pointfree documentation

Release 1.1.1

Mark Shroyer

April 20, 2013

CONTENTS

Author Mark Shroyer

Email code@markshroyer.com

Release 1.1.1

Date April 20, 2013

`pointfree` is a module providing Pythonic pointfree-style programming. This is its documentation.

CHAPTER

ONE

OVERVIEW

1.1 Introduction

`pointfree` is a small module that makes certain functional programming constructs more convenient to use in Python.

Specifically, it provides:

- A decorator to enable *automatic* partial application of functions and methods.
- Notations for function composition through operator overloading.
- Helper functions to make composing generators more elegant.

The objective is to support the pointfree programming style in a lightweight and easy to use manner – and in particular, to serve as a nice syntax for the kind of generator pipelines described in David Beazley's PyCon 2008 presentation, "Generator Tricks for Systems Programmers".

1.2 Examples

The `pointfree` module is about using function composition notation in conjunction with automatic partial application. Both of these features are achieved by wrapping functions in the `pointfree` class (which can also be applied as a decorator).

Several "pre-wrapped" helper functions are provided by the module. For instance, if you wanted to define a function that returns the sum of squares of the lengths of the strings in a list, you could do so by combining the helpers `pfmap()` and `pfreduce()`:

```
>>> from pointfree import *
>>> from operator import add

>>> fn = pfmap(len) >> pfmap(lambda n: n**2) >> pfreduce(add, initial=0)
>>> fn(["foo", "barr", "bazzz"])
50
```

Aside from the built-in helpers, you can define your own composable functions by applying `pointfree` as a decorator. Building upon an example from Beazley's presentation, suppose you have defined the following functions for operating on lines of text:

```
>>> import re

>>> @pointfree
... def gen_grep(pat, lines):
```

```
...     patc = re.compile(pat)
...     for line in lines:
...         if patc.search(line):
...             yield line

>>> @pointfree
... def gen_repeat(times, lines):
...     for line in lines:
...         for n in range(times):
...             yield line

>>> @pointfree
... def gen_upcase(lines):
...     for line in lines:
...         yield line.upper()
```

And you have some text too:

```
>>> bad_poetry = \
... """roses are red
... violets are blue
... I like generators
... and this isn't a poem
... um let's see...
... oh yeah and daffodils are flowers too""".split("\n")
```

Now say you want to find just the lines of your text that contain the name of a flower and print them, twice, in upper case. (A common problem, I'm sure.) The given functions can be combined to do so as follows, using `pointfree's` automatic partial application and its function composition operators:

```
>>> f = gen_grep(r'(roses|violets|daffodils)') \
...     >> gen_upcase \
...     >> gen_repeat(2) \
...     >> pfprint_all

>>> f(bad_poetry)
ROSES ARE RED
ROSES ARE RED
VIOLETS ARE BLUE
VIOLETS ARE BLUE
OH YEAH AND DAFFODILS ARE FLOWERS TOO
OH YEAH AND DAFFODILS ARE FLOWERS TOO
```

In addition to the >> operator for "forward" composition (borrowed from F#), functions can also be composed with the * operator, which is intended to be remniscent of the circle operator "" from algebra, or the corresponding dot operator in Haskell:

```
>>> @pointfree
... def f(x):
...     return x**2

>>> @pointfree
... def g(x):
...     return x+1

>>> h = f * g
>>> h(2)
9
```

Of course you don't have to define your methods using decorator notation in order to use `pointfree`; you can directly instantiate the class from an existing function or method:

```
>>> (pf(lambda x: x*2) * pf(lambda x: x+1))(3)
8
```

(`pf` is provided as a shorthand alias for the `pointfree` class.)

If you want automatic partial application but not the composition operators, use the module's `partial` decorator instead:

```
>>> @partial
... def add_three(a, b, c):
...     return a + b + c

>>> add_three(1)(2)(3)
6
```

The module's partial application support has some subtle intentional differences from normal Python function application rules. Please see the *module reference* for details.

1.3 Getting the module

Full documentation is available on the web at:

http://markshroyer.com/docs/pointfree/latest/

The easiest way to install the latest release on your machine is to get it from PyPI using pip:

```
$ pip install pointfree
```

or easy_install:

```
$ easy_install pointfree
```

Or you can download the module manually and perform the standard distutils incantations:

```
$ tar xzf pointfree-*.tar.gz
$ cd pointfree-*
$ python setup.py install
```

The module's development repository is hosted on Github:

https://github.com/markshroyer/pointfree

and the very latest development version can also be installed using pip:

```
$ pip install git+git://github.com/markshroyer/pointfree.git
```

`pointfree` is compatible with the following Python implementations:

- CPython 2.6, 2.7, 3.0, 3.1, 3.2, and 3.3
- PyPy 1.9.0
- IronPython 2.7.1

Python 3 is fully supported, including PEP 3102 keyword-only arguments.

CHAPTER

TWO

MODULE REFERENCE

Pythonic pointfree programming.

- Full documentation: http://pointfree.readthedocs.org/en/latest/
- Project page: https://github.com/markshroyer/pointfree

2.1 Copyright notice

2.2 Usage

The general use case is to wrap functions in the `pointfree` wrapper / decorator class, granting them both automatic partial application support and a pair of function composition operators:

```
>>> from pointfree import *

>>> @pointfree
... def pfadd(a, b):
...     return a + b

>>> @pointfree
... def pfexp(n, exp):
...     return n ** exp

>>> fn = pfexp(exp=2) * pfadd(1)
>>> fn(3)
16
```

`pointfree.pointfree` inherits from the `pointfree.partial` class (not to be confused with `functools.partial()`), which provides automatic partial application but not the function composition operators.

See `partial's` documentation for details of the partial application semantics, and `pointfree's` documentation for information about the function composition operators.

The module also includes a number of pre-defined helper functions which can be combined for various purposes:

```
>>> fn = pfmap(lambda x: x**3) >> pfprint_all

>>> fn(range(4))
0
1
8
27
```

Refer to the section *Composable helper functions* for information about the helpers provided by this module.

2.3 Wrapper classes

class `pointfree.`**`partial`**(*func*, **pargs*, ***kargs*)

Wraps a regular Python function or method into a callable object supporting automatic partial application.

Parameters

- **func** – Function or method to wrap
- **pargs** – Optional, positional arguments for the wrapped function
- **kargs** – Optional, keyword arguments for the wrapped function

Example:

```
>>> @partial
... def foo(a,b,c):
...     return a + b + c
>>> foo(1,2,3)
6
>>> foo(1)(2)(3)
6
```

Generally speaking, the evaluation strategy with regard to automatic partial application is to apply all given arguments to the underlying function as soon as possible.

When a `partial` instance is called, the positional and keyword arguments supplied are combined with the instance's own cache of arguments for the wrapped function (which is empty to begin with, for instances directly wrapping – or applied as decorators to – pure Python functions or methods). If the combined set of arguments is sufficient to invoke the wrapped function, then the function is called and its result returned. If the combined arguments are *not* sufficient, then a new copy of the wrapper is returned instead, with the new combined argument set in its cache.

Calling a `partial` object never changes its state; instances are immutable for practical purposes, so they can be called and reused indefinitely:

```
>>> p = q = foo(1,2)
>>> p(3)
6
>>> q(4)  # Using the same instance twice
7
```

Arguments with default values do not need to be explicitly specified in order for evaluation to occur. In the following example, `foo2` can be evaluated as soon as we have specified the arguments `a` and `b`:

```
>>> @partial
... def foo2(a, b, c=3):
...     return a + b + c

>>> foo2(1,2)
6
>>> foo2(1)(2)
6
```

However, if extra arguments are supplied prior to evaluation, and if the underlying function is capable of accepting those arguments, then those will be passed to the function as well. If we call `foo2` as follows, the third argument will be passed to the wrapped function as `c`, overriding its default value:

```
>>> foo2(1,2,5)
8
>>> foo2(3)(4,5)
12
```

This works similarly with functions that accept variable positional argument lists:

```
>>> @partial
... def foo3(a, *args):
...     return a + sum(args)

>>> foo3(1)
1
>>> foo3(1,2)
3
>>> foo3(1,2,3)
6
```

Or variable keyword argument lists:

```
>>> @partial
... def foo4(a, **kargs):
...     kargs.update({'a': a})
...     return kargs

>>> result = foo4(3, b=4, c=5)
>>> for key in sorted(result.keys()):
...     print("%s: %s" % (key, result[key]))
a: 3
b: 4
c: 5
```

But if you try to supply an argument that the function cannot accept, a `TypeError` will be raised as soon as you attempt to do so – the wrapper doesn't wait until the underlying function is called before raising the exception (unlike with `functools.partial()`):

```
>>> @partial
... def foo5(a, b, c):
...     return a + b + c

>>> foo5(d=7)
Traceback (most recent call last):
    ...
TypeError: foo5() got an unexpected keyword argument 'd'
```

There are some sutble differences between how automatic partial application works in this module and the semantics of regular Python function application (or, again, of `functools.partial()`). First, keyword

arguments to partially applied functions can override an argument specified in a previous call:

```
>>> @partial
... def foo6(a, b, c):
...     return (a, b, c)

>>> foo6(1)(b=2)(b=3)(4)  # overriding b given as keyword
(1, 3, 4)
>>> foo6(1,2)(b=3)(4)  # overriding b given positionally
(1, 3, 4)
```

Also, the wrapper somewhat blurs the line between positional and keyword arguments for the sake of flexibilty. If an argument is specified with a keyword and then "reached" by a positional argument in a subsequent call, the remaining positional argument values "wrap around" the argument previously specified as a keyword.

This second difference is best illustrated by example. Again using the function `foo6` from above, if we specify `b` as a keyword argument:

```
>>> p = foo6(b=2)
```

and then apply two positional arguments to the resulting `partial` instance, those arguments will be used to specify `a` and `c`, skipping over `b` because it has already been specified:

```
>>> p(1,3)
(1, 2, 3)
```

This approach was chosen because it allows us to compose partial applications of functions where a previous argument has been specified as a keyword argument.

As well as functions, `partial` can be applied to methods, including class and static methods:

```
>>> class Foo7(object):
...     m = 2
...
...     def __init__(self, n):
...         self.n = n
...
...     @partial
...     def bar_inst(self, a, b, c):
...         return self.m + self.n + a + b + c
...
...     @partial
...     @classmethod
...     def bar_class(klass, a, b, c):
...         return klass.m + a + b + c
...
...     @partial
...     @staticmethod
...     def bar_static(a, b, c):
...         return a + b + c

>>> f = Foo7(3)
>>> f.bar_inst(4)(5)(6)
20
>>> f.bar_class(3)(4)(5)
14
>>> f.bar_static(2)(3)(4)
9
```

The wrapper can also be instantiated from another `partial` instance:

```
>>> def foo8(a, b, c, *args):
...     return a + b + c + sum(args)

>>> p = partial(foo8, 1)
>>> q = partial(p, 2)
>>> q(3)
6
```

Or even from a `functools.partial()` instance:

```
>>> p = functools.partial(foo8, 1)
>>> q = partial(p)
>>> q(2)(3)
6
```

However, it cannot currently wrap a Python builtin function (or a `functools.partial()` instance which wraps a builtin function), as Python does not currently provide sufficient reflection for its builtins.

While you will probably apply `partial` as a decorator when defining your own functions, you can also wrap existing functions by instantiating the class directly:

```
>>> partial(foo8)(1)(2)(3)
6
```

Or like with `functools.partial()`, you can specify arguments for the wrapped function when you instantiate a wrapper:

```
>>> p = partial(foo8, 1)
>>> p(2)(3)
6
```

But unlike calling an existing wrapper instance, the wrapped function will not be invoked during instantiation even if enough arguments are supplied in order to do so; invocation does not occur until the `partial` instance is called at least once, even with an empty argument list:

```
>>> p = partial(foo8, 1, 2, 3)
>>> type(p)
<class 'pointfree.partial'>
>>> p()
6
>>> p(4)
10
```

class `pointfree.`**`pointfree`**(*func*, **pargs*, ***kargs*)

Wraps a regular Python function or method into a callable object supporting the >> and * function composition operators, as well as automatic partial application inherited from `partial`.

Parameters

- **func** – Function or method to wrap
- **pargs** – Optional, positional arguments for the wrapped function
- **kargs** – Optional, keyword arguments for the wrapped function

This class inherits its partial application behavior from `partial`; refer to its documentation for details.

On top of automatic partial application, the `pointfree` wrapper adds two function composition operators, >> and *, for "forward" and "reverse" function composition respectively. For example, given the following wrapped functions:

```
>>> @pointfree
... def pfadd(a, b):
...     return a + b

>>> @pointfree
... def pfmul(a, b):
...     return a * b
```

The following forward composition defines the function `f()` as one which takes a given number, adds one to it, and then multiplies the result of the addition by two:

```
>>> f = pfadd(1) >> pfmul(2)
>>> f(1)
4
```

Reverse composition simply works in the opposite direction. In this example, `g()` takes a number, multiplies it by three, and then adds four:

```
>>> g = pfadd(4) * pfmul(3)
>>> g(5)
19
```

The alias `pf` is provided for `pointfree` to conserve electrons when wrapping functions inline:

```
>>> def add(a, b):
...     return a + b

>>> def mul(a, b):
...     return a * b

>>> f = pf(add, 1) >> pf(mul, 2)
>>> f(2)
6
```

When using `pointfree` as a decorator on class or static methods, you must ensure that it is the "topmost" decorator, so that the resulting object is a `pointfree` instance in order for the composition operators to work.

2.4 Composable helper functions

`pointfree.`**`pfmap`**(*func*, *iterable*)

A pointfree map function: Returns an iterator over the results of applying a function of one argument to the items of a given iterable. The function is provided "lazily" to the given iterable; each function application is performed on the fly as it is requested.

Parameters

- **func** – A function of one argument to apply to each item
- **iterable** – An iterator yielding input for the function

Return type Iterator of function application results

Example:

```
>>> f = pfmap(lambda x: x+1) \
...     >> pfmap(lambda x: x*2) \
...     >> pfcollect
```

```
>>> f(range(5))
[2, 4, 6, 8, 10]
```

pointfree.**pfreduce**(*func*, *iterable*[, *initial=None*])

A pointfree reduce / left fold function: Applies a function of two arguments cumulatively to the items supplied by the given iterable, so as to reduce the iterable to a single value. If an initial value is supplied, it is placed before the items from the iterable in the calculation, and serves as the default when the iterable is empty.

Parameters

- **func** – A function of two arguments
- **iterable** – An iterable yielding input for the function
- **initial** – An optional initial input for the function

Return type Single value

Example:

```
>>> from operator import add

>>> sum_of_squares = pfreduce(add, initial=0) * pfmap(lambda n: n**2)
>>> sum_of_squares([3, 4, 5, 6])
86
```

pointfree.**pffilter**(*param*, *iterable*)

Pointfree filter function.

Example:

```
>>> f = pffilter(lambda n: n % 2 == 0) \
...     >> pfcollect

>>> f(range(5))
[0, 2, 4]
```

pointfree.**pfcollect**(*iterable*[, *n=None*])

Collects and returns a list of values from the given iterable. If the n parameter is not specified, collects all values from the iterable.

Parameters

- **iterable** – An iterable yielding values for the list
- **n** – An optional maximum number of items to collect

Return type List of values from the iterable

Example:

```
>>> @pointfree
... def fibonaccis():
...     a, b = 0, 1
...     while True:
...         a, b = b, a+b
...         yield a

>>> (pfcollect(n=10) * fibonaccis)()
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

pointfree.**pfprint**(*item*[, *end='\n', file=sys.stdout*])

Prints an item.

Parameters

- **item** – The item to print
- **end** – String to append to the end of printed output
- **file** – File to which output is printed

Return type None

Example:

```
>>> from operator import add

>>> fn = pfreduce(add, initial=0) >> pfprint
>>> fn([1, 2, 3, 4])
10
```

pointfree.**pfprint_all**(*iterable*[, *end='\n'*, *file=sys.stdout*])

Prints each item from an iterable.

Parameters

- **iterable** – An iterable yielding values to print
- **end** – String to append to the end of printed output
- **file** – File to which output is printed

Return type None

Example:

```
>>> @pointfree
... def prefix_all(prefix, iterable):
...     for item in iterable:
...         yield "%s%s" % (prefix, item)

>>> fn = prefix_all("An item: ") >> pfprint_all

>>> fn(["foo", "bar", "baz"])
An item: foo
An item: bar
An item: baz
```

pointfree.**pfignore_all**(*iterable*)

Consumes all the items from an iterable, discarding their output. This may be useful if evaluating the iterable produces some desirable side-effect, but you have no need to collect its output.

Parameters **iterable** – An iterable

Return type None

Example:

```
>>> result = []

>>> @pointfree
... def append_all(collector, iterable):
...     for item in iterable:
...         collector.append(item)
...         yield item

>>> @pointfree
```

```
... def square_all(iterable):
...     for item in iterable:
...         yield item**2

>>> fn = square_all \
...      >> append_all(result) \
...      >> pfignore_all
>>> fn([1, 2, 3, 4])
>>> result
[1, 4, 9, 16]
```

CHAPTER

THREE

FAQ

- **Q. Python already includes a partial application class in the standard library's functools module; why not just use that?**

  There are two major reasons that I felt the need to write a new implementation of partial function application for this module.

  First, use of the function composition operators provided by the `pointfree` decorator requires cooperation between the partial application mechanism and the implementation of overloaded operators; the result of a partial application must be an object which defines the necessary operators, so at the very least I would need to wrap `functools.partial()` anyway. (And that in itself would not be easy, because `functools.partial()` does not provide a way to test whether enough arguments have been provided to call the underlying function.)

  The second reason is a subjective matter of taste. The standard library's `partial()` requires explicit creation of a new object every time you wish to perform partial application and then a separate call in order to actually invoke the underlying function, and this is more verbose and (in my opinion) less elegant than I would like. For a contrived example:

```
>>> from functools import partial

>>> def add_thrice(a, b, c):
...     return a + b + c

>>> plusone = partial(add_thrice, 1)
>>> plusone(2, 3)
6
>>> plusthree = partial(plusone, 2)
>>> plusthree(3)
6
```

  In contrast, pointfree's `partial` decorator lets you perform partial application with the same syntax as "full" application:

```
>>> from pointfree import partial

>>> @partial
... def add_thrice(a, b, c):
...     return a + b + c

>>> plusone = add_thrice(1)
>>> plusone(2, 3)
6
>>> plusthree = plusone(2)
>>> plusthree(3)
6
```

There are also several minor ways in which `functools.partial()` is not ideal for supporting the pointfree style. If you have a function of two arguments and you specify the first as a keyword argument, you cannot then specify the second positionally in a subsequent application; this would prevent such a partially-applied function from being composed with other functions:

```
>>> from functools import partial

>>> def add(a, b):
...     return a + b

>>> p = partial(add, a=1)
>>> p(2)
Traceback (most recent call last):
    ...
TypeError: add() got multiple values for keyword argument 'a'
```

Whereas you can do this with pointfree, due to its slightly different semantics for positional argument application (which is fully described in the *Module reference*):

```
>>> from pointfree import partial

>>> @partial
... def add(a, b):
...     return a + b

>>> p = add(a=1)
>>> p(2)
3
```

Also, with the standard library's partial class you don't see errors immediately when you apply invalid positional or keyword arguments; the exception is only raised when you later `__call__` the partial object:

```
>>> from functools import partial

>>> def add(a, b):
...     return a + b

>>> p = partial(add, c=3)  # No error is raised yet
>>> q = partial(p, 1)      # Still no error
>>> q(2)                   # Now we get an error!
Traceback (most recent call last):
    ...
TypeError: add() got an unexpected keyword argument 'c'
```

But with pointfree's partial application, the error is raised immediately:

```
>>> from pointfree import partial

>>> @partial
... def add(a, b):
...     return a + b

>>> p = add(c=3)
Traceback (most recent call last):
    ...
TypeError: add() got an unexpected keyword argument 'c'
```

- **Q. Are there any disadvantages to pointfree's partial application style?**

  Because Python does not currently expose built-in functions for introspection, the pure-Python

`pointfree.partial` wrapper does not work with built-in functions.

Also, with the pointfree implementation of partial application you cannot specify optional positional arguments in *multiple* applications, because evaluation will occur automatically as soon as enough arguments have been specified. So, for instance, with `functools.partial()`:

```
>>> from functools import partial

>>> def add_all(*argv):
...     return sum(argv)

>>> f = partial(add_all, 1, 2)
>>> g = partial(f, 3, 4)
>>> g(5)
15
```

Whereas with pointfree, the function would be evaluated as soon as it has been supplied any arguments:

```
>>> from pointfree import partial

>>> partial(add_all)(1, 2) # evaluated immediately
3
```

Despite these limitations, I prefer the brevity of the pointfree implementation (which is of course why I wrote it). Naturally, your mileage may vary.

CHAPTER

FOUR

LICENSE

Object form, made available under the License, as indicated by a
copyright notice that is included in or attached to the work
(an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object
form, that is based on (or derived from) the Work and for which the
editorial revisions, annotations, elaborations, or other modifications
represent, as a whole, an original work of authorship. For the purposes
of this License, Derivative Works shall not include works that remain
separable from, or merely link (or bind by name) to the interfaces of,
the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including
the original version of the Work and any modifications or additions
to that Work or Derivative Works thereof, that is intentionally
submitted to Licensor for inclusion in the Work by the copyright owner
or by an individual or Legal Entity authorized to submit on behalf of
the copyright owner. For the purposes of this definition, "submitted"
means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity
on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

(a) You must give any other recipients of the Work or
Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices
stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works
that You distribute, all copyright, patent, trademark, and
attribution notices from the Source form of the Work,
excluding those notices that do not pertain to any part of
the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its
distribution, then any Derivative Works that You distribute must
include a readable copy of the attribution notices contained
within such NOTICE file, excluding those notices that do not
pertain to any part of the Derivative Works, in at least one
of the following places: within a NOTICE text file distributed
as part of the Derivative Works; within the Source form or
documentation, if provided along with the Derivative Works; or,
within a display generated by the Derivative Works, if and
wherever such third-party notices normally appear. The contents
of the NOTICE file are for informational purposes only and
do not modify the License. You may add Your own attribution
notices within Derivative Works that You distribute, alongside
or as an addendum to the NOTICE text from the Work, provided
that such additional attribution notices cannot be construed
as modifying the License.

You may add Your own copyright statement to Your modifications and
may provide additional or different license terms and conditions
for use, reproduction, or distribution of Your modifications, or
for any such Derivative Works as a whole, provided Your use,
reproduction, and distribution of the Work otherwise complies with
the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
any Contribution intentionally submitted for inclusion in the Work
by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions.
Notwithstanding the above, nothing herein shall supersede or modify
the terms of any separate license agreement you may have executed
with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
names, trademarks, service marks, or product names of the Licensor,
except as required for reasonable and customary use in describing the
origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
agreed to in writing, Licensor provides the Work (and each
Contributor provides its Contributions) on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied, including, without limitation, any warranties or conditions
of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
PARTICULAR PURPOSE. You are solely responsible for determining the
appropriateness of using or redistributing the Work and assume any
risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX