a manual MTU configuration return 0.

Returns:

- integer.

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface FastEthernet1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' mtu 4470',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
```

```
>>> parse = CiscoConfParse(config, factory=True)
>>> obj = parse.find_objects('^interface\sFast')[0]
>>> obj.manual_mtu
0
>>> obj = parse.find_objects('^interface\sATM')[0]
>>> obj.manual_mtu
4470
>>>
```

name

Return the interface name as a string, such as 'GigabitEthernet0/1'

Returns:

- str. The interface name as a string, or '' if the object is not an interface.

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface FastEthernet1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config, factory=True)
>>> obj = parse.find_objects('^interface\sFast')[0]
>>> obj.name
'FastEthernet1/0'
>>> obj = parse.find_objects('^interface\sATM')[0]
>>> obj.name
'ATM2/0'
>>> obj = parse.find_objects('^interface\sATM')[1]
>>> obj.name
'ATM2/0.100'
>>>
```

native_vlan

Return an integer with the native vlan number. Return 1, if the switchport has no explicit native vlan configured; return 0 if the port isn't a switchport

ordinal_list

Return a tuple of numbers representing card, slot, port for this interface. If you call ordinal_list on GigabitEthernet2/25.100, you'll get this python tuple of integers: (2, 25). If you call ordinal_list on GigabitEthernet2/0/25.100 you'll get this python list of integers: (2, 0, 25). This method strips all subinterface information in the returned value.

Returns:

- tuple. A tuple of port numbers as integers.

Warning: ordinal_list should silently fail (returning an empty python list) if the interface doesn't parse correctly

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface FastEthernet1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config, factory=True)
>>> obj = parse.find_objects('^interface\sFast')[0]
>>> obj.ordinal_list
(1, 0)
>>> obj = parse.find_objects('^interface\sATM')[0]
>>> obj.ordinal_list
(2, 0)
>>>
```

port

Return the interface's port number

Returns:

- int. The interface number.

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface FastEthernet1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config, factory=True)
>>> obj = parse.find_objects('^interface\sFast')[0]
>>> obj.port
0
>>> obj = parse.find_objects('^interface\sATM')[0]
>>> obj.port
0
```

```
>>>
```

port_type

Return Loopback, ATM, GigabitEthernet, Virtual-Template, etc...

Returns:

- str. The port type.

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface FastEthernet1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config, factory=True)
>>> obj = parse.find_objects('^interface\sFast')[0]
>>> obj.port_type
'FastEthernet'
>>> obj = parse.find_objects('^interface\sATM')[0]
>>> obj.port_type
'ATM'
>>>
```

portchannel_number

Return an integer for the port-channel which it's configured in. Return -1 if it's not configured in a port-channel

re_match(*regex*, *group=1*, *default=''*)

Use `regex` to search the *IOSCfgLine* text and return the regular expression group, at the integer index.

Args:

- regex (str): A string or python regular expression, which should be matched. This regular expression should contain parenthesis, which bound a match group.

Kwargs:

- group (int): An integer which specifies the desired regex group to be returned. `group` defaults to 1.
- default (str): The default value to be returned, if there is no match. By default an empty string is returned if there is no match.

Returns:

- str. The text matched by the regular expression group; if there is no match, `default` is returned.

This example illustrates how you can use `re_match()` to store the mask of the interface which owns "1.1.1.5" in a variable called `netmask`.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial1/1',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> for obj in parse.find_objects(r'ip\saddress'):
...     netmask = obj.re_match(r'1\.1\.1\.5\s(\S+)')
>>>
>>> print "The netmask is", netmask
The netmask is 255.255.255.252
>>>
```

re_match_iter_typed(*regex*, *group=1*, *result_type=<type 'str'>*, *default=''*)

Use `regex` to search the children of *IOSCfgLine* text and return the contents of the regular expression group, at the integer `group` index, cast as `result_type`; if there is no match, `default` is returned.

Args:

- regex (str): A string or python compiled regular expression, which should be matched. This regular expression should contain parenthesis, which bound a match group.

Kwargs:

- group (int): An integer which specifies the desired regex group to be returned. `group` defaults to 1.
- result_type (type): A type (typically one of: `str`, `int`, `float`, or `IPv4Obj`). All returned values are cast as `result_type`, which defaults to `str`.
- default (any): The default value to be returned, if there is no match.

Returns:

- `result_type`. The text matched by the regular expression group; if there is no match, `default` is returned. All values are cast as `result_type`.

This example illustrates how you can use *re_match_iter_typed()* to build an `IPv4Obj()` address object for each interface.

```
>>> from ciscoconfparse import CiscoConfParse
>>> from ciscoconfparse.ccp_util import IPv4Obj
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial2/0',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>> INTF_RE = re.compile(r'interface\s\S+')
>>> ADDR_RE = re.compile(r'ip\saddress\s(\S+\s+\S+)')
>>> for obj in parse.find_objects(INTF_RE):
...     print obj.text, obj.re_match_iter_typed(ADDR_RE, result_type=IPv4Obj)
```

```
interface Serial1/0 <IPv4Obj 1.1.1.1/30>
interface Serial2/0 <IPv4Obj 1.1.1.5/30>
>>>
```

re_match_typed(*regex*, *group=1*, *result_type=<type 'str'>*, *default=''*)

Use `regex` to search the *IOSCfgLine* text and return the contents of the regular expression group, at the integer `group` index, cast as `result_type`; if there is no match, `default` is returned.

Args:

- regex (str): A string or python regular expression, which should be matched. This regular expression should contain parenthesis, which bound a match group.

Kwargs:

- group (int): An integer which specifies the desired regex group to be returned. `group` defaults to 1.
- result_type (type): A type (typically one of: `str`, `int`, `float`, or `IPv4Obj`). All returned values are cast as `result_type`, which defaults to `str`.
- default (any): The default value to be returned, if there is no match.

Returns:

- `result_type`. The text matched by the regular expression group; if there is no match, `default` is returned. All values are cast as `result_type`.

This example illustrates how you can use *re_match_typed()* to build an association between an interface name, and its numerical slot value. The name will be cast as `str()`, and the slot will be cast as `int()`.

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial2/0',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> slots = dict()
>>> for obj in parse.find_objects(r'^interface'):
...     name = obj.re_match_typed(regex=r'^interface\s(\S+)',
...         default='UNKNOWN')
...     slot = obj.re_match_typed(regex=r'Serial(\d+)',
...         result_type=int,
...         default=-1)
...     print "Interface {0} is in slot {1}".format(name, slot)
...
Interface Serial1/0 is in slot 1
Interface Serial2/0 is in slot 2
>>>
```

re_search(*regex*, *default=''*)

Use `regex` to search this *IOSCfgLine*'s text.

Args:

- regex (str): A string or python regular expression, which should be matched.

Kwargs:

- default (str): A value which is returned if `re_search()` doesn't find a match while looking for `regex`.

Returns:

- str. The *IOSCfgLine* text which matched. If there is no match, `default` is returned.

re_search_children(*regex*)

Use `regex` to search the text contained in the children of this *IOSCfgLine*.

Args:

- regex (str): A string or python regular expression, which should be matched.

Returns:

- list. A list of matching *IOSCfgLine* objects which matched. If there is no match, an empty `list()` is returned.

re_sub(*regex*, *replacergx*, *ignore_rgx=None*)

Replace all strings matching `linespec` with `replacestr` in the *IOSCfgLine* object; however, if the *IOSCfgLine* text matches `ignore_rgx`, then the text is *not* replaced.

Args:

- linespec (str): A string or python regular expression, which should be matched.
- replacestr (str): A string or python regular expression, which should replace the text matched by `linespec`.

Kwargs:

- ignore_rgx (str): A string or python regular expression; the replacement is skipped if *IOSCfgLine* text matches `ignore_rgx`. `ignore_rgx` defaults to None, which means no lines matching `linespec` are skipped.

Returns:

- str. The new text after replacement

This example illustrates how you can use *re_sub()* to replace `Serial1` with `Serial0` in a configuration...

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial1/1',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> for obj in parse.find_objects('Serial'):
...     print "OLD", obj.text
...     obj.re_sub(r'Serial1', r'Serial0')
...     print "  NEW", obj.text
OLD interface Serial1/0
  NEW interface Serial0/0
```

```
OLD interface Serial1/1
  NEW interface Serial0/1
>>>
```

replace(*linespec*, *replacestr*, *ignore_rgx=None*)

Replace all strings matching `linespec` with `replacestr` in the *IOSCfgLine* object; however, if the *IOSCfgLine* text matches `ignore_rgx`, then the text is *not* replaced. The `replace()` method is simply an alias to the `re_sub()` method.

Args:

- linespec (str): A string or python regular expression, which should be matched
- replacestr (str): A string or python regular expression, which should replace the text matched by `linespec`.

Kwargs:

- ignore_rgx (str): A string or python regular expression; the replacement is skipped if *IOSCfgLine* text matches `ignore_rgx`. `ignore_rgx` defaults to None, which means no lines matching `linespec` are skipped.

Returns:

- str. The new text after replacement

This example illustrates how you can use *replace()* to replace `Serial1` with `Serial0` in a configuration...

```
>>> config = [
...     '!',
...     'interface Serial1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface Serial1/1',
...     ' ip address 1.1.1.5 255.255.255.252',
...     '!',
...     ]
>>> parse = CiscoConfParse(config)
>>>
>>> for obj in parse.find_objects('Serial'):
...     print "OLD", obj.text
...     obj.replace(r'Serial1', r'Serial0')
...     print "  NEW", obj.text
OLD interface Serial1/0
  NEW interface Serial0/0
OLD interface Serial1/1
  NEW interface Serial0/1
>>>
```

subinterface_number

Return a string representing the card, slot, port for this interface or subinterface. If you call subinterface_number on GigabitEthernet2/25.100, you'll get this python string: '2/25.100'. If you call interface_number on GigabitEthernet2/0/25 you'll get this python string '2/0/25'. This method strips all subinterface information in the returned value.

Returns:

- string.

Warning: subinterface_number should silently fail (returning an empty python string) if the interface doesn't parse correctly

This example illustrates use of the method.

```
>>> config = [
...     '!',
...     'interface FastEthernet1/0',
...     ' ip address 1.1.1.1 255.255.255.252',
...     '!',
...     'interface ATM2/0',
...     ' no ip address',
...     '!',
...     'interface ATM2/0.100 point-to-point',
...     ' ip address 1.1.1.5 255.255.255.252',
...     ' pvc 0/100',
...     '  vbr-nrt 704 704',
...     '!',
...     ]
>>> parse = CiscoConfParse(config, factory=True)
>>> obj = parse.find_objects('^interface\sFast')[0]
>>> obj.subinterface_number
'1/0'
>>> obj = parse.find_objects('^interface\sATM')[-1]
>>> obj.subinterface_number
'2/0.100'
>>>
```

trunk_vlans_allowed

Return a CiscoRange() with the list of allowed vlan numbers. Return 0 if the port isn't a switchport

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

H

I

L

M

N

O

P

R

S

T