PyCounters Documentation

Release 0.6

Boaz Leskes

June 16, 2013

CONTENTS

A light weight library to monitor performance and events in production systems.

CHAPTER

ONE

TYPICAL USE CASES

- Number of items/requests processed per second.
- Average processing time of items.
- Average waiting time on resources/locks.
- Time spent in DB layer.
- Cache hit/miss rates.

CHAPTER

TWO

SOME SIMPLE EXAMPLES

2.1 Measuring execution frequency

Count the number of times per second a function is executed:

```
from pycounters.shortcuts import frequency

@frequency()
def f():
    """ some interesting work like serving a request """
    pass
```

Note: Measurements are done by averaging out a sliding window of 5 minutes. Window size is configurable.

2.2 Measuring average executing time

Count the average wall clock time a function runs:

```
from pycounters.shortcuts import time

@time()
def f():
    """ some interesting work like serving a request """
    pass
```

Note: PyCounter's shortcut decorator will use the function name in it's output. This can be configured (see *Shortcut functions*).

2.3 Measuring custom event frequency

Counting some event somewhere in your code:

```
from pycounters.shortcuts import occurrence

def some_code():
    ...
```

```
if TEST_FOR_SOMETHING:
    occurrence("event_name")
...
```

CHAPTER

THREE

NICE, BUT IS IT JUST THAT SIMPLE?

Well, almost (see *Moving Parts* for a complete answer.) To let the counters report their statistics you need to initialize an instance of the LogReporter:

```
import pycounters
import logging

reporter=pycounters.reporters.LogReporter(logging.getLogger("counters"))
pycounters.register_reporter(reporter)
pycounters.start_auto_reporting(seconds=300)
```

Once adding this code, all the counters will periodically report their stats to a log named "counters". Here is an example:

```
2011-06-03 18:12:44,881 | 9130|1286490432 | counters | INFO | posting 0.589342236519
2011-06-03 18:12:44,888 | 9130|1286490432 | counters | INFO | search 1.47849245866
```

Note: The above logs indicate that the search function took 1.48 seconds on average to execute. The posting function took only 0.59 seconds.

CHAPTER

FOUR

INSTALLATION

Easy install PyCounters to get it up and running:

```
easy_install pycounters
```

Take a look at the *Tutorial* for more details.

CHAPTER

FIVE

COOL, BUT IT WOULD BE GREAT IF ...

PyCounters is in it's early stages. If you have any ideas for improvements, features which are aboslutely a must or things you feel are outright stupid - I'd love to hear. Make ticket on https://bitbucket.org/bleskes/pycounters/issues .

Here is what I have in mind so far:

- Django integration (I'm currently working on this)
- Geckoboard output

Of course, you are more then welcome to browse and/or fork the code: https://bitbucket.org/bleskes/pycounters

CHAPTER

SIX

FURTHER READING

6.1 Tutorial

6.1.1 Installing pycounters

PyCounters is pure python. All you need is to run easy_install (or pip):

```
easy_install pycounters
```

Of course, you can always checkout the code from BitBucket on https://bitbucket.org/bleskes/pycounters

6.1.2 Introduction

PyCounters is a library to help you collect interesting metrics from production code. As an case study for this tutorial, we will use a simple Python-based server (taken from the python docs):

```
import SocketServer

class MyTCPHandler(SocketServer.BaseRequestHandler):
    """
    The RequestHandler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print "%s wrote:" % self.client_address[0]
        print self.data
        # just send back the same data, but upper-cased
        self.request.send(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    server = SocketServer.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
```

```
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

6.1.3 Step 1 - Adding Events

For this basic server, we will add events to report the following metrics:

- Number of requests per second
- Average time for handling a request

Both of these metrics are connected to the handle method of the MyTCPHandler class in the example. The number of requests per second the server serves is exactly the number of times the handle() method is called. The average time for handling a request is exactly the average execution time of handle()

Both of these metrics are measure by decorating handle() the *shortcut* decorators `frequency` and `time`:

```
import SocketServer
from pycounters import shortcuts

class MyTCPHandler(SocketServer.BaseRequestHandler):
    ...

    @shortcuts.time("requests_time")
    @shortcuts.frequency("requests_frequency")
    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print "%s wrote:" % self.client_address[0]
        print self.data
        # just send back the same data, but upper-cased
        self.request.send(self.data.upper())
```

Note:

- Every decorator is given a name ("requests_time" and "requests_frequency"). These names will come back in the report generated by PyCounters. More on this in the next section.
- The shortcut decorators actually do two things - report events and add counters for them. For now, it's OK but you might want to separate the two. More on this later in the tutorial

---

6.1.4 Step 2 - Reporting

Now that the metrics are being collected, they need to be reported. This is the job of the *reporters*. In this example, we'll save a report every 5 minutes to a JSON file at /tmp/server.counters.json (check out the *Reporters* section for other options). To do so, create an instance of `JSONFileReporter` when the server starts:

```
import SocketServer
from pycounters import shortcuts, reporters, start_auto_reporting, register_reporter

....

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    JSONFile = "/tmp/server.counters.json"
```

```
    reporter = reporters.JSONFileReporter(output_file=JSONFile)
    register_reporter(reporter)

    start_auto_reporting()


    # Create the server, binding to localhost on port 9999
    server = SocketServer.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

Note: To make pycounters periodically output a report you must call start_auto_reporting()

By default auto reports are generated every 5 minutes (change that by using the seconds parameter of start_auto_reporting()). After five minutes the reporter will save it's report. Here is an example of the contest of /tmp/server.counters.json:

```
{"requests_time": 0.00039249658584594727, "requests_frequency": 0.014266581369872909}
```

6.1.5 Step 3 - Counters and reporting events without a decorator

Average request time and request frequency were both nicely measured by decorating MyTCPHandler::handle(). Some metrics do not fit as nicely into the decorator model.

The server in our example receives a string from the a client and returns it upper_cased. Say we want to measure the average number of characters the server processes. To achieve this we can use another shortcut function `value`:

```
import SocketServer
from pycounters import shortcuts

class MyTCPHandler(SocketServer.BaseRequestHandler):
    ...

    @shortcuts.time("requests_time")
    @shortcuts.frequency("requests_frequency")
    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print "%s wrote:" % self.client_address[0]
        print self.data

        # measure the average length of data
        shortcuts.value("requests_data_len",len(self.data))

        # just send back the same data, but upper-cased
        self.request.send(self.data.upper())
```

Until now, the shortcut decorators and functions were perfect for what we wanted to do. Naturally, this is not always the case. Before going on, it is handy to explain more about these shortcuts and how PyCounters work (see *Moving Parts* for more about this).

PyCounters is built of three main building blocks:

- ***Events* - to reports values and occurrences in your code (in the example: incoming request, the time it took to** process them and the number of bytes the processed).

- *Counters* - to capture events and analyse them (in the example: measuring requests per second, averaging request processing time and averaging the number of bytes processed per request).
- *Reporters* - to periodically generate a report of all active counters.

PyCounters' shortcuts will both report events and create a counter to analyse it. Every shortcut has a default counter type but you can override it (see *Shortcuts*). For example, say we wanted to measure the *total* number of bytes the server has processed rather than the average. To achieve this, the "requests_data_len" counter needs to be changed to `TotalCounter`. The easiest way to achieve this is to add a parameter to the shortcut `shortcuts.value("requests_data_len",len(data),auto_add_counter=TotalCounter)` (don't forget to change your imports too). However, we will go another way about it.

PyCounter's event reporting is very light weight. It practically does nothing if no counter is defined to capture those events. Because of this, it is a good idea to report all important events through the code and choose later what you exactly want analyzed. To do this we must separate event reporting from the definition of counters.

Note: When you create a counter, it will by default listen to one event, *named exactly as the counter's name*. However, if the events parameter is passed to a counter at initialization, it will listen *only* to the specified events.

Note: This approach also means you can analyze things differently on a single thread, by installing thread specific counters. For example, trace a specific request more heavily due to some debug flag. Thread specific counters are not currently available but will be in the future.

Reporting an event without defining a counter is done by using one of the functions described under *Event reporting* . Since we want to report a value, we will use `pycounters.report_value()`:

```
import SocketServer
from pycounters import shortcuts,reporters,report_value

class MyTCPHandler(SocketServer.BaseRequestHandler):
    ...

    @shortcuts.time("requests_time")
    @shortcuts.frequency("requests_frequency")
    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print "%s wrote:" % self.client_address[0]
        print self.data

        # measure the average length of data
        report_value("requests_data_len",len(self.data))

        # just send back the same data, but upper-cased
        self.request.send(self.data.upper())
```

To add the `TotalCounter` counter, we change the initialization part of the code:

```
import SocketServer
from pycounters import shortcuts, reporters, report_value,counters, register_counter, start_auto_repo

....

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    JSONFile = "/tmp/server.counters.json"
```

```
    data_len_counter = counters.TotalCounter("requests_data_len") # create the counter
    register_counter(data_len_counter) # register it, so it will start processing events

    reporter = reporters.JSONFileReporter(output_file=JSONFile)
    register_reporter(reporter)

    start_auto_reporting()


    # Create the server, binding to localhost on port 9999
    server = SocketServer.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

6.1.6 Step 4 - A complete example

Here is the complete code with all the changes so far (also available at the PyCounters repository):

```
import SocketServer
from pycounters import shortcuts, reporters, register_counter, counters, report_value, register_repo

class MyTCPHandler(SocketServer.BaseRequestHandler):
    """
    The RequestHandler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    @shortcuts.time("requests_time")
    @shortcuts.frequency("requests_frequency")
    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print "%s wrote:" % self.client_address[0]
        print self.data

        # measure the average length of data
        report_value("requests_data_len",len(self.data))

        # just send back the same data, but upper-cased
        self.request.send(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    JSONFile = "/tmp/server.counters.json"

    data_len_counter = counters.TotalCounter("requests_data_len") # create the counter
    register_counter(data_len_counter) # register it, so it will start processing events

    reporter = reporters.JSONFileReporter(output_file=JSONFile)
    register_reporter(reporter)

    start_auto_reporting()
```

```
    # Create the server, binding to localhost on port 9999
    server = SocketServer.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

6.1.7 Step 5 - More about Events and Counters

In the above example, the MyTCPHandler::handle method is decorated with two short functions: `frequency` and `time`: . This is the easiest way to set up PyCounters to measure things but it has some down sides. First, every shortcut decorate throws it's own events. That means that for every execution of the handle method, four events are sent. That is inefficient. Second, and more importantly, it also means that Counters definition are spread around the code.

In bigger projects it is better to separate event throwing from counting. For example, we can decorate the handle function with `report_start_end`:

```
@pycounters.report_start_end("request")
def handle(self):
    # self.request is the TCP socket connected to the client
```

And define two counters to analyze 'different' statistics about this function:

```
avg_req_time = counters.AverageTimeCounter("requests_time",events=["request"])
register_counter(avg_req_time)

req_per_sec = counters.FrequencyCounter("requests_frequency",events=["request"])
register_counter(req_per_sec)
```

Note: Multiple counters with different names can be set up to analyze the same event using the events argument in their constructor.

Doing things this way has a couple of advantages:

- It is conceptually cleaner - you report what happened and measure multiple aspects of it
- It is more flexible - you can easily analyse more things about your code by simply adding counters.
- You can decide at runtime what to measure (by changing registered counters)

6.1.8 Step 6 - Another example of using Events and Counters

In this example we will create a few counters listening to the same events. Let say, we want to get maximum, minimum, average and sum of values of request data length in 15 minutes window. To achieve this, we need to create 4 counters, all of them listening to 'requests_data_len' event.

```
import SocketServer
from pycounters import shortcuts, reporters, register_counter, counters, report_value, register_repor

class MyTCPHandler(SocketServer.BaseRequestHandler):
    """
    The RequestHandler class for our server.
```

```
    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    @shortcuts.time("requests_time")
    @shortcuts.frequency("requests_frequency")
    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print "%s wrote:" % self.client_address[0]
        print self.data

        # measure the average length of data
        report_value("requests_data_len",len(self.data))

        # just send back the same data, but upper-cased
        self.request.send(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    JSONFile = "/tmp/server.counters.json"

    data_len_avg_counter = counters.AverageWindowCounter("requests_data_len_avg",\
        events=["requests_data_len"], window_size=900) # create the avarage window counter
    register_counter(data_len_avg_counter) # register it, so it will start processing events

    data_len_total_counter = counters.WindowCounter("requests_data_len_total",\
        events=["requests_data_len"], window_size=900) # create the window sum counter
    register_counter(data_len_total_counter)

    data_len_max_counter = counters.MaxWindowCounter("requests_data_len_max",\
        events=["requests_data_len"], window_size=900) # create the max window counter
    register_counter(data_len_max_counter)

    data_len_min_counter = counters.MinWindowCounter("requests_data_len_min",\
        events=["requests_data_len"], window_size=900) # create the min window counter
    register_counter(data_len_min_counter)

    reporter = reporters.JSONFileReporter(output_file=JSONFile)
    register_reporter(reporter)

    start_auto_reporting()


    # Create the server, binding to localhost on port 9999
    server = SocketServer.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

You can change size of window by specifying different window_size parameter when creating a counter.

6.1.9 Step 7 - Utilities

In the example so far, we've outputted the collected metrics to a JSON file. Using that JSON file, we can easily build simple tools to report the metrics further. The *Utilities reference* package contains a set of utilities to help building such tools.

At the moment, PyCounter comes with a utility to help writing munin plugins. Here is an example of a munin plugin that taks the JSON report procude by the Tutorial and presents it in the way munin understands:

```
#!/usr/bin/python

from pycounters.utils.munin import Plugin

config = [
    {
        "id" : "requests_per_sec",
        "global" : {
            # graph global options: http://munin-monitoring.org/wiki/protocol-config
            "title" : "Request Frequency",
            "category" : "PyCounters example"
        },
        "data" : [
            {
                "counter" : "requests_frequency",
                "label"   : "requests per second",
                "draw"    : "LINE2",
            }
        ]
    },
    {
        "id" : "requests_time",
        "global" : {
            "title" : "Request Average Handling Time",
            "category" : "PyCounters example"
        },
        "data" : [
            {
                "counter" : "requests_time",
                "label"   : "Average time per request",
                "draw"    : "LINE2",
            }
        ]
    },
    {
        "id" : "requests_total_data",
        "global" : {
            "title" : "Total data processed",
            "category" : "PyCounters example"
        },
        "data" : [
            {
                "counter" : "requests_data_len",
                "label"   : "total bytes",
                "draw"    : "LINE2",
            }
        ]
    }

]
```

```
p = Plugin("/tmp/server.counters.json",config) # initialize the plugin

p.process_cmd() # process munin command and output requested data or config
```

Try it out (after the server has run for more than 5 minutes and a report was outputted to the JSON file) by running `python munin_plugin config` and `python munin_plugin`.

6.1.10 Step 8 - Multiprocess support

Some application (like a web server) do not run in a single process. Still, you want to collect global metrics like the ones discussed before in this tutorial.

PyCounters supports aggreating information from multiple running processes. To do so call `pycounters.configure_multi_process_collection()` on every process you want to aggregate data from. The parameters to this method will tell PyCounters what port to use for aggregation and, if running on multiple servers, which server to collect data on.

6.2 Moving Parts

PyCounters architecture is built around three main concepts:

- *Events* reporting (start and end of functions, numerical values etc.)
- *Counters* for collecting the above events and analyzing them (on demand).
- *Reporters* for outputting the collected statistics.

In short, PyCounters is built to allow adding event reporting with piratically no performance impact. Counters add some minimal overhead. Only on output does PyCounters do some calculation (every 5 minutes, depending on configuration).

When using PyCounters, consider the following:

- Triggering events is extremely lite weight. All events with no corresponding Counters are ignored.
- Therefore you can add as many events as you want.
- Counters can be registered and unregistered on demand. Only collect what you need.
- Outputting is a relatively rare event - don't worry about the calculation it does.

6.2.1 Events

PyCounters defines two types of events:

start and end events Start and end events are used to report the start and end of a function or any other block of code. These events are typically caught by timing counters but anything is possible. Start and end events should be reported through the `report_start()` , `report_end()` or the `report_start_end()` decorator.

value events These events report a value to the counters. You typically use these to track averages of things but you can get creative. For example - reporting 1 on a cache hit and 0 on a cache miss to an AverageWindowCounter will give you the average rate of cache hits. Value events can be reported by using the `report_value()` function.

Note: There is no special way in PyCounters to create new event it is enough, to create a counter listening to that event.

6.2.2 Counters

All the "smartness" of PyCounters is bundled withing a set of Counters. Counters are in charge of intercepting and interpreting events reported by different parts of the program. As mentioned before, you can register a Counter when you want to analyze specific events (by default events of identical name, if you need more control, use events parameter). You do so by using the `register_counter()` function:

```
counter = AverageWindowCounter("some_name")
register_counter(counter)
```

You can also unregister the counter once you don't need it anymore:

```
unregister_counter(counter=counter)
```

or by name:

```
unregister_counter(name="some_name")
```

Note: After unregistering the counter all events named "some_name" will be ignored (unless some other counter listens to them).

Note: You can only register a single counter for any given name.

6.2.3 Reporters

Reporters are used to collect a report from the currently registered Counters. Reporters are not supposed to run often as that will have a performance impact.

At the moment PyCounters can only output to python logs and JSON files. For example, to output to logs, create an instance of `LogReporter` . You can then manually output reports (using `output_report`) or turn on auto reporting (using `start_auto_reporting` .)

```
reporter=pycounters.reporters.LogReporter(logging.getLogger("counters"))
pycounters.register_reporter(reporter)
#... some where later
pycounters.output_report()
```

6.2.4 Shortcuts

These are functions which both report events and auto add the most common Counter for them. See *Shortcut functions* for more details and *Some simple examples* in the main documentation page for usage examples.

6.3 Object and function reference

6.3.1 Event reporting

pycounters.**report_start**(*name*)
reports an event's start. NOTE: you *must* fire off a corresponding event end with report_end

pycounters.**report_end**(*name*)
reports an event's end. NOTE: you *must* have fired off a corresponding event start with report_start

pycounters.**report_start_end**(*name=None*)
returns a function decorator and/or context manager which raises start and end events. If name is None events name is set to the name of the decorated function. In that case report_start_end can not be used as a context manager.

pycounters.**report_value**(*name*, *value*)
reports a value event to the counters.

6.3.2 Counters

class pycounters.counters.**EventCounter**(*name*, *events=None*)
Counts the number of times an end event has fired.

clear(*dump=True*)
Clears the stored information

get_value()
gets the value of this counter

report_event(*name*, *property*, *param*)
reports an event to this counter

class pycounters.counters.**TotalCounter**(*name*, *events=None*)
Counts the total of events' values.

clear(*dump=True*)
Clears the stored information

get_value()
gets the value of this counter

report_event(*name*, *property*, *param*)
reports an event to this counter

class pycounters.counters.**AverageWindowCounter**(**args*, ***kwargs*)
Calculates a running average of arbitrary values

clear(*dump=True*)
Clears the stored information

get_value()
gets the value of this counter

report_event(*name*, *property*, *param*)
reports an event to this counter

class pycounters.counters.**AverageTimeCounter**(**args*, ***kwargs*)
Counts the average time between start and end events

clear(*dump=True*)
Clears the stored information

get_value()
gets the value of this counter

report_event(*name*, *property*, *param*)
reports an event to this counter

class pycounters.counters.**FrequencyCounter**(**args*, ***kwargs*)
Use to count the frequency of some occurrences in a sliding window. Occurrences can be reported directly via a value event (X occurrences has happened now) or via an end event which will be interpreted as a single occurrence.

clear(*dump=True*)
Clears the stored information

get_value()
gets the value of this counter

report_event(*name*, *property*, *param*)
reports an event to this counter

class pycounters.counters.**WindowCounter**(**args*, ***kwargs*)
Counts the number of end events in a sliding window

clear(*dump=True*)
Clears the stored information

get_value()
gets the value of this counter

report_event(*name*, *property*, *param*)
reports an event to this counter

class pycounters.counters.**MaxWindowCounter**(**args*, ***kwargs*)
Counts maximum of events values in window

clear(*dump=True*)
Clears the stored information

get_value()
gets the value of this counter

report_event(*name*, *property*, *param*)
reports an event to this counter

class pycounters.counters.**MinWindowCounter**(**args*, ***kwargs*)
Counts minimum of events values in window

clear(*dump=True*)
Clears the stored information

get_value()
gets the value of this counter

report_event(*name*, *property*, *param*)
reports an event to this counter

6.3.3 Reporters

class pycounters.reporters.**LogReporter**(*output_log=None*)
Log based reporter.

class pycounters.reporters.**JSONFileReporter**(*output_file=None*)
Reports to a file in a JSON format.

static safe_read(*filename*)
safely reads a value in a JSON format frome file

static safe_write(*value*, *filename*)
safely writes value in a JSON format to file

pycounters.**register_reporter**(*reporter=None*)
add a reporter to PyCounters. Registered reporters will output collected metrics

pycounters.**unregister_reporter**(*reporter=None*)
remove a reporter from PyCounters.

pycounters.**output_report**()
Manually cause the current values of all registered counters to be reported.

pycounters.**start_auto_reporting**(*seconds=300*)
Start reporting in a background thread. Reporting frequency is set by seconds param.

Multi-process reporting

pycounters.**configure_multi_process_collection**(*collecting_address=[('', 60907), ('', 60906)], timeout_in_sec=120, role=2*)
configures PyCounters to collect values from multiple processes

Parameters

- **collecting_address** – a list of (address,port) tuples address of machines and ports data should be collected on. the extra tuples are used as backup in case the first address/port combination is (temporarily) unavailable. PyCounters would automatically start using the preferred address/port when it becomes available again. This behavior is handy when restarting the program and the old port is not yet freed by the OS.
- **timeout_in_sec** – timeout configuration for connections. Default should be good enough for pratically everyone.
- **role** – the role of this process. Leave at the default of AUTO_ROLE for pycounters to automatically choose a collecting leader.

6.3.4 Registering counters

pycounters.**register_counter**(*counter*, *throw_if_exists=True*)
Register a counter with PyCounters

pycounters.**unregister_counter**(*counter=None*, *name=None*)
Removes a previously registered counter

6.3.5 Shortcut functions

pycounters.shortcuts.**count**(*name=None*, *auto_add_counter=<class 'pycounters.counters.types.EventCounter'>*)

A shortcut decorator to count the number times a function is called. Uses the `counters.EventCounter` counter by default. If the parameter name is not supplied events are reported under the name of the wrapped function.

pycounters.shortcuts.**frequency**(*name=None*, *auto_add_counter=<class 'pycounters.counters.types.FrequencyCounter'>*)

A shortcut decorator to count the frequency in which a function is called. Uses the `counters.FrequencyCounter` counter by default. If the parameter name is not supplied events are reported under the name of the wrapped function.

pycounters.shortcuts.**occurrence**(*name*, *auto_add_counter=<class 'pycounters.counters.types.FrequencyCounter'>*)

A shortcut function reports an occurrence of something. Uses the `counters.FrequencyCounter` counter by default.

pycounters.shortcuts.**time**(*name=None*, *auto_add_counter=<class 'pycounters.counters.types.AverageTimeCounter'>*)

A shortcut decorator to count the average execution time of a function. Uses the `counters.AverageTimeCounter` counter by default. If the parameter name is not supplied events are reported under the name of the wrapped function.

pycounters.shortcuts.**value**(*name*, *value*, *auto_add_counter=<class 'pycounters.counters.types.AverageWindowCounter'>*)

A shortcut function to report a value of something. Uses the `counters.AverageWindowCounter` counter by default.

6.4 Utilities reference

6.4.1 A helper class for Munin plugins

class pycounters.utils.munin.**Plugin**(*json_output_file=None*, *config=None*, *max_file_age_in_seconds=900*)

a small utility to write munin plugins based on the output of the JSONFile reporter

example usage (munin_plugin.py) :

```
#!/usr/bin/python

from pycounters.utils.munin import Plugin

config = [
    {
        "id" : "graph_id",
        "global" : {
            # graph global options: http://munin-monitoring.org/wiki/protocol-config
            "title" : "Title",
            "info"  : "Some info",
            "category" : "PyCounters"
        },
        "data" : [
            {
                "counter" : "Somepycountername",
                "label"   : "A human redable form",
```

```
                "draw"    : "LINE2"
            }
            #...

        ]
    }
]

p = Plugin("pycounters_output_file.json", config) # initialize the plugin

p.process_cmd() # process munin command and output requested data or config
```

output_config(*config*)
executes the config command

output_data(*config*)
executes the data command

process_cmd()
process munin command and output requested data or config

PYTHON MODULE INDEX

p