pikos Documentation

Release

Author

February 27, 2015

Contents

Pikos is a profiling and investigation tool suite for python applications. The name is inspired by Pikos Apikos the main character in a mid 80s Greek puppet TV series. Pikos was an investigative journalist assigned to find out about a missing person case in the remote and strange land of “Froutopia”, a country populated by large fruits that can talk.

Key aims of Pikos are:

- Help identify areas of the an application that need to improve.
- Use, group and augment rather than replace commonly used tools like cProfile and line_profiler
- Provide effective memory monitoring throughout python.
- Be multi-platform.
- Provide real-time access to profile data and allow live analysis while the application is running.

CHAPTER 1

Repository

Pikos is hosted on github: https://github.com/enthought/pikos

CHAPTER 2

Installation

The package requires a recent version of psutil (>=0.4.1):

```
python setup.py install
```

To build with the real-time fork of cProfile please provide the *–with-real-time-lsprof* before any setup command:

```
python setup.py --with-real-time-lsprof install
```

You will need a build of libzmq to compile and link against. If the needed files are not available at system default locations, they will need to be manually provided to the `build_ext` command:

```
python setup.py --with-real-time-lsprof build_ext -I <include directory for zmq> -L <libary directory
python setup.py --with-real-time-lsprof install
```

or in one line as:

```
python setup.py --with-real-time-lsprof build_ext -I <include directory for zmq> -L <library directo
```

Finally to run the test suite please give:

```
python setup.py test
```

Optional packages of external profilers:

- yappi (>=0.62), http://code.google.com/p/yappi/
- line_profiler (>=1.0b3), http://pypi.python.org/pypi/line_profiler

Optional packages for the live monitoring tools:

- pyzmq (>= 2.1.11) http://www.zeromq.org/bindings:python
- traits (>= 4.1.0) https://github.com/enthought/traits
- traitsui (>= 4.1.0) https://github.com/enthought/traitsui
- pyface (>= 4.1.0 https://github.com/enthought/pyface
- envisage (>= 4.1.0 https://github.com/enthought/envisage
- chaco (>= 4.1.0) https://github.com/enthought/chaco
- numpy (>= 1.6.1) http://numpy.scipy.org

CHAPTER 3

Usage

The main component in the pikos toolset is the *Monitor*. A monitor creates a number of records during the execution of the code which are passed on the recorder to be stored into memory or file.

3.1 In code

Monitors can be used programmatically in a number of ways.

1. Enabled/Disabled using the corresponding functions:

```
from pikos.api import screen
from pikos.monitors.api import FunctionMonitor

monitor = Monitor(recorder=screen())
monitor.enable()

# monitored code
#

monitor.disable()
```

2. A monitor instance can be used as a context manager:

```
from pikos.api import screen
from pikos.monitors.api import FunctionMonitor

monitor = Monitor(recorder=screen())

with monitor:
    # monitored code
    #
    pass
```

3. With the use of the *attach* method a monitor becomes a decorator:

```
from pikos.api import screen
from pikos.monitors.api import FunctionMonitor

monitor = Monitor(recorder=screen())

@monitor.attach
def monitored_function():
    # monitored code
```

```
    #
    pass
```

4. Finally the `pikos.api` module provides easy to use decorator factories for the standard monitors. The factories can optionally accept a recorder and dictate if a focused monitor should be used:

```
from pikos.api import function_monitor, csv_file

@function_monitor(recorder=csv_file(), focused=True)
def monitored_function():
    # monitored code
    #
    pass
```

3.2 Command line

The standard pikos monitors can be also used throught a command prompt tool, *pikos-run*:

```
usage: pikos-run [-h] [-o OUTPUT] [--buffered] [--recording {screen,text,csv}]
                 [--focused-on FOCUSED_ON]
                 {functions,line_memory,lines,function_memory} script

Execute the python script inside the pikos monitor context.

positional arguments:
  {functions,line_memory,lines,function_memory}
                        The monitor to use
  script                The script to run.

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        Output results to a file
  --buffered            Use a buffered stream.
  --recording {screen,text,csv}
                        Select the type of recording to use.
  --focused-on FOCUSED_ON
                        Provide the module path(s) of the method where
                        recording will be focused. Comma separated list of
                        importable functions
```

3.3 Example

Given the code bellow:

```
""" Mandelbrot Set example

Code from the Tentative Numpy Tutorial

url: http://wiki.scipy.org/Tentative_NumPy_Tutorial/Mandelbrot_Set_Example

"""

from numpy import *
```

```
import pylab


def mandelbrot(h, w, maxit=20):
    '''Returns an image of the Mandelbrot fractal of size (h,w).
    '''
    y,x = ogrid[-1.4:1.4:h*1j, -2:0.8:w*1j]
    c = x+y*1j
    z = c
    divtime = maxit + zeros(z.shape, dtype=int)

    for i in xrange(maxit):
        z = z**2 + c
        diverge = z*conj(z) > 2**2            # who is diverging
        div_now = diverge & (divtime==maxit)  # who is diverging now
        divtime[div_now] = i                  # note when
        z[diverge] = 2                        # avoid diverging too much

    return divtime


if __name__ == '__main__':
    pylab.imshow(mandelbrot(400,400))
    pylab.show()
```

Running:

```
pikos-run line_memory examples/mandelbrot_set_example.py --recording csv --focused-on=mandelbrot
```

from the root directory will run the *mandelbrot* example and record the memory usage on function entry and exit while inside the *mandelbrot* method. The monitoring information will be recorded in csv format in the *monitor_records.csv* (default filename).

3.3.1 CSV Sample

```
index,function,lineNo,RSS,VMS,line,filename
0,mandelbrot,16,64679936,382787584,"    y,x = ogrid[-1.4:1.4:h*1j, -2:0.8:w*1j]",examples/mandelbrot_
1,mandelbrot,17,64962560,383229952,    c = x+y*1j,examples/mandelbrot_set_example.py
2,mandelbrot,18,67678208,386056192,    z = c,examples/mandelbrot_set_example.py
3,mandelbrot,19,67817472,386056192,"    divtime = maxit + zeros(z.shape, dtype=int)",examples/mandelb
4,mandelbrot,21,69103616,387338240,    for i in xrange(maxit):,examples/mandelbrot_set_example.py
5,mandelbrot,22,69103616,387338240,        z = z**2 + c,examples/mandelbrot_set_example.py
6,mandelbrot,23,71671808,389902336,        diverge = z*conj(z) > 2**2            # who is diverging,ex
7,mandelbrot,24,76263424,394760192,        div_now = diverge & (divtime==maxit)  # who is diverging no
8,mandelbrot,25,76529664,394760192,        divtime[div_now] = i                  # note when,examples/
9,mandelbrot,26,76529664,394760192,        z[diverge] = 2                        # avoid diverging too
10,mandelbrot,21,76537856,394760192,    for i in xrange(maxit):,examples/mandelbrot_set_example.py
11,mandelbrot,22,76537856,394760192,        z = z**2 + c,examples/mandelbrot_set_example.py
12,mandelbrot,23,74452992,392675328,        diverge = z*conj(z) > 2**2            # who is diverging,e
13,mandelbrot,24,76881920,395235328,        div_now = diverge & (divtime==maxit)  # who is diverging n
14,mandelbrot,25,77012992,395235328,        divtime[div_now] = i                  # note when,examples
15,mandelbrot,26,77012992,395235328,        z[diverge] = 2                        # avoid diverging to
16,mandelbrot,21,77012992,395235328,    for i in xrange(maxit):,examples/mandelbrot_set_example.py
17,mandelbrot,22,77012992,395235328,        z = z**2 + c,examples/mandelbrot_set_example.py
18,mandelbrot,23,79441920,397795328,        diverge = z*conj(z) > 2**2            # who is diverging,e
19,mandelbrot,24,79572992,397795328,        div_now = diverge & (divtime==maxit)  # who is diverging n
```

The first column is the record index, followed by the function name and the line number where (just before execution) the RSS, VMS memory counters for the python process are recorded. The last two column contains the python line of the function.

Note: This record type is specific to the `LineMemoryMonitor`.

3.3.2 Plot Data

loading the csv file into ipython we can plot the graph of record index to RSS memory usage of the python process while executing the *mandelbrot* function:

```
In [1]: import numpy

In [2]: import pylab

In [3]: data = numpy.loadtxt('monitor_records.csv',usecols=[0, 3], delimiter=',', skiprows=1)

In [4]: pylab.plot(data[:, 0], data[:, 1], drawstyle='steps')

In [5]: pylab.show()
```

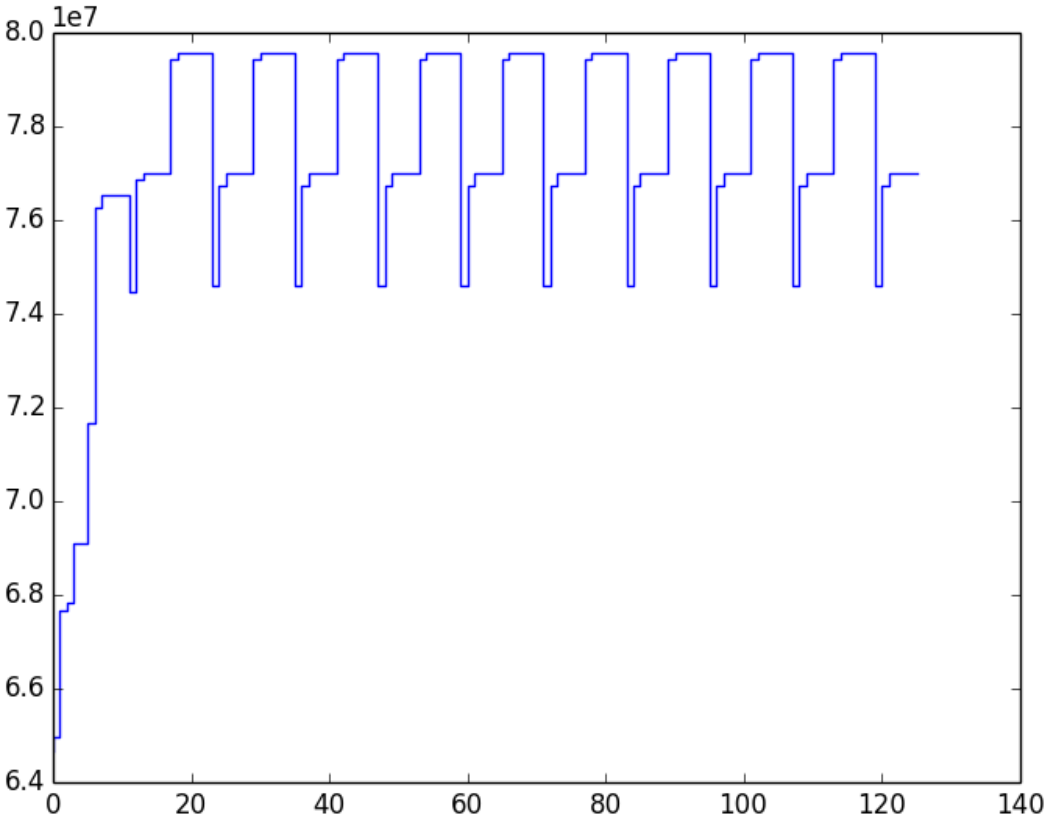

Figure 3.1: The plot of record index vs RSS memory usage (in bytes) of the python process when running the *mandelbrot* function.

CHAPTER 4

Contents

4.1 Library Reference

Pikos is designed in layers. At the top layer we find the `Monitor` a decorator that acts as the entry point for the monitors provided by pikos. The next layer is the various monitors that are responsible to collect information (e.g. memory) during the execution of the decorated function. The retrieved information is recorded through the recorders and controlled with the filters.

4.1.1 Monitor Decorator

class `pikos.monitors.monitor.`**Monitor**

Bases: `object`

Base class of Pikos provided monitors.

The class provides the *.attach* decorating method to attach a pikos monitor to a function or method. Subclasses might need to provide their own implementation if required.

__init__()

x.__init__(...) initializes x; see help(type(x)) for signature

__call__(*...*) <==> *x(...)*

enable()

This method should enable the monitor.

disable()

This method should disable the monitor.

attach(*instance*, *function*)

Attach (i.e. wrap) the monitor to the decorated function.

Basic decoration functionality without any arguments.

Parameters

- **instance** (*object*) – The monitor instance to attach.
- **function** (*callable*) – The function to wrap

Returns

fn [callable] A MonitorAttach instance.

4.1.2 Monitors

A monitor is a context manager object. The class is initialized with a recorder class. Each instance of a monitor class can be reused, the `__enter__` method makes sure that the code that is executed inside the context will be monitored and that the associated recorder has been initialized. During the execution of the decorated function The information is collected into a name tuple and the tuple is forwarded to recorder that has been associated with the current monitor.

Pikos currently provides the following monitors:

`FunctionMonitor`	Record python function events.
`LineMonitor`	Record python line events.
`FunctionMemoryMonitor`	Record process memory on python function events.
`LineMemoryMonitor`	Record process memory on python line events.
`FocusedFunctionMonitor`	Record python function events.
`FocusedLineMonitor`	Record python line events.
`FocusedFunctionMemoryMonitor`	Record process memory on python function events.
`FocusedLineMemoryMonitor`	Record process memory on python function events.

External Monitors

Pikos can act as entry point for external libraries and profilers.

`PythonCProfiler`	The normal python `Profile` subclassed and adapted to work with the pikos Monitor decorator.
`LineProfiler`	
`YappiProfiler`	A pikos compatible profiler class using the yappi library.

Note: These profilers are experimental and not yet integrated fully with the pikos framework. Please check individual documentation for more information.

4.1.3 Recorders

The recorder (a subclass of `AbstractRecorder`) is responsible for recording the tuples from the monitor. What is recordered is controlled by a filter function

Pikos currently provides the following recorders:

`TextStreamRecorder`	The TextStreamRecorder is simple recorder that formats and writes the records directly to a stream.
`TextFileRecorder`	The TextStreamRecorder that creates the file for the records.
`CSVRecorder`	The CSV Recorder is a simple text based recorder that records the tuple of values using a scv writer.
`CSVFileRecorder`	A CSVRecorder that creates the stream (i.e.
`ListRecorder`	The ListRecorder is simple recorder that records the tuple of values in memory as a list.
`ZeroMQRecorder`	The ZeroMQ Recorder is a recorder that publishes each set of values on a 0MQ publish socket.

Note: The standard Recorders are record type agnostic so it is possible to use the same recorder for multiple monitors. However, this feature is experimental and users are advised to use with care.

4.1.4 Filters

A filter controls if a record tuple will be recorded or not. The callable accepts the record and makes a decision based on that (i.e. return `True` or `False`. Functions (normal and lamda) and callable classes can be used in this case to remove clutter and speed up monitoring only of the desired code.

Pikos currently provides the following predefined filters:

`OnValue`	A record filter that returns True if record has a specific value.
`OnChange`	A record filter that checks if the record field has changed.

4.1.5 Records

Each monitor uses a specific record. A record is a subclass of named tuple augmented with two methods, `header`, `line` that can be optionally used to format the output.

Note: Currently only the `TextStreamRecorder` can take advantage of the additional format information.

The monitor records available are:

`FunctionRecord`	The record tuple for function events.
`LineRecord`	The record for line trace events.
`FunctionMemoryRecord`	The record tuple for memory usage on function events.
`LineMemoryRecord`	The record tuple for memory usage on line events.

Monitors

class `pikos.monitors.function_monitor.`**`FunctionMonitor`**(*recorder*, *record_type=None*)

Bases: `pikos.monitors.monitor.Monitor`

Record python function events.

The class hooks on the setprofile function to receive function events and record them.

`__init__`(*recorder*, *record_type=None*)

Initialize the monitoring class.

Parameters

- **recorder** (*object*) – A subclass of `AbstractRecorder` or a class that implements the same interface to handle the values to be logged.
- **record_type** (*type*) – A class object to be used for records. Default is `FunctionRecord`.

`enable`()

Enable the monitor.

The first time the method is called (the context is entered) it will set the setprofile hooks and initialize the recorder.

`disable`()

Disable the monitor.

The last time the method is called (the context is exited) it will unset the setprofile hooks and finalize the recorder.

on_function_event(*frame*, *event*, *arg*)
Record the current function event.

Called on function events, it will retrieve the necessary information from the *frame*, create a `FunctionRecord` and send it to the recorder.

gather_info(*frame*, *event*, *arg*)
Gather information for the record.

class `pikos.monitors.focused_function_monitor.`**FocusedFunctionMonitor**(**arguments*, ***keywords*)

Bases: `pikos.monitors.focused_function_mixin.FocusedFunctionMixin`, `pikos.monitors.function_monitor.FunctionMonitor`

Record python function events.

The class hooks on the setprofile function to receive function events and record them if they take place inside the provided functions.

__init__(**arguments*, ***keywords*)
Initialize the monitoring class.

Parameters

- ***arguments** (*list*) – The list of arguments required by the base monitor. They will be passed on the super class of the mixing
- ****keywords** (*dict*) – Dictionary of keyword arguments. The *functions* keyword if defined should be a list of function or method objects inside which recording will take place.

---

class `pikos.monitors.function_memory_monitor.`**FunctionMemoryMonitor**(*recorder*, *record_type=None*)

Bases: `pikos.monitors.function_monitor.FunctionMonitor`

Record process memory on python function events.

The class hooks on the setprofile function to receive function events and record the current process memory when they happen.

__init__(*recorder*, *record_type=None*)
Initialize the monitoring class.

Parameters **recorder** (*object*) – A subclass of `AbstractRecorder` or a class that implements the same interface to handle the values to be logged.

record_type: class object A class object to be used for records. Default is `FunctionMemoryMonitor`

enable()
Enable the monitor.

The first time the method is called (the context is entered) it will initialize the Process class, set the setprofile hooks and initialize the recorder.

disable()
Disable the monitor.

The last time the method is called (the context is exited) it will unset the setprofile hooks and finalize the recorder and set `_process` to None.

gather_info(*frame*, *event*, *arg*)
Gather information for the record.

class pikos.monitors.focused_function_memory_monitor.**FocusedFunctionMemoryMonitor**(**arguments*, ***keywords*)

Bases: pikos.monitors.focused_function_mixin.FocusedFunctionMixin, pikos.monitors.function_memory_monitor.FunctionMemoryMonitor

Record process memory on python function events.

The class hooks on the setprofile function to receive function events and record while inside the provided functions the current process memory when they happen.

Public

functions [FunctionSet] A set of function or method objects inside which recording will take place.

__init__(**arguments*, ***keywords*)
Initialize the monitoring class.

Parameters

- ***arguments** (*list*) – The list of arguments required by the base monitor. They will be passed on the super class of the mixing
- ****keywords** (*dict*) – Dictionary of keyword arguments. The *functions* keyword if defined should be a list of function or method objects inside which recording will take place.

---

class pikos.monitors.line_monitor.**LineMonitor**(*recorder*, *record_type=None*)
Bases: pikos.monitors.monitor.Monitor

Record python line events.

The class hooks on the settrace function to receive trace events and record when a line of code is about to be executed.

__init__(*recorder*, *record_type=None*)
Initialize the monitoring class.

Parameters **recorder** (*object*) – A subclass of `AbstractRecorder` or a class that implements the same interface to handle the values to be recorded.

record_type: class object A class object to be used for records. Default is `LineMonitor`

enable()
Enable the monitor.

The first time the method is called (the context is entered) it will set the settrace hook and initialize the recorder.

disable()
Disable the monitor.

The last time the method is called (the context is exited) it will unset the settrace hook and finalize the recorder.

on_line_event (*frame*, *why*, *arg*)
Record the current line trace event.

Called on trace events and when they refer to line traces, it will retrieve the necessary information from the *frame*, create a `LineRecord` and send it to the recorder.

gather_info (*frame*)
Gather information into a tuple.

class pikos.monitors.focused_line_monitor.**FocusedLineMonitor**(**arguments*, ***keywords*)

Bases: pikos.monitors.focused_line_mixin.FocusedLineMixin, pikos.monitors.line_monitor.LineMonitor

Record python line events.

The class hooks on the settrace function to receive trace events and record when a line of code is about to be executed. The events are recorded only when the interpreter is working inside the functions that are provided in the *functions* attribute.

__init__ (**arguments*, ***keywords*)
Initialize the monitoring class.

Parameters

- ***arguments** (*list*) – The list of arguments required by the base monitor. They will be passed on the super class of the mixing
- ****keywords** (*dict*) – Dictionary of keyword arguments. The *functions* keyword if defined should be a list of function or method objects inside which recording will take place.

---

class pikos.monitors.line_memory_monitor.**LineMemoryMonitor**(*recorder*, *record_type=None*)

Bases: pikos.monitors.line_monitor.LineMonitor

Record process memory on python line events.

The class hooks on the settrace function to receive trace events and record the current process memory when a line of code is about to be executed.

__init__ (*recorder*, *record_type=None*)
Initialize the monitoring class.

Parameters **recorder** (*object*) – A subclass of `AbstractRecorder` or a class that implements the same interface to handle the values to be recorded.

record_type: class object A class object to be used for records. Default is `LineMemoryMonitor`

enable()
Enable the monitor.

The first time the method is called (the context is entered) it will initialize the Process class, set the settrace hooks and initialize the recorder.

disable()
Disable the monitor.

The last time the method is called (the context is exited) it will unset the settrace hooks and finalize the recorder and set `_process` to None.

gather_info (*frame*)
Gather memory information for the line.

class pikos.monitors.focused_line_memory_monitor.**FocusedLineMemoryMonitor**(**arguments*, ***keywords*)

Bases: pikos.monitors.focused_line_mixin.FocusedLineMixin, pikos.monitors.line_memory_monitor.LineMemoryMonitor

Record process memory on python function events.

The class hooks on the settrace function to receive trace events and record the current process memory when a line of code is about to be executed. The events are recorded only when the interpreter is working inside the functions that are provided in the *functions* attribute.

__init__(**arguments*, ***keywords*)

Initialize the monitoring class.

Parameters

- ***arguments** (*list*) – The list of arguments required by the base monitor. They will be passed on the super class of the mixing
- ****keywords** (*dict*) – Dictionary of keyword arguments. The *functions* keyword if defined should be a list of function or method objects inside which recording will take place.

---

class pikos.monitors.focused_function_mixin.**FocusedFunctionMixin**(**arguments*, ***keywords*)

Bases: object

Mixing class to support recording python function events *focused* on a set of functions.

The method is used along a function event based monitor. It mainly overrides the on_function_event method to only record events when the interpreter is working inside one of predefined functions.

Public

functions [FunctionSet] A set of function or method objects inside which recording will take place.

__init__(**arguments*, ***keywords*)

Initialize the monitoring class.

Parameters

- ***arguments** (*list*) – The list of arguments required by the base monitor. They will be passed on the super class of the mixing
- ****keywords** (*dict*) – Dictionary of keyword arguments. The *functions* keyword if defined should be a list of function or method objects inside which recording will take place.

on_function_event(*frame*, *event*, *arg*)

Record the function event if we are inside one of the functions.

attach(*instance*, **args*, ***kwards*)

Attach (i.e. wrap) the monitor to the decorated function.

This method supports decorating functions with and without keyword arguments.

Parameters

- **instance** (*object*) – The monitor instance to attach.
- ***args** (*list*) – The list of arguments passed to the decorator.

- ****kwargs** (*dict*) – The dictionary of keyword arguments passed to the decorator.

Returns **fn** (*callable*) – Depending on the usage of the decorator (with or without arguments). The return callable is an instance of: - MonitorAttach, if the decorator is used without arguments. - FocusedMonitorAttach, if the decorator is used with arguments.

Raises **TypeError** – Raised if the monitor cannot be attached to the function.

class pikos.monitors.focused_line_mixin.**FocusedLineMixin**(**arguments*, ***keywords*)
Bases: object

Mixing class to support recording python line events *focused* on a set of functions.

The method is used along a line event based monitor. It mainly overrides the *on_line_event* method to only record events when the interpreter is working inside the predefined functions.

Public

functions [FunctionSet] A set of function or method objects inside which recording will take place.

__init__(**arguments*, ***keywords*)
Initialize the monitoring class.

Parameters

- ***arguments** (*list*) – The list of arguments required by the base monitor. They will be passed on the super class of the mixing
- ****keywords** (*dict*) – Dictionary of keyword arguments. The *functions* keyword if defined should be a list of function or method objects inside which recording will take place.

on_line_event(*frame*, *why*, *arg*)
Record the line event if we are inside the functions.

attach(*instance*, **args*, ***kwards*)
Attach (i.e. wrap) the monitor to the decorated function.

This method supports decorating functions with and without keyword arguments.

Parameters

- **instance** (*object*) – The monitor instance to attach.
- ***args** (*list*) – The list of arguments passed to the decorator.
- ****kwargs** (*dict*) – The dictionary of keyword arguments passed to the decorator.

Returns **fn** (*callable*) – Depending on the usage of the decorator (with or without arguments). The return callable is an instance of: - MonitorAttach, if the decorator is used without arguments. - FocusedMonitorAttach, if the decorator is used with arguments.

Raises **TypeError** – Raised if the monitor cannot be attached to the function.

Records

class pikos.monitors.records.**FunctionRecord**
Bases: pikos.monitors.records.FunctionRecord

The record tuple for function events.

Field	Description
index	The current index of the record.
type	The type of the event (see Python trace method).
function	The name of the function.
lineNo	The line number when the function is defined.
filename	The filename where the function is defined.

header = u'{:<8} {:<11} {:<30} {:<5} {}'

line = u'{:<8} {:<11} {:<30} {:<5} {}'

---

class pikos.monitors.records.**LineRecord**

Bases: pikos.monitors.records.LineRecord

The record for line trace events.

Field	Description
index	The current index of the record.
function	The name of the function.
lineNo	The line number when the function is defined.
line	The line that is going to be executed.
filename	The filename where the function is defined.

header = u'{:<12} {:<50} {:<7} {} – {}'

line = u'{:<12} {:<50} {:<7} {} – {}'

---

class pikos.monitors.records.**FunctionMemoryRecord**

Bases: pikos.monitors.records.FunctionMemoryRecord

The record tuple for memory usage on function events.

Field	Description
index	The current index of the record.
type	The type of the event (see Python trace method).
function	The name of the function.
RSS	The resident memory counter.
VMS	The virtual memory counter.
lineNo	The line number when the function is defined.
filename	The filename where the function is defined.

header = u'{:<8} | {:<11} | {:<12} | {:<15} | {:<15} | {:>6} | {}'

line = u'{:>8} | {:<11} | {:<12} | {:>15} | {:>15} | {:>6} | {}'

---

class pikos.monitors.records.**LineMemoryRecord**

Bases: pikos.monitors.records.LineMemoryRecord

The record tuple for memory usage on line events.

Field	Description
index	The current index of the record.
function	The name of the function
RSS	The resident memory counter.
VMS	The virtual memory counter.
lineNo	The line number when the function is defined.
filename	The filename where the function is defined.

header = u'{:^12} | {:^30} | {:^7} | {:^15} | {:^15} | {} {}'

line = u'{:<12} | {:<30} | {:<7} | {:>15} | {:>15} | {} {}'

Recorders

class pikos.recorders.abstract_recorder.**AbstractRecorder**
Bases: object

Abstract recorder class.

A recorder is reposnible for storing the record data that are provided by the monitor or profiler. The records are expected to be nametuple-like classes.

prepare(*record*)
Perform any setup required before the recorder is used.

Parameters **record** (*NamedTuple*) – The record class that is going to be used.

finalize()
Perform any tasks to finalize and clean up when the recording has completed.

record(*data*)
Record a measurement.

Parameters **data** (*NamedTuple*) – An instance of the record class that is going to be used.

---

class pikos.recorders.csv_recorder.**CSVRecorder**(*stream*, *filter_=None*, ***csv_kwargs*)
Bases: pikos.recorders.abstract_recorder.AbstractRecorder

The CSV Recorder is a simple text based recorder that records the tuple of values using a scv writer.

Private

_filter [callable] Used to check if the set *record* should be *recorded*. The function accepts a tuple of the *record* values and return True is the input sould be recored.

_writer [csv.writer] The *writer* object is owned by the CSVRecorder and exports the record values according to the configured dialect.

_ready [bool] Singify that the Recorder is ready to accept data.

__init__(*stream*, *filter_=None*, ***csv_kwargs*)
Class initialization.

Parameters

- **stream** (*file*) – A *file*-like object to use for output.
- **filter_** (*callable*) – A callable function that accepts a data tuple and returns True if the input sould be recorded.
- ****csv_kwargs** – Key word arguments to be passed to the *cvs.writer*.

prepare(*record*)
Write the header in the csv file the first time it is called.

finalize()
Finalize the recorder.

A do nothing method.

Raises **RecorderError** – Raised if the method is called without the recorder been ready to accept data.

record(*data*)
Record the data entry when the filter function returns True.

Parameters **values** (*NamedTuple*) – The record entry.

Raises **RecorderError** – Raised if the method is called without the recorder been ready to accept data.

---

class pikos.recorders.csv_file_recorder.**CSVFileRecorder**(*filename*, *filter_=None*, ***csv_kwargs*)
Bases: pikos.recorders.csv_recorder.CSVRecorder

A CSVRecorder that creates the stream (i.e. file) for the records.

Private

_filter [callable] Used to check if the set *record* should be *recorded*. The function accepts a tuple of the *record* values and return True is the input sould be recored.

_writer [csv.writer] The *writer* object is owned by the CSVRecorder and exports the record values according to the configured dialect.

_ready [bool] Singify that the Recorder is ready to accept data.

_filename [string] The name and path of the file to be used for output.

_file [file] The file object where records are stored.

__init__(*filename*, *filter_=None*, ***csv_kwargs*)
Class initialization.

Parameters

- **filename** (*string*) – The file path to use.
- **filter_** (*callable*) – A callable function that accepts a data tuple and returns True if the input sould be recorded. Default is None.
- ****csv_kwargs** – Key word arguments to be passed to the *cvs.writer*.

prepare(*record*)
Open the csv file and write the header in the csv file.

finalize()
Finalize the recorder.

Raises **RecorderError** – Raised if the method is called without the recorder been ready to accept data.

---

class pikos.recorders.list_recorder.**ListRecorder**(*filter_=None*)
Bases: pikos.recorders.abstract_recorder.AbstractRecorder

The ListRecorder is simple recorder that records the tuple of values in memory as a list.

Public

records [list] List of records. The Recorder assumes that the record method is provided with a tuple and accumulates all the records in a list.

Private

_filter [callable] Used to check if the data entry should be recorded. The function accepts a namedtuple record and return True is the input sould be recored.

__init__(*filter_=None*)
Class initialization.

Parameters **filter_** (*callable*) – A callable function to filter out the data entries that are going to be recorded.

prepare(*record*)
Prepare the recorder to accept data.

Note: nothing to do for the ListRecorder.

finalize()
Finalize the recorder.

Note: nothing to do for the ListRecorder.

ready
Is the recorder ready to accept data?

record(*data*)
Record the data entry when the filter function returns True.

Parameters **data** (*NamedTuple*) – The record entry.

---

class pikos.recorders.text_stream_recorder.**TextStreamRecorder**(*text_stream*, *filter_=None*, *formatted=False*, *auto_flush=False*)
Bases: pikos.recorders.abstract_recorder.AbstractRecorder

The TextStreamRecorder is simple recorder that formats and writes the records directly to a stream.

Private

_stream [TextIOBase] A text stream what supports the TextIOBase interface. The Recorder will write the values as a single line.

_filter [callable] Used to check if the set *record* should be *recorded*. The function accepts a tuple of the *record* values and return True is the input should be recorded.

_template [str] A string (using the *Format Specification Mini-Language*) to format the set of values in a line. It is constructed when the *prepare* method is called.

_auto_flush [bool] A bool to enable/disable automatic flushing of the string after each record process.

_ready [bool] Signify that the Recorder is ready to accept data.

__init__(*text_stream*, *filter_=None*, *formatted=False*, *auto_flush=False*)
Class initialization.

Parameters

- **text_stream** (*TextIOBase*) – A text stream what supports the TextIOBase interface.
- **filter_** (*callable*) – A callable function that accepts a data tuple and returns True if the input sould be recorded.
- **formatted** (*Bool*) – Use the predefined formatting in the records. Default value is false.
- **auto_flush** (*Bool*) – When set the stream buffer is always flushed after each record process. Default value is False.

prepare(*record*)
Prepare the recorder to accept data.

Parameters **data** (*NamedTuple*) – An example record to prepare the recorder and write the header to the stream.

finalize()
Finalize the recorder

A do nothing method.

Raises **RecorderError** – Raised if the method is called without the recorder been ready to accept data.

record(*data*)
Rerord the data entry when the filter function returns True.

Parameters **data** (*NamedTuple*) – The record entry.

Raises **RecorderError** – Raised if the method is called without the recorder been ready to accept data.

Note: Given the value of `_auto_flush` the recorder will flush the stream buffers after each record.

---

class pikos.recorders.text_file_recorder.**TextFileRecorder**(*filename*, *filter_=None*, *formatted=False*, *auto_flush=False*)
Bases: pikos.recorders.text_stream_recorder.TextStreamRecorder

The TextStreamRecorder that creates the file for the records.

Private

_stream [TextIOBase] A text stream what supports the TextIOBase interface. The Recorder will write the values as a single line.

_filter [callable] Used to check if the set *record* should be *recorded*. The function accepts a tuple of the *record* values and return True is the input should be recorded.

_template [str] A string (using the *Format Specification Mini-Language*) to format the set of values in a line. It is constructed when the *prepare* method is called.

_auto_flush [bool] A bool to enable/disable automatic flushing of the string after each record process.

_ready [bool] Signify that the Recorder is ready to accept data.

_filename [string] The name and path of the file to be used for output.

__init__ (*filename*, *filter_=None*, *formatted=False*, *auto_flush=False*)
Class initialization.

Parameters

- **filename** (*string*) – The file path to use.
- **filter_** (*callable*) – A callable function that accepts a data tuple and returns True if the input sould be recorded. Default is None.
- **formatted** (*Bool*) – Use the predefined formatting in the records. Default value is false.
- **auto_flush** (*Bool*) – When set the stream buffer is always flushed after each record process. Default value is False.

prepare (*record*)
Open the file and write the header.

finalize ()
Finalize the recorder.

Raises **RecorderError** – Raised if the method is called without the recorder been ready to accept data.

---

class pikos.recorders.zeromq_recorder.**ZeroMQRecorder** (*zmq_host='127.0.0.1'*, *zmq_port=9001*, *filter_=None*, *wait_for_ready=True*, ***kwargs*)
Bases: pikos.recorders.abstract_recorder.AbstractRecorder

The ZeroMQ Recorder is a recorder that publishes each set of values on a 0MQ publish socket.

Private

_filter [callable] Used to check if the set *record* should be *recored*. The function accepts a tuple of the *record* values and return True is the input sould be recored.

_ready [bool] Singify that the Recorder is ready to accept data. Please use the Recorder.ready property

__init__ (*zmq_host='127.0.0.1'*, *zmq_port=9001*, *filter_=None*, *wait_for_ready=True*, ***kwargs*)
Class initialization.

Parameters **filter_** (*callable*) – A callable function that accepts a data tuple and returns True if the input sould be recorded.

prepare (*record*)
Write the header in the csv file the first time it is called.

finalize ()
Signal that recording has ended.

ready
Is the recorder ready to accept data?

record (*record*)
Rerord entry onlty when the filter function returns True.

filters

class pikos.filters.on_value.**OnValue**(*field*, **args*)
Bases: object

A record filter that returns True if record has a specific value.

field = str
The field to check for change.

values
A list of values to use for the filtering.

Note: This filter only works with nametuple like records.

__init__(*field*, **args*)
Initialize the filter class.

Parameters

- **field** (*str*) – The field to check for change
- ***args** – A list of values to look for.

__call__(*record*)
Check for the value in the field.

---

class pikos.filters.on_change.**OnChange**(*field*)
Bases: object

A record filter that checks if the record field has changed.

A copy of the field value is stored in the object and compared against new values. On value changed the object returns True.

field = str
The field to check for change.

previous
Holds the value of the field.

Note:

•Filters and recorders can be shared between monitors. The filter however is not aware of ownership so use with care when sharing the same instance.

•This filter only works with nametuple like records.

__init__(*field*)
Initialize the filter class.

Parameters **field** (*str*) – The field to check for change

__call__(*record*)
Check if the field in the new record has changed.

External monitors

class pikos.external.python_cprofiler.**PythonCProfiler**(**args*, ***kwrds*)
Bases: cProfile.Profile, pikos.monitors.monitor.Monitor

The normal python Profile subclassed and adapted to work with the pikos Monitor decorator.

The class fully supports the Monitor decorator for functions and generators but does not support recorders.

Note: Due to the function wrapping a small overhead is expected especially if the decorated function is recursive calls. The wrapper function and the __enter__ and __exit__ methods of the context manager might also appear in the list of functions that have been called.

---

class pikos.external.yappi_profiler.**YappiProfiler**(*recorder=None*, *builtins=False*)
Bases: pikos.monitors.monitor.Monitor

A pikos compatible profiler class using the yappi library.

Private

_buildins [bool] Boolean to enable.disable profiling of the buildins.

The class partially supports the Monitor decorator for functions (not generators) but does not support recorders.

Note: The class mirrors the module interface. Please refer to the online documentation of the yappi module for information and usage of the interface (http://code.google.com/p/yappi/).

---

start(*builtins=None*)

stop()

enum_stats(*fenum*)

enum_thread_stats()

get_stats(**args*, ***kwrds*)

print_stats(**args*, ***kwrds*)

clear_stats()

is_running()

clock_type()

---

pikos.external.line_profiler.**LineProfiler**
alias of <bound method Mock.__name__ of

CHAPTER 5

Indices and tables

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

5.1 Usage

The main component in the pikos toolset is the *Monitor*. A monitor creates a number of records during the execution of the code which are passed on the recorder to be stored into memory or file.

5.1.1 In code

Monitors can be used programmatically in a number of ways.

1. Enabled/Disabled using the corresponding functions:

```
from pikos.api import screen
from pikos.monitors.api import FunctionMonitor

monitor = Monitor(recorder=screen())
monitor.enable()

# monitored code
#

monitor.disable()
```

2. A monitor instance can be used as a context manager:

```
from pikos.api import screen
from pikos.monitors.api import FunctionMonitor

monitor = Monitor(recorder=screen())

with monitor:
    # monitored code
    #
    pass
```

3. With the use of the *attach* method a monitor becomes a decorator:

```
from pikos.api import screen
from pikos.monitors.api import FunctionMonitor

monitor = Monitor(recorder=screen())

@monitor.attach
def monitored_function():
    # monitored code
    #
    pass
```

4. Finally the `pikos.api` module provides easy to use decorator factories for the standard monitors. The factories can optionally accept a recorder and dictate if a focused monitor should be used:

```
from pikos.api import function_monitor, csv_file

@function_monitor(recorder=csv_file(), focused=True)
def monitored_function():
    # monitored code
    #
    pass
```

5.1.2 Command line

The standard pikos monitors can be also used throught a command prompt tool, *pikos-run*:

```
usage: pikos-run [-h] [-o OUTPUT] [--buffered] [--recording {screen,text,csv}]
                 [--focused-on FOCUSED_ON]
                 {functions,line_memory,lines,function_memory} script

Execute the python script inside the pikos monitor context.

positional arguments:
  {functions,line_memory,lines,function_memory}
                        The monitor to use
  script                The script to run.

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        Output results to a file
  --buffered            Use a buffered stream.
  --recording {screen,text,csv}
                        Select the type of recording to use.
  --focused-on FOCUSED_ON
                        Provide the module path(s) of the method where
                        recording will be focused. Comma separated list of
                        importable functions
```

5.1.3 Example

Given the code bellow:

```
""" Mandelbrot Set example

Code from the Tentative Numpy Tutorial
```

```
url: http://wiki.scipy.org/Tentative_NumPy_Tutorial/Mandelbrot_Set_Example

"""

from numpy import *
import pylab


def mandelbrot(h, w, maxit=20):
    '''Returns an image of the Mandelbrot fractal of size (h,w).
    '''
    y,x = ogrid[-1.4:1.4:h*1j, -2:0.8:w*1j]
    c = x+y*1j
    z = c
    divtime = maxit + zeros(z.shape, dtype=int)

    for i in xrange(maxit):
        z = z**2 + c
        diverge = z*conj(z) > 2**2            # who is diverging
        div_now = diverge & (divtime==maxit)  # who is diverging now
        divtime[div_now] = i                  # note when
        z[diverge] = 2                        # avoid diverging too much

    return divtime


if __name__ == '__main__':
    pylab.imshow(mandelbrot(400,400))
    pylab.show()
```

Running:

```
pikos-run line_memory examples/mandelbrot_set_example.py --recording csv --focused-on=mandelbrot
```

from the root directory will run the *mandelbrot* example and record the memory usage on function entry and exit while inside the *mandelbrot* method. The monitoring information will be recorded in csv format in the *monitor_records.csv* (default filename).

CSV Sample

```
index,function,lineNo,RSS,VMS,line,filename
0,mandelbrot,16,64679936,382787584,"    y,x = ogrid[-1.4:1.4:h*1j, -2:0.8:w*1j]",examples/mandelbrot_
1,mandelbrot,17,64962560,383229952,    c = x+y*1j,examples/mandelbrot_set_example.py
2,mandelbrot,18,67678208,386056192,    z = c,examples/mandelbrot_set_example.py
3,mandelbrot,19,67817472,386056192,"    divtime = maxit + zeros(z.shape, dtype=int)",examples/mandelb
4,mandelbrot,21,69103616,387338240,    for i in xrange(maxit):,examples/mandelbrot_set_example.py
5,mandelbrot,22,69103616,387338240,        z = z**2 + c,examples/mandelbrot_set_example.py
6,mandelbrot,23,71671808,389902336,        diverge = z*conj(z) > 2**2            # who is diverging,ex
7,mandelbrot,24,76263424,394760192,        div_now = diverge & (divtime==maxit)  # who is diverging no
8,mandelbrot,25,76529664,394760192,        divtime[div_now] = i                  # note when,examples/
9,mandelbrot,26,76529664,394760192,        z[diverge] = 2                        # avoid diverging too
10,mandelbrot,21,76537856,394760192,    for i in xrange(maxit):,examples/mandelbrot_set_example.py
11,mandelbrot,22,76537856,394760192,        z = z**2 + c,examples/mandelbrot_set_example.py
12,mandelbrot,23,74452992,392675328,        diverge = z*conj(z) > 2**2            # who is diverging,e
13,mandelbrot,24,76881920,395235328,        div_now = diverge & (divtime==maxit)  # who is diverging n
14,mandelbrot,25,77012992,395235328,        divtime[div_now] = i                  # note when,examples
15,mandelbrot,26,77012992,395235328,        z[diverge] = 2                        # avoid diverging to
```

```
16,mandelbrot,21,77012992,395235328,    for i in xrange(maxit):,examples/mandelbrot_set_example.py
17,mandelbrot,22,77012992,395235328,        z = z**2 + c,examples/mandelbrot_set_example.py
18,mandelbrot,23,79441920,397795328,        diverge = z*conj(z) > 2**2            # who is diverging,e
19,mandelbrot,24,79572992,397795328,        div_now = diverge & (divtime==maxit)  # who is diverging n
```

The first column is the record index, followed by the function name and the line number where (just before execution) the RSS, VMS memory counters for the python process are recorded. The last two column contains the python line of the function.

Note: This record type is specific to the `LineMemoryMonitor`.

Plot Data

loading the csv file into ipython we can plot the graph of record index to RSS memory usage of the python process while executing the *mandelbrot* function:

```
In [1]: import numpy

In [2]: import pylab

In [3]: data = numpy.loadtxt('monitor_records.csv',usecols=[0, 3], delimiter=',', skiprows=1)

In [4]: pylab.plot(data[:, 0], data[:, 1], drawstyle='steps')

In [5]: pylab.show()
```

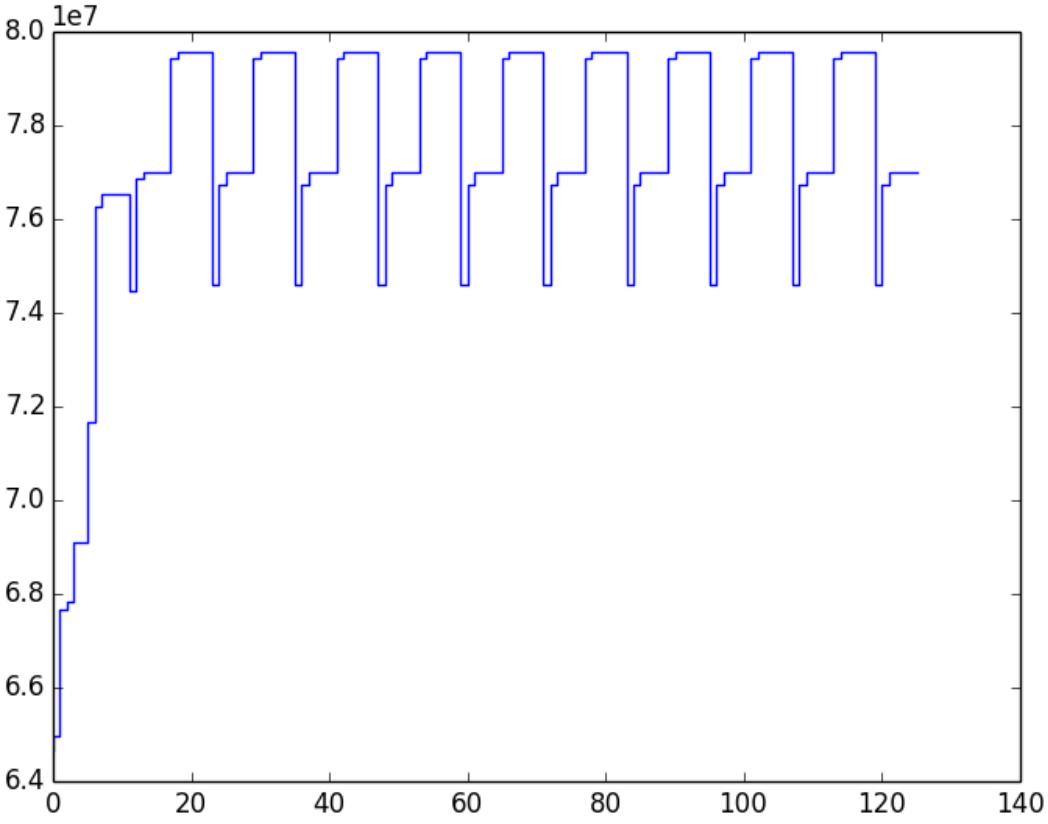

Figure 5.1: The plot of record index vs RSS memory usage (in bytes) of the python process when running the *mandelbrot* function.

Symbols

A

C

D

E

F

G

H

I

L

M

O

P

R

S

T

V

Y

Z