

Don't call us, we'll call you: callback patterns and idioms in Python

http://www.aleax.it/bayp_cback.pdf



©2008 Google -- aleax@google.com

Audience levels for this talk

守

Shu

("Retain")

破

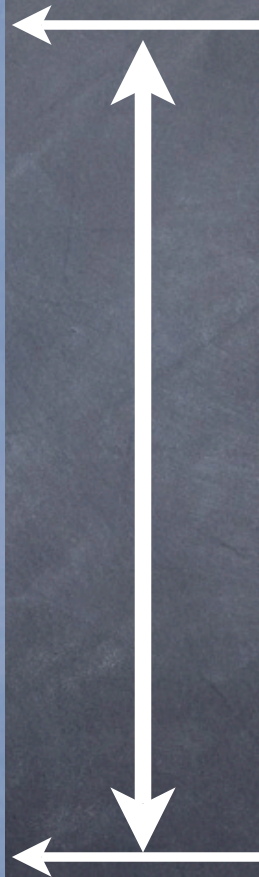
Ha

("Detach")

離

Ri

("Transcend")



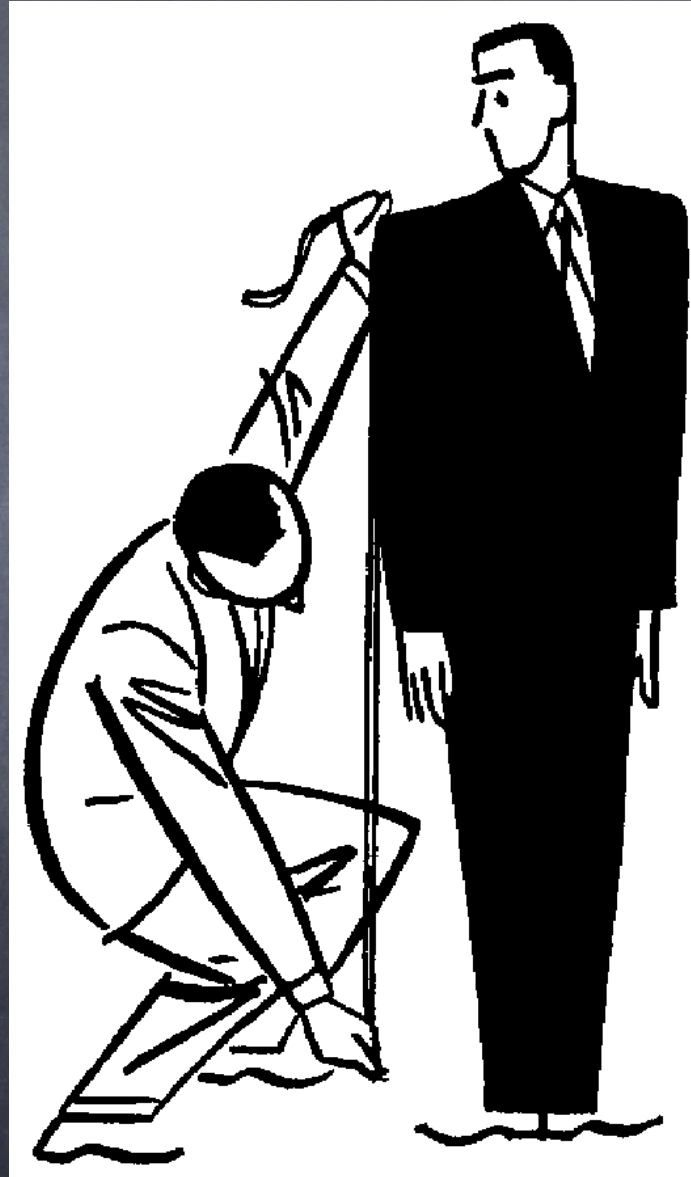
The "Callback" concept

- it's all about library/framework code that "calls back" into YOUR code
 - rather than the "traditional" (procedural) approach where YOU call code supplied as entry points by libraries &c
- it's "the Hollywood principle": "don't call us, we'll call you"
 - coinage: Richard E. Sweet, in "The Mesa Programming Environment", SigPLAN Notices, July 1985
- for: customization (flexibility) and "event-driven" architectures ("actual" events OR "structuring of control-flow" ["pseudo" events])

"Callback" implementation

- hand a callable over to "somebody"
- the "somebody" may store it "somewhere"
 - a container, an attribute, whatever
 - or even just keep it as a local variable
- and calls it "when appropriate"
 - when it needs some specific functionality (i.e., for customization)
 - or, when appropriate events "occur" (state changes, user actions, network or other I/O, timeouts, system events, ...) or "are made up" (structuring of control-flow)

Customization



Customizing sort (by key)

```
mylist.sort(key=str.toupper)
```

handily, speedily embodies the DSU pattern:

```
def DSU_sort(mylist, key):  
    aux = [ (key(v), j, v)  
            for j, v in enumerate(mylist)]  
    aux.sort()  
    mylist[:] = [v for k, j, v in aux]
```

Note that a little "workaround" is needed wrt the usual "call a method on each object" OO idiom...

OO customizing: the TM DP

- "Template Method" Design Pattern: perform the callbacks by "self delegation":

```
class TMparent(object):  
    ...self.somehook()...
```

- and customize by inheriting & overriding:

```
class TMchild(TMparent):  
    ...def somehook(self):...
```

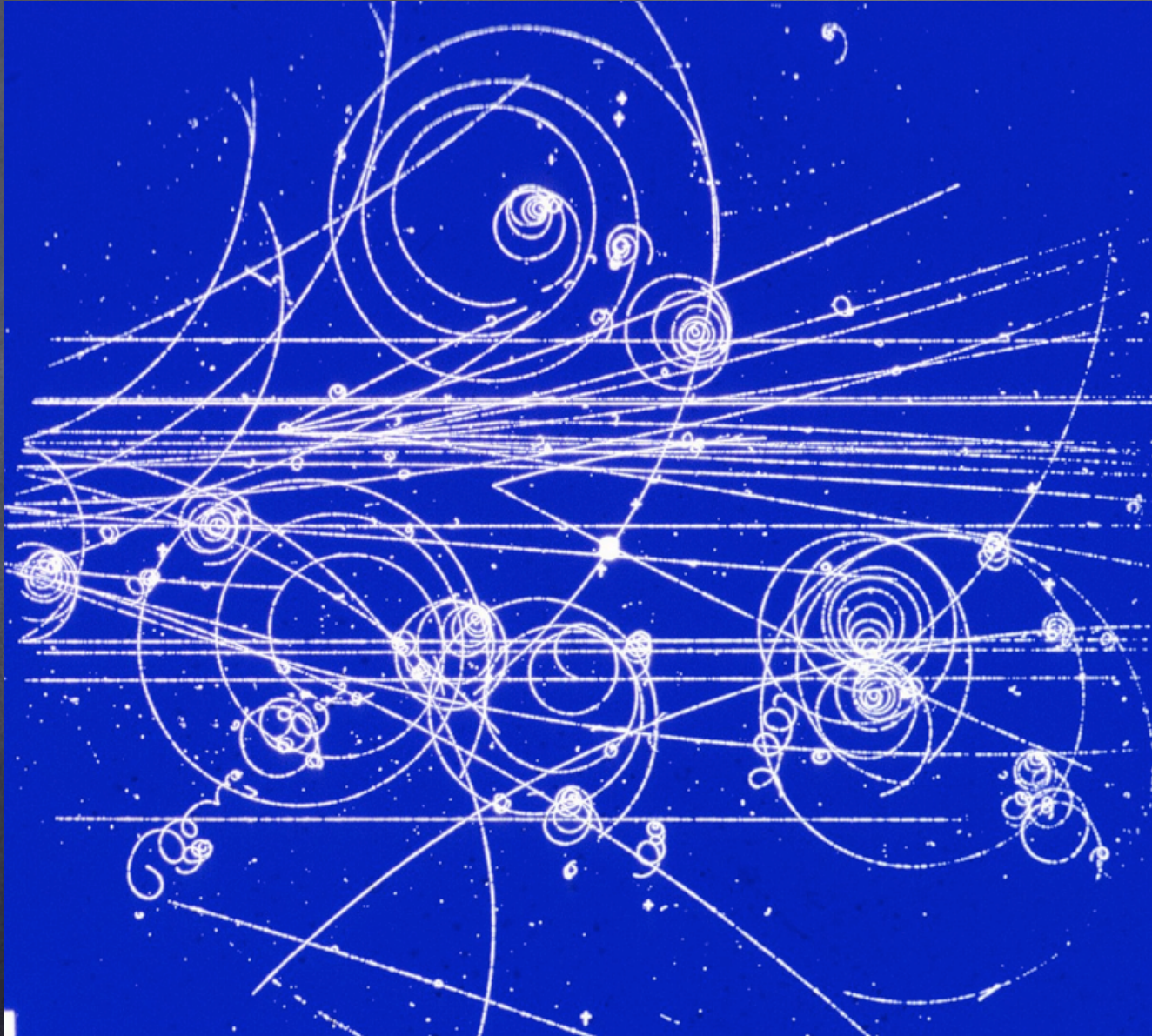
- handy, compact, sometimes a bit rigid

- <http://video.google.com/videoplay?docid=-5434189201555650834> and http://www.aleax.it/goo_pydp.pdf (49ff) for more

Customizing scheduling

- sched needs TWO callback functionalities:
 - what time is it right now?
 - wait (sleep) until time T
- the OO way (more structured) would be:
import time
s=sched(time)
- the FP way (more flexible) is instead:
s=sched(time.time, time.sleep)
- you might supply default callbacks, or not
- (Dependency Injection DP & variants)

Events



Kinds of "Event" callbacks

- Events "proper" ...:
 - GUI frameworks (mouse, keyboard, ...)
 - Observer/Observable design pattern
 - asynchronous (event-driven) I/O (net &c)
 - "system-event" callbacks
- Pseudo-events for "structuring" execution:
 - "event-driven" parsing (SAX &c)
 - "scheduled" callbacks (sched)
 - "concurrent" callbacks (threads &c)
 - timing and debugging (timeit, pdb, ...)

Events in GUI frameworks

- the most classic of event-driven fields
- e.g, consider Tkinter:
- elementary callbacks e.g. for buttons:
 - `b=Button(parent, text='boo!', command=...)`
- flexible, advanced callbacks and events:
 - `wgt.bind(event, handler)`
 - event: string describing the event (e.g. '`<Enter>`', '`<Leave>`', '`<Key>`', ...)
 - handler: callable taking Event argument (w. attributes `.widget`, `.x`, `.y`, `.type`, ...)
 - can also bind by class, all, root window...

The Observer DP

- a "target object" lets you add "observers"
 - could be simple callables, or objects
- when the target's state changes, it calls back to "let the observers know"
- design choices: "general" observers (callbacks on ANY state change), "specific" observers (callbacks on SPECIFIC state changes; level of specificity may vary), "grouped" observers (objects with >1 methods for kinds of state-change), ...

Callback issues

- what arguments are to be used on the call?
 - no arguments: simplest, a bit "rough"
 - in Observer: pass as argument the target object whose state just changed
 - lets 1 callable observe several targets
 - or: a "description" of the state changes
 - saves "round-trips" to obtain them
 - other: identifier or description of event
- but -- what about other arguments (related to the callable, not to the target/event)...?

Fixed args in callbacks

- `functools.partial(callable, *a, **kw)`
 - pre-bind any or all arguments
- however, note the difference...:
 - `x.setCbk(functools.partial(f, *a, **kw))`
 - vs
 - `x.setCbk(f, *a, **kw)`
 - ...having the set-callback itself accept (and pre-bind) arguments is far neater/handier
- `sombunall`¹ Python callback systems do that

¹: Robert Anton Wilson

Callback "dispatching"

- what if more than one callback is set for a single event (or, Observable target)?
 - remember and call the latest one only
 - simplest, roughest
 - or, remember and call them all
 - LIFO? FIFO? or...?
 - how do you `_remove_` a callback?
 - can one callback "preempt" others?
- can events (or state changes) be "grouped"?
 - use object w/methods instead of callable

Callbacks and Errors

- are "errors" events like any others?
- or are they best singled-out?
<http://www.python.org/pycon/papers/deferex/>
- Twisted Matrix's "Deferred" pattern: one Deferred object holds...
 - N "chained" callbacks for "successes" +
 - M "chained" callbacks for "errors"
 - each callback is held WITH opt *a, **kw
 - plus, argument for "event / error identification" (or, result of previous callback along the appropriate "chain")

System-events callbacks

- for various Python "system-events":
 - `atexit.register(callable, *a, **k)`
 - `oldhandler = signal.signal(signum, callable)`
 - `sys.displayhook`, `sys.excepthook`,
`sys.settrace(callable)`,
`sys.setprofile(callable)`
- some extension modules do that, too...:
 - `readline.set_startup_hook`,
`set_pre_input_hook`, `set_completer`

"Pseudo" events

- "events" can be a nice way to structure execution (control) flow
 - so in some cases "we make them up" (!) just to allow even-driven callbacks in otherwise non-obvious situations;-)
- parsing, scheduling, concurrency, timing, debugging, ...

Event-driven parsing

- e.g. SAX for XML
 - "events" are start and end of tags
 - handlers are responsible for keeping stack or other structure as needed
 - often not necessary to keep all...!
- at the other extreme: XML's DOM
- somewhere in-between: "pull DOM" ...
 - events as "stream" rather than callback
 - can "expand node" for DOMy subtrees

Scheduled callbacks

- standard library module sched
- `s = sched.Sched(timefunc, delayfunc)`
 - e.g, `Sched(time.time, time.sleep)`
- `evt = s.enter(delay, priority, callable, arg)`
 - or `s.enterabs(time, priority, callable, arg)`
 - may `s.cancel(evt)` later
- `s.run()` runs events until queue is empty (or an exception is raised in callable or delayfunc: it propagates but leaves `s` in stable state, `s.run` can be called again later)

"Concurrent" callbacks

- `threading.Thread(target=..,args=..,kwargs=..)`
 - call backs to `target(*args,**kwargs)`
 - at the `t.start()` event [or later...]
 - **in a separate thread** (the key point!-)
- `stacklet.tasklet(callable)`
 - calls back according to setup
 - when tasklet active and front-of-queue
 - channels, reactivation, rescheduling
- `processing.Process(...like threading.Thread...)`
- NWS' sleigh: `eachElem`, `eachWorker`

Timing and debugging

- `timeit.Timer(stmt, setup)`
 - `*string*` arguments to compile & execute
 - a dynamic-language twist on callback!-)
 - "event" for callback:
 - `setup`: once, before anything else
 - `stmt`: many times, for timing
- the `pdb` debugger module lets you use either strings or callables...:
 - `pdb.run` and `.runeval`: strings
 - `pdb.runcall`: callable, arguments

Q & A

http://www.a1eax.it/bayp_cback.pdf

