greenlet Documentation

Release 0.4.0

Armin Rigo, Christian Tismer

Dec 06, 2017

Contents

CHAPTER 1

Motivation

The “greenlet” package is a spin-off of Stackless, a version of CPython that supports micro-threads called “tasklets”. Tasklets run pseudo-concurrently (typically in a single or a few OS-level threads) and are synchronized with data exchanges on “channels”.

A “greenlet”, on the other hand, is a still more primitive notion of micro-thread with no implicit scheduling; coroutines, in other words. This is useful when you want to control exactly when your code runs. You can build custom scheduled micro-threads on top of greenlet; however, it seems that greenlets are useful on their own as a way to make advanced control flow structures. For example, we can recreate generators; the difference with Python’s own generators is that our generators can call nested functions and the nested functions can yield values too. (Additionally, you don’t need a “yield” keyword. See the example in `test/test_generator.py`).

Greenlets are provided as a C extension module for the regular unmodified interpreter.

1.1 Example

Let’s consider a system controlled by a terminal-like console, where the user types commands. Assume that the input comes character by character. In such a system, there will typically be a loop like the following one:

```
def process_commands(*args):
    while True:
        line = ''
        while not line.endswith('\n'):
            line += read_next_char()
        if line == 'quit\n':
            print("are you sure?")
            if read_next_char() != 'y':
                continue    # ignore the command
        process_command(line)
```

Now assume that you want to plug this program into a GUI. Most GUI toolkits are event-based. They will invoke a call-back for each character the user presses. [Replace “GUI” with “XML expat parser” if that rings more bells to you `:-)`] In this setting, it is difficult to implement the read_next_char() function needed by the code above. We have two incompatible functions:

```
def event_keydown(key):
    ??

def read_next_char():
    ?? should wait for the next event_keydown() call
```

You might consider doing that with threads. Greenlets are an alternate solution that don't have the related locking and shutdown problems. You start the process_commands() function in its own, separate greenlet, and then you exchange the keypresses with it as follows:

```
def event_keydown(key):
         # jump into g_processor, sending it the key
    g_processor.switch(key)

def read_next_char():
        # g_self is g_processor in this simple example
    g_self = greenlet.getcurrent()
        # jump to the parent (main) greenlet, waiting for the next key
    next_char = g_self.parent.switch()
    return next_char

g_processor = greenlet(process_commands)
g_processor.switch(*args)   # input arguments to process_commands()

gui.mainloop()
```

In this example, the execution flow is: when read_next_char() is called, it is part of the g_processor greenlet, so when it switches to its parent greenlet, it resumes execution in the top-level main loop (the GUI). When the GUI calls event_keydown(), it switches to g_processor, which means that the execution jumps back wherever it was suspended in that greenlet – in this case, to the switch() instruction in read_next_char() – and the `key` argument in event_keydown() is passed as the return value of the switch() in read_next_char().

Note that read_next_char() will be suspended and resumed with its call stack preserved, so that it will itself return to different positions in process_commands() depending on where it was originally called from. This allows the logic of the program to be kept in a nice control-flow way; we don't have to completely rewrite process_commands() to turn it into a state machine.

CHAPTER 2

Usage

2.1 Introduction

A “greenlet” is a small independent pseudo-thread. Think about it as a small stack of frames; the outermost (bottom) frame is the initial function you called, and the innermost frame is the one in which the greenlet is currently paused. You work with greenlets by creating a number of such stacks and jumping execution between them. Jumps are never implicit: a greenlet must choose to jump to another greenlet, which will cause the former to suspend and the latter to resume where it was suspended. Jumping between greenlets is called “switching”.

When you create a greenlet, it gets an initially empty stack; when you first switch to it, it starts to run a specified function, which may call other functions, switch out of the greenlet, etc. When eventually the outermost function finishes its execution, the greenlet’s stack becomes empty again and the greenlet is “dead”. Greenlets can also die of an uncaught exception.

For example:

```python
from greenlet import greenlet

def test1():
    print(12)
    gr2.switch()
    print(34)

def test2():
    print(56)
    gr1.switch()
    print(78)

gr1 = greenlet(test1)
gr2 = greenlet(test2)
gr1.switch()
```

The last line jumps to test1, which prints 12, jumps to test2, prints 56, jumps back into test1, prints 34; and then test1 finishes and gr1 dies. At this point, the execution comes back to the original `gr1.switch()` call. Note that 78 is never printed.

2.2 Parents

Let's see where execution goes when a greenlet dies. Every greenlet has a "parent" greenlet. The parent greenlet is initially the one in which the greenlet was created (this can be changed at any time). The parent is where execution continues when a greenlet dies. This way, greenlets are organized in a tree. Top-level code that doesn't run in a user-created greenlet runs in the implicit "main" greenlet, which is the root of the tree.

In the above example, both gr1 and gr2 have the main greenlet as a parent. Whenever one of them dies, the execution comes back to "main".

Uncaught exceptions are propagated into the parent, too. For example, if the above test2() contained a typo, it would generate a NameError that would kill gr2, and the exception would go back directly into "main". The traceback would show test2, but not test1. Remember, switches are not calls, but transfer of execution between parallel "stack containers", and the "parent" defines which stack logically comes "below" the current one.

2.3 Instantiation

`greenlet.greenlet` is the greenlet type, which supports the following operations:

`greenlet(run=None, parent=None)` Create a new greenlet object (without running it). `run` is the callable to invoke, and `parent` is the parent greenlet, which defaults to the current greenlet.

`greenlet.getcurrent()` Returns the current greenlet (i.e. the one which called this function).

`greenlet.GreenletExit` This special exception does not propagate to the parent greenlet; it can be used to kill a single greenlet.

The `greenlet` type can be subclassed, too. A greenlet runs by calling its `run` attribute, which is normally set when the greenlet is created; but for subclasses it also makes sense to define a `run` method instead of giving a `run` argument to the constructor.

2.4 Switching

Switches between greenlets occur when the method switch() of a greenlet is called, in which case execution jumps to the greenlet whose switch() is called, or when a greenlet dies, in which case execution jumps to the parent greenlet. During a switch, an object or an exception is "sent" to the target greenlet; this can be used as a convenient way to pass information between greenlets. For example:

```
def test1(x, y):
    z = gr2.switch(x+y)
    print(z)

def test2(u):
    print(u)
    gr1.switch(42)

gr1 = greenlet(test1)
gr2 = greenlet(test2)
gr1.switch("hello", " world")
```

This prints "hello world" and 42, with the same order of execution as the previous example. Note that the arguments of test1() and test2() are not provided when the greenlet is created, but only the first time someone switches to it.

Here are the precise rules for sending objects around:

g.switch(*args, **kwargs) Switches execution to the greenlet `g`, sending it the given arguments. As a special case, if `g` did not start yet, then it will start to run now.

Dying greenlet If a greenlet's `run()` finishes, its return value is the object sent to its parent. If `run()` terminates with an exception, the exception is propagated to its parent (unless it is a `greenlet.GreenletExit` exception, in which case the exception object is caught and *returned* to the parent).

Apart from the cases described above, the target greenlet normally receives the object as the return value of the call to `switch()` in which it was previously suspended. Indeed, although a call to `switch()` does not return immediately, it will still return at some point in the future, when some other greenlet switches back. When this occurs, then execution resumes just after the `switch()` where it was suspended, and the `switch()` itself appears to return the object that was just sent. This means that `x = g.switch(y)` will send the object `y` to `g`, and will later put the (unrelated) object that some (unrelated) greenlet passes back to us into `x`.

Note that any attempt to switch to a dead greenlet actually goes to the dead greenlet's parent, or its parent's parent, and so on. (The final parent is the "main" greenlet, which is never dead.)

2.5 Methods and attributes of greenlets

g.switch(*args, **kwargs) Switches execution to the greenlet `g`. See above.

g.run The callable that `g` will run when it starts. After `g` started, this attribute no longer exists.

g.parent The parent greenlet. This is writeable, but it is not allowed to create cycles of parents.

g.gr_frame The current top frame, or None.

g.dead True if `g` is dead (i.e. it finished its execution).

bool(g) True if `g` is active, False if it is dead or not yet started.

g.throw([typ, [val, [tb]]]) Switches execution to the greenlet `g`, but immediately raises the given exception in `g`. If no argument is provided, the exception defaults to `greenlet.GreenletExit`. The normal exception propagation rules apply, as described above. Note that calling this method is almost equivalent to the following:

```
def raiser():
    raise typ, val, tb
g_raiser = greenlet(raiser, parent=g)
g_raiser.switch()
```

except that this trick does not work for the `greenlet.GreenletExit` exception, which would not propagate from `g_raiser` to `g`.

2.6 Greenlets and Python threads

Greenlets can be combined with Python threads; in this case, each thread contains an independent "main" greenlet with a tree of sub-greenlets. It is not possible to mix or switch between greenlets belonging to different threads.

2.7 Garbage-collecting live greenlets

If all the references to a greenlet object go away (including the references from the parent attribute of other greenlets), then there is no way to ever switch back to this greenlet. In this case, a GreenletExit exception is generated into the greenlet. This is the only case where a greenlet receives the execution asynchronously. This gives `try:finally:`

blocks a chance to clean up resources held by the greenlet. This feature also enables a programming style in which greenlets are infinite loops waiting for data and processing it. Such loops are automatically interrupted when the last reference to the greenlet goes away.

The greenlet is expected to either die or be resurrected by having a new reference to it stored somewhere; just catching and ignoring the GreenletExit is likely to lead to an infinite loop.

Greenlets do not participate in garbage collection; cycles involving data that is present in a greenlet's frames will not be detected. Storing references to other greenlets cyclically may lead to leaks.

2.8 Tracing support

Standard Python tracing and profiling doesn't work as expected when used with greenlet since stack and frame switching happens on the same Python thread. It is difficult to detect greenlet switching reliably with conventional methods, so to improve support for debugging, tracing and profiling greenlet based code there are new functions in the greenlet module:

`greenlet.gettrace()` Returns a previously set tracing function, or None.

`greenlet.settrace(callback)` Sets a new tracing function and returns a previous tracing function, or None. The callback is called on various events and is expected to have the following signature:

```python
def callback(event, args):
    if event == 'switch':
        origin, target = args
        # Handle a switch from origin to target.
        # Note that callback is running in the context of target
        # greenlet and any exceptions will be passed as if
        # target.throw() was used instead of a switch.
        return
    if event == 'throw':
        origin, target = args
        # Handle a throw from origin to target.
        # Note that callback is running in the context of target
        # greenlet and any exceptions will replace the original, as
        # if target.throw() was used with the replacing exception.
        return
```

For compatibility it is very important to unpack args tuple only when event is either `'switch'` or `'throw'` and not when `event` is potentially something else. This way API can be extended to new events similar to `sys.settrace()`.

CHAPTER 3

C API Reference

Greenlets can be created and manipulated from extension modules written in C or C++, or from applications that embed Python. The `greenlet.h` header is provided, and exposes the entire API available to pure Python modules.

3.1 Types

Type name	Python name
PyGreenlet	greenlet.greenlet

3.2 Exceptions

Type name	Python name
PyExc_GreenletError	greenlet.error
PyExc_GreenletExit	greenlet.GreenletExit

3.3 Reference

`PyGreenlet_Import()` A macro that imports the greenlet module and initializes the C API. This must be called once for each extension module that uses the greenlet C API.

`int PyGreenlet_Check(PyObject *p)` Macro that returns true if the argument is a PyGreenlet.

`int PyGreenlet_STARTED(PyGreenlet *g)` Macro that returns true if the greenlet `g` has started.

`int PyGreenlet_ACTIVE(PyGreenlet *g)` Macro that returns true if the greenlet `g` has started and has not died.

`PyGreenlet *PyGreenlet_GET_PARENT(PyGreenlet *g)` Macro that returns the parent greenlet of `g`.

int PyGreenlet_SetParent(PyGreenlet *g, PyGreenlet *nparent) Set the parent greenlet of `g`. Returns 0 for success. If -1 is returned, then `g` is not a pointer to a PyGreenlet, and an AttributeError will be raised.

PyGreenlet *PyGreenlet_GetCurrent(void) Returns the currently active greenlet object.

PyGreenlet *PyGreenlet_New(PyObject *run, PyObject *parent) Creates a new greenlet object with the callable `run` and parent `parent`. Both parameters are optional. If `run` is NULL, then the greenlet will be created, but will fail if switched in. If `parent` is NULL, the parent is automatically set to the current greenlet.

PyObject *PyGreenlet_Switch(PyGreenlet *g, PyObject *args, PyObject *kwargs)
Switches to the greenlet `g`. `args` and `kwargs` are optional and can be NULL. If `args` is NULL, an empty tuple is passed to the target greenlet. If kwargs is NULL, no keyword arguments are passed to the target greenlet. If arguments are specified, `args` should be a tuple and `kwargs` should be a dict.

PyObject *PyGreenlet_Throw(PyGreenlet *g, PyObject *typ, PyObject *val, PyObject *tb)
Switches to greenlet `g`, but immediately raise an exception of type `typ` with the value `val`, and optionally, the traceback object `tb`. `tb` can be NULL.

CHAPTER 4

Indices and tables

- search