Asynchronous features of plac

Author:	Michele Simionato
E-mail:	michele.simionato@gmail.com

Date:	June 2010
Download page:	http://pypi.python.org/pypi/plac
Project page:	http://micheles.googlecode.com/hg/plac/doc/plac.html
Installation:	`easy_install -U plac`
License:	BSD license
Requires:	Python 2.5+

Contents

Introduction

plac_ started out as a command-line arguments parser, but it quickly grown as a general purpose tool to write interpreters for command-line languages. In particular a number of facilities to write *interactive* interpreters where implemented. However, before release 0.6 plac_ did not provide any facility to address the issue of long running commands. The problem is that by default a plac_ interpreter blocks until a that command terminates: that makes the interactive experience quite painful for long running commands. An example is better than a thousand words, so consider the following fake importer:

```
import time
import plac

class Importer(object):
    "A fake importer with an import_file command"
    commands = ['import_file']
    def __init__(self, dsn):
        self.dsn = dsn
    def import_file(self, fname):
        "Import a file into the database"
        for n in range(10000):
            time.sleep(.01)
            if n % 100 == 99:
                yield 'Imported %d lines' % (n+1)
            yield # to keep the interface responsive

if __name__ == '__main__':
    plac.Interpreter(plac.call(Importer)).interact()
```

If you run the `import_file` commands, you will have to wait for 100 seconds before entering a new command:

```
$ python importer1.py dsn
A fake importer with an import_file command
i> import_file file1
Imported 100 lines
Imported 200 lines
Imported 300 lines
... <wait a couple of minutes>
Imported 10000 lines
```

Being unable to enter any command is quite annoying: in such situation one would like to run the long running commands in the background, to keep the interface responsive. plac_ provides all the ways to do it.

Threaded commands

The most familiar way to execute a task in the background (even if not necessarily the best way) is to run it into a separated thread. In our example it is sufficient to replace the line

```
commands = ['import_file']
```

with

```
thcommands = ['import_file']
```

to tell to the plac_ interpreter that the command `import_file` should be run into a separated thread. Here is an example session:

```
i> import_file file1
<ThreadedTask 1 [import_file file1] RUNNING>
```

The import task started in a separated thread. You can see the progress of the task by using the special command `_output`:

```
i> _output 1
<ThreadedTask 1 [import_file file1] RUNNING>
Imported 100 lines
Imported 200 lines
```

If you look after a while, you will get more lines of output:

```
i> _output 1
<ThreadedTask 1 [import_file file1] RUNNING>
Imported 100 lines
Imported 200 lines
Imported 300 lines
Imported 400 lines
```

If you look after a time long enough, the task will be finished:

```
i> _output 1 <ThreadedTask 1 [import_file file1] FINISHED>
```

You can launch many tasks one after the other:

```
i> import_file file2
<ThreadedTask 5 [import_file file2] RUNNING>
i> import_file file3
<ThreadedTask 6 [import_file file3] RUNNING>
```

```
i> _list
<ThreadedTask 5 [import_file file2] RUNNING>
<ThreadedTask 6 [import_file file3] RUNNING>
```

It is even possible to kill a task:

```
i> _kill 5
<ThreadedTask 5 [import_file file2] TOBEKILLED>
i> _output 5
<ThreadedTask 5 [import_file file2] KILLED>
```

You should notice that since it is impossible to kill a thread, the `_kill` commands actually works by setting the status of the task to `TOBEKILLED`. Internally the generator corresponding to the command is executed in the thread and the status is checked at each iteration: when the status become `TOBEKILLED` a `GeneratorExit` exception is raised and the thread terminates. This explain the importance of the empty `yield` in the `import_file` generator (line 15 of `importer1.py`): since we are resuming the generator every 10 milliseconds we check for the `TOBEKILLED` status 100 times per second (more than enough). If we did not add the `yield` the status would have been checked only once every 100 iterations, i.e. once per second, not enough to get a responsive interface.

Running commands as external processes

Threads are not loved much in the Python world and actually most people prefer to use processes instead. For this reason plac_ provides the option to execute long running commands as external processes. Unfortunately the current implementation only works in Unix-like operating systems (including Mac OS X) because it relies on fork via the multiprocessing_ module. To enable the feature in our example it is sufficient to replace the line

```
thcommands = ['import_file']
```

with

```
mpcommands = ['import_file'].
```

The user experience is exactly the same as with threads and you will not see any difference at the user interface level. Still, it is important to notice that using processes is quite different than using threads: in particular, when using processes you can only yield pickleable values and you cannot re-raise an exception first raised in a different process, because traceback objects are not pickleable.

Asynchronous mode

Command-line interfaces based on plac_ are usually blocking, i.e. the interpreter waits until it gets input from `stdin`. However plac_ also support an synchronous mode based on an event loop which is continuously running. You can submit commands to the event loop, and they are run one step at the time. For the approach to work you must make sure that the command yields back to the event loop often enough, otherwise the interface will block. In our example to enable the asynchronous mode it is sufficient to replace the line

```
mpcommands = ['import_file']
```

with

```
asyncommands = ['import_file'].
```

The user experience is quite similar to the threading and multiprocessing mode, with a single difference: the readline features are not available (the reason is that the plac_ event loop is based on a `select` call). However on Unix you can use rlwrap_ on top of plac_ and you will get command history even if not command completion.

The builtin event loop of plac_ is a toy: the asynchronous mode of plac_ is interesting if you are using a third party asynchronous framework, such as Twisted or the Tornado web server. In such cases you can use your framework event loop instead of the plac_ event loop. There is already a project (monocle_) providing glue code for three asynchronous frameworks (Twisted, Tornado and the standard library asyncore framework). I do not like to reinvent the wheel, therefore I wrote the plac_ eventloop to be compatible with the monocle_ event loop: that means that you can integrate plac_ with all the event loops wrapped by monocle_.

Here is an example (you need to install monocle_ first):

```
import plac
from monocle import asyncoreloop
from importer3 import Importer

if __name__ == '__main__':
    i = plac.Interpreter(plac.call(Importer))
    i.eventloop = asyncoreloop
    i.loop()
```

This code starts a server on port 2199: multiple clients can connect to it and run commands.