

# The Python:



This is the second ever Python: Rag - a celebratory issue to mark the first month since the ..er last one.

Would anyone like to write one of the 'Monthly Module' articles? - Pick a module, and write something about it - why you like it, why you dislike it, an example of its use, or how it changed your life in strange and unexpected ways.

## News

Two more Python conferences coming up:

SciPy August 18 to 23  
<http://conference.scipy.org/>

New Zealand in November:  
<http://nz.pycon.org/>

```
>>> import datetime
>>> datetime.date.today().strftime("%B %Y")
'August 2009'
```

## Contents

### Today's Tip

Sorting a list of objects.

### Monthly Module

The logging module is looked at.

### Beginners Bits

Lists.

### Albatross Article

Warts and all.

## ATTRIBUTION

[www.pythonrag.org](http://www.pythonrag.org)

The Python: Rag is a monthly newsletter covering any aspect of the Python programming language.

If you wish to contribute - send your text to [editor@pythonrag.org](mailto:editor@pythonrag.org)

The topic should have some relationship to python, the style can be anything, code and images can be included. The level of python knowledge can be beginner to brain bashing.

Your article may not be published if (in our opinion) - it is completely nutty.

Any opinions expressed in these articles are those of the authors only, and not the producers of The Python: Rag.

## Today's Tip

I had a list of objects I needed to sort, depending on the value of an attribute of my objects.

No problem, I've done it before, one uses:

```
mylist.sort([cmp[, key[, reverse]]])
```

So I need to write a comparison function that compares the attribute of two objects, returning -1 if less, 0 if equal and +1 if greater.

But then I heard that the 'cmp' is going in Python 3, and I want my code to be as near Python 3 as possible - so I read up, just what is this key parameter?

It seems it should be a function, which on being given an element, should return a value that the sort function would use to sort with.

Not too bad - the attribute of my objects I want to sort on happens to be an integer, so I just need to write a function which, given the object, returns the attribute.

Hey - I don't even need a function - how about:

```
mylist.sort(key=lambda x: x.myattr)
```

- where myattr is the attribute of the object.

And that's it - a lot easier than writing that compare function.

And on a somewhat related subject:

If you are interested in a 'peculiar' list - read up on the heapq module - this is a list ordered according to a 'heap' algorithm in which the item at index 0 is always the smallest item and using heappush to add items, and heappop to remove items keeps this heap in order.

So for example, look at the code on the right, no matter in what order you push things onto the heap, they are always popped off smallest to largest. Neat uh?

You can take an ordinary list and turn it into a 'heap' list by the 'heapify(x)' command which transforms the list x in-place.

```
import heapq

x = []
heapq.heappush(x, 3)
heapq.heappush(x, 9)
heapq.heappush(x, 2)
a = heapq.heappop(x)
print a
a = heapq.heappop(x)
print a
a = heapq.heappop(x)
print a
```

*This prints 2, 3, 9, smallest to largest regardless of the order they were pushed on to the heap.*

## Monthly Module - logging

Here's an example of how I use the logging module - first I create a module `my_logger.py` which I subsequently import into the other modules of my application.

It holds the `_DEBUG_ON` flag, and the function `set_logger()` which sets up my logging infrastructure:

```
# my_logger.py

"""This is 'my_logger' module, which is imported into all
   the other modules of my application."""

import logging
import logging.handlers
from functools import wraps

# Create a global logger

_vs_logger = None

# Set default DEBUG_ON True - in which case debug messages
# are saved to the log file. Or set it to False - in which
# case only INFO and ERROR messages are saved to the log file

_DEBUG_ON = True

def set_logger():
    "Set up the logger"

    global _vs_logger

    _vs_logger = logging.getLogger("my_logger")

    # Set the logger level
    if _DEBUG_ON:
        _vs_logger.setLevel(logging.DEBUG)
    else:
        _vs_logger.setLevel(logging.INFO)

    # Set the format
    form = logging.Formatter("%(asctime)s - %(levelname)s - %(message)s")

    # Add the log message handler to the logger
    # The location of the logfile is given here as 'logfile.txt'
    # in an actual application I would take a bit of care
    # where this is located

    handler = logging.handlers.RotatingFileHandler("logfile.txt",
                                                    maxBytes=20000,
                                                    backupCount=5)

    handler.setFormatter(form)
    _vs_logger.addHandler(handler)
```

## logging continued ..

The Formatter defines how the log records will be formatted - in this case it creates a date-time stamp, and shows the level of the log - debug, info or error, and then the message itself.

The RotatingFileHandler creates a logfile called logfile.txt, and when this is larger than 20000 bytes, it is renamed logfile.txt.1 and a new logfile.txt is created - these files are 'rotated' up to logfile.txt.5 with older records deleted. The logging module has other handlers available.

my\_logger.py also contains the following functions - which I can then call from my application code whenever I want something to be logged.

info\_log(message) always logs the string given by message.

debug\_log(message) only logs message if the \_DEBUG\_ON flag is set to True - which I would set when developing the program, but then change to False when completed.

exception\_log(message) is used in exception clauses to log the exception.

It's also possible to create decorators to log function and method calls - such a decorator for a method call is shown below.

```
def info_log(message):
    "Log message with level info"
    if _vs_logger:
        _vs_logger.info(str(message))

def debug_log(message):
    "Log message with level debug"
    if _DEBUG_ON and _vs_logger:
        _vs_logger.debug(str(message))

def logmethod(f):
    "Creates a decorator to log a method"
    @wraps(f)
    def wrapper(self, *args, **kwds):
        debug_log("%s in %s called" % (f.__name__, self.__class__.__name__))
        return f(self, *args, **kwds)
    return wrapper

def exception_log(message):
    "Log message with level error plus exception traceback"
    if _vs_logger:
        _vs_logger.exception(str(message))
```

## logging continued ..

And here's an example program showing how I would use `my_logger`.

Immediately after importing it, I would call `set_logger()` to set up the logging framework, and then at any point I can insert `debug_log` and `info_log` calls.

`info_log` calls will always be logged.

`debug_log` calls will only be logged if the global variable `_DEBUG_ON` set at the top of `my_logger` is `True`.

The example shows how `exception_log` would be used in an exception, and also how the method decorator could be used.

```
from my_logger import *

# set up the logger
set_logger()

# now, in any part of my program I can record
# a 'debug' message - if the global _DEBUG_ON
# is True - otherwise nothing will be recorded
debug_log("my program has reached this point")

# I can also record an 'info' message - which is
# recorded whether _DEBUG_ON is True or False
# presumably because more important information
# is needed to be saved
info_log("This is important")

# I also like to record exceptions
try:
    # a block of code which may cause an error
    x = 1/0
    # That usually does it!
except Exception, e:
    # handle the error
    # and record it
    exception_log(e)

info_log("so - we are now past the exception")

# as a final example, I sometimes like
# to record if a method
# in a class has been called

class MyClass:

    @logmethod
    def mymethod(self):
        pass

x = MyClass()

# and calling its method should
# create a log if _DEBUG_ON is True
x.mymethod()
```

## logging continued ..

So running the example program gives the output in 'logfile.txt' of:

```
2009-07-18 14:09:22,383 - DEBUG - my program has reached this point
2009-07-18 14:09:22,384 - INFO - This is important
2009-07-18 14:09:22,384 - ERROR - integer division or modulo by zero
Traceback (most recent call last):
  File "example.py", line 20, in <module>
    x = 1/0
ZeroDivisionError: integer division or modulo by zero
2009-07-18 14:09:22,384 - INFO - so - we are now past the exception
2009-07-18 14:09:22,385 - DEBUG - mymethod in MyClass called
```

The logging module has far more capabilities than I've shown here.

I particularly like to use the method I've described in GUI applications. It's not much use if a gui app spits out exception calls to std out, and also 'print' statements scattered over a gui for diagnostic purposes are a bit off. But recording diagnostics to a log file is very useful - they can be viewed at leisure and the log file can be passed to interested parties, using the debug option also allows further detail just by setting a single variable.

## Beginners Bits - Lists.

An example of a list is:

```
a = [ 3, 6, 0, "Hello", 2.9 ]
```

The square brackets define it as a list and each element in the list is separated by commas. The elements can be any Python object. Each element of the list can be accessed by its index, which is 0 for the first element, so for example:

```
print a[1]      # would print the value 6
```

The 'in' keyword is used in two ways with list; as a test that an object is in the list:

```
if 3 in a:  
    print "yes the value 3 is indeed in the list a"
```

and 'in' is also used with the for loop to go through each item in turn;

```
for x in a:  
    print x
```

would print each of the items in a

Another very useful way of using lists in the 'for' loop is with the 'enumerate' function

```
for key, item in enumerate(a):  
    print key, item
```

As this loops through the elements of list a, the 'key' takes values 0, 1, 2, etc., whereas 'item' takes the actual values of the list elements. So this is a good way to get both index and value together.

There are a several functions associated with lists, perhaps the most commonly used being the append method, for example:

```
a.append(5)
```

Which appends an item to a list - in this case appending the element 5 to list a so a would now be [ 3, 6, 0, "Hello", 2.9, 5 ]

```
and:  
x = a.pop(i)
```

Which returns the element from a with index i, and deletes that element from the list, so  
x = a.pop(1)

would give an x value of 6 and leave a as [ 3, 0, "Hello", 2.9, 5 ]

## Warts and all

It's all very well writing about how great Python is - but what about the warts? What about the real crappy stuff?

Well there's not a lot, as in fact, it's a damn fine language.

There's the packaging of course, all this distutils, eggs and stuff seems fussy. The Freeze, py2exe and that ilk, well - I'm not sure. It all seems strangely complicated. Why not just bang packages under site-packages and leave it at that? Am I missing something?

The GIL doesn't bother me, other people seem to get annoyed by it - don't know why.

Embedding C may be possible, Python is often said to be a glue language - but when I've tried to read up and understand the topic, my opinion is - it's remarkable that anyone has done it at all. In fact I have my doubts that they have, it doesn't seem likely, there's some con trick going on there.

The move from 2. to 3. is a bit painful, all those libraries need sorting out. On the other hand, these things have to be done, if not then the language becomes obsolete. On the whole I'm for it - just something we have to put up with.

As for GUI's - I curse all GUI's.

I think we should start again from scratch; create a GUI that's pythonic, is completely idiosyncratic, but - and here's where it should differ from all other known GUI's - it should actually work.

Though of course, it won't have a decent print mechanism, no GUI has, that's just one of those things beyond the capability of the human race. It is not meant to be. Getting the little dots to fit on a piece of paper in the way you want them to is eye wateringly difficult, oh you think you've done it - but then put it in landscape and expand the view HAA HA HA Ha Ha Haaa.... er .. sorry about that.

Web Frameworks - are suspiciously close to being GUI's, and therefore come under the same curse. I find it hard to believe the human race has been working with computers for half a century, and we still haven't come up with a reasonably effective human-computer interface. It's no wonder most technical people prefer the command line - not that I'm a great fan of that either.

Anything to do with writing web applications is a waste of time - it's obsolete almost as soon as it's out of the door. Put in a lot of work into web-enabling an application and you might as well throw your life away, in a couple of years there will be the next great thing, and everything prior will be history.

Of course I realise you may have no choice - well, you have my sympathy, but life is short, just walk away, go do something else.

But I'm digressing, that's not really Python's problem after all.

What about decorator's? they seem to be elegant, powerful things - but I don't like funny symbols. As soon as you throw in another symbol, then readability takes a hit. The new annotations are getting that way to. I think the pace should slow down a bit - no more funny symbols, or the like. I am forced to agree that where a decorator is used sensibly - like checking logged in status before a function - then it removes clutter, and hence improves readability. So, OK, I'll go with it, but if another symbol comes along I may have to write a stern letter to someone.

What else is annoying - the speed? Not particularly a problem, it's as fast as most other interpreted languages, and chips are getting faster all the time. It's a rather nice thought that as hardware becomes more powerful Python will invade the territory of lower level languages. It's happened before. In fact - this is where Python will eventually supercede languages such as C# and Java; its easier - and as the hardware improves, that will become more important. Python is the future.

There is the difference between mutable and immutable variables, in a perfect language would they all simply be mutable with no distinction? Is this an artefact imposed by the implementation of the language rather than a desired feature? Or is this something good that I'm not aware of - perhaps a computer scientist out there could tell me.

So if that's it - I've not actually got anything to say against the language at all - just those peripheral things you can't help dealing with. The core language is so simple and clean there's not much to complain about, and all the complicated stuff happens in libraries.

Talking of libraries, some are better than others. As a basic principle - if it is hard to figure out how to use it, then the library is crap, and will be replaced as someone creates a better one. That's evolution.

On the other hand, there are plenty of great things about the language:

The namespace concept is extremely neat - packages.modules.classes.methods all joined by those little dots. Though I have yet to use the fact that a function is a namespace as I just can't think of an application for it.

A few of the newer things are good to - list comprehensions are fantastic, and have cleaned up code while retaining readability - now that's what I like.

All the iterator stuff has a true and proper feel to it.

The clear syntax, generally free of symbols, and the way that classes can encapsulate an awful lot of complexity out of sight, is just great.

So in summary; Python rocks!