async_gui Documentation

Release 0.1.1

Roman Haritonov

December 23, 2016

Contents

CHAPTER 1

Usage

1.1 Installation

Install with pip or easy_install:

```
$ pip install --upgrade async_gui
```

or download latest version from GitHub:

```
$ git clone https://github.com/reclosedev/async_gui.git
$ cd async_gui
$ python setup.py install
```

To run tests:

```
$ python setup.py test
```

1.2 First steps

First create *Engine* instance corresponding to your GUI toolkit (see *Supported GUI toolkits*):

```
from async_gui.tasks import Task
from async_gui.toolkits.pyqt import PyQtEngine

engine = PyQtEngine()
```

It contains decorator *@engine.async* which allows you to write asynchronous code in serial way without callbacks. Example button click handler:

```
@engine.async
def on_button_click(self, *args):
    self.status_label.setText("Downloading image...")
    # Run single task in separate thread
    image_data = yield Task(self.load_url,
                            "http://www.google.com/images/srpr/logo4w.png")
    pixmap = QtGui.QPixmap.fromImage(QtGui.QImage.fromData(image_data))
    self.image_label.setPixmap(pixmap)
    self.status_label.setText("Downloading pages...")
    urls = ['http://www.google.com',
            'http://www.yandex.ru',
            'http://www.python.org']
```

```
    # Run multiple task simultaneously in thread pool
    pages = yield [Task(self.load_url, url) for url in urls]
    self.status_label.setText("Done")
    avg_size = sum(map(len, pages)) / len(pages)
    self.result_label.setText("Average page size: %s" % avg_size)
```

Before starting GUI mainloop set *Engine.main_app* property, so it can update GUI while executing tasks:

```
app = QtGui.QApplication(sys.argv)
engine.main_app = app
...
sys.exit(app.exec_())
```

Note: It's not necessarily for PyQT4/PySide and Gtk.

1.3 Tasks in threads

Tasks yielded from generator (`on_button_click()`) executed in thread pool, but GUI updates done in the GUI thread.

If exception occur during execution of task callable, it will be thrown in GUI thread and can be catched:

```
try:
    result = yield Task(func_that_may_raise_exception)
except SomeException as e:
    # show message, log it, etc...
```

If generator yields list of *Task*, they are executed simultaneously. If you want to control number of simultaneous tasks use *async_gui.tasks.MultiTask*. Example with 4 workers:

```
from async_gui.tasks import MultiTask
...
results = yield MultiTask([Task(do_something, i) for i in range(10)],
                          max_workers=4)
```

If one of the tasks raises exception then it will be raised in GUI thread and results will not be returned. To skip errors use `skip_errors=True` argument for *async_gui.tasks.MultiTask*. Only successful results will be returned

```
results = yield MultiTask([Task(do_something, i) for i in range(10)],
                          skip_errors=True)
```

New in version 0.2.0: `unordered` argument for MultiTask

Results from *MultiTask* returned when all tasks finished, but if you want to process results one by one as it's ready, you can set `unordered` argument to True, results will be generator, which yields finished tasks:

```
tasks = [Task(func, i) for i in range(10)]
results_gen = yield MultiTask(tasks, unordered=True)
for result in results_gen:
    show_in_gui(result)
```

See Unordered MultiTask example

1.4 Tasks in processes

For CPU-bound applications you can use ability to run tasks in pool of processes. Just change *Task* to *ProcessTask*

```
from async_gui.tasks import ProcessTask
...
result = yield ProcessTask(function_for_process)
```

Note: `function_for_process` should be importable. Also see multiprocessing documentation for limitations and Programming guidelines.

If generator yields list of *ProcessTask*, they are executed in pool of processes. Default number of simultaneous tasks equals to number of CPU cores. If you want to control number of simultaneous tasks use *async_gui.tasks.MultiProcessTask*.

See full example in *examples* directory. There is also examples of Tkinter, Gtk and WxPython applications.

1.5 Tasks in greenlets

You can also run tasks in gevent greenlets. See *GTask* and *MultiGTask*

Note: You need to apply gevent monkey-patch yourself, see Gevent docs

1.6 Returning result

In Python < 3.3 you can't return result from generator. But if you need to, you can use *return_result()* function which is shortcut for raising *ReturnResult* exception.

CHAPTER 2

Supported GUI toolkits

Currently supported gui toolkits:

Toolkit	Engine class
PyQt4	`async_gui.toolkits.pyqt.PyQtEngine`
PySide	`async_gui.toolkits.pyside.PySideEngine`
Tkinter	`async_gui.toolkits.tk.TkEngine`
WxPython	`async_gui.toolkits.wx.WxEngine`
GTK	`async_gui.toolkits.pygtk.GtkEngine`

You can simply add support for another toolkit by subclassing *`async_gui.engine.Engine`* and overriding *`Engine.update_gui()`*

For more information see *API*, examples, sources and tests.

CHAPTER 3

API

3.1 engine

Core functionality.

exception async_gui.engine.**ReturnResult**(*result*)
Exception Used to return result from generator

class async_gui.engine.**Engine**(*pool_timeout=0.02*)
Engine base class

After creating engine instance, set *main_app* property (not needed with PyQt/PySide)

Decorate generator with *@async* to execute tasks yielded from generator in separate executor and rest operations in GUI thread.

Subclasses should implement *update_gui()*.

Parameters **pool_timeout** – time in seconds which GUI can spend in a loop

main_app = None
main application instance

async(*func*)
Decorator for asynchronous generators.

Any Task, ProcessTask or GTask yielded from generator will be executed in separate thread, process or greenlet accordingly. For example gui application can has following button click handler:

```
engine = PyQtEngine()
...
@engine.async
def on_button_click():
    # do something in GUI thread
    data = yield Task(do_time_consuming_work, param)
    update_gui_with(data)  # in main GUI thread
```

If some task raises *ReturnResult*, it's value will be returned .. seealso:: *return_result()*

create_runner(*gen*)
Creates *Runner* instance

Parameters **gen** – generator which returns async tasks

Can be overridden if you want custom Runner

update_gui()
Allows GUI to process events

Should be overridden in subclass

class async_gui.engine.**Runner**(*engine*, *gen*)
Internal class that runs tasks returned by generator

Parameters

- **engine** – *Engine* instance
- **gen** – Generator which yields tasks

run()
Runs generator and executes tasks

async_gui.engine.**return_result**(*result*)
Allows to return result from generator

Internally it raises *ReturnResult* exception, so take in mind, that it can be catched in catch-all block

3.2 tasks

Contains task classes which when yilded from async generator will be executed in thread pool, or process pool

class async_gui.tasks.**Task**(*func*, **args*, ***kwargs*)
Represents single async operation.

Accepts callable and optionally its `args` and `kwargs`:

```
result = yield Task(time_consuming_operation, arg, some_kwarg=1)
```

executor_class
Executor class (from *concurrent.futures*) overridden in subclasses default is *ThreadPoolExecutor*

alias of `ThreadPoolExecutor`

max_workers = 1
Maximum number of workers, mainly used in MultiTask

class async_gui.tasks.**ProcessTask**(*func*, **args*, ***kwargs*)
Task executed in separate process pool

class async_gui.tasks.**MultiTask**(*tasks*, *max_workers=None*, *skip_errors=False*, *unordered=False*)
Tasks container, executes passed tasks simultaneously in ThreadPool

Parameters

- **tasks** – list/tuple/generator of tasks
- **max_workers** – number of simultaneous workers, default is number of tasks
- **skip_errors** – if True, tasks which raised exceptions will not be in resulting list/generator
- **unordered** – if True, result will be returned as generator, which yields task's results as it's ready.

wait(*executor*, *spawned_futures*, *timeout=None*)
Return True if all tasks done, False otherwise

class async_gui.tasks.**MultiProcessTask**(*tasks*, *max_workers=None*, *skip_errors=False*, ***kwargs*)

Tasks container, executes passed tasks simultaneously in ProcessPool

Same parameters as *MultiTask* but one is different:

Parameters **max_workers** – number of simultaneous workers, default is number of CPU cores

3.3 gevent_tasks

Tasks executing in gevent Pool

Note: You need to apply gevent monkey-patch yourself, see docs

class async_gui.gevent_tasks.**GTask**(*func*, **args*, ***kwargs*)

Task executed in *gevent* Pool

class async_gui.gevent_tasks.**MultiGTask**(*tasks*, *max_workers=None*, *skip_errors=False*, *unordered=False*)

Multiple tasks executed in *gevent* Pool simultaneously

Parameters

- **tasks** – list/tuple/generator of tasks
- **max_workers** – number of simultaneous workers, default is number of tasks
- **skip_errors** – if True, tasks which raised exceptions will not be in resulting list/generator
- **unordered** – if True, result will be returned as generator, which yields task's results as it's ready.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

a

A

C

E

G

M

P

R

T

U

W