

Python Dependency Injection

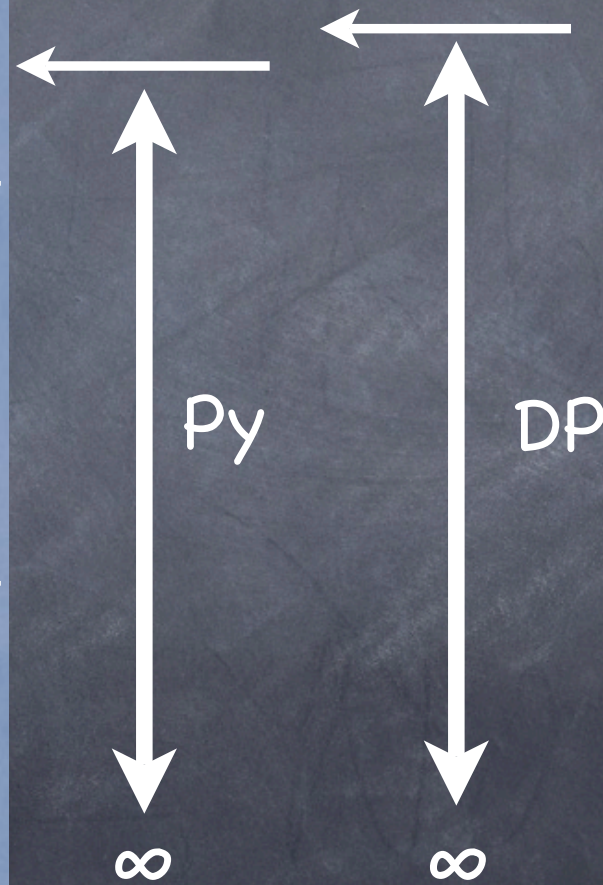
Alex Martelli (aleax@google.com)

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



Copyright ©2008, Google Inc

The "levels" of this talk



The novice goes astray and says,
"The Art failed me."

The master goes astray and says,
"I failed the Art."

Dependency Injection DP

- Name: "Dependency Injection"
- Forces: an object depends on other concrete objects which it instantiates (or accesses as singletons, ...)
 - we may want to control dependencies for all the usual good reasons
 - in particular, unit-testing may require mocking otherwise-concrete objects
- we'll see examples & alternative solutions throughout the rest of this talk

A simple scheduler

```
class ss(object):
    def __init__(self):
        self.i = itertools.count().next
        self.q = somemodule.PriorityQueue()
    def AddEvent(self, when, c, *a, **k):
        self.q.push((when, self.i(), c, a, k))
    def Run(self):
        while self.q:
            when, n, c, a, k = self.q.pop()
            time.sleep(when - time.time())
            c(*a, **k)
```


(A "side note")

```
class PriorityQueue(object):
    def __init__(self):
        self.l = []
    def __len__(self):
        return len(self.l)
    def push(self, obj):
        heapq.heappush(self.l, obj)
    def pop(self):
        return heapq.heappop(self.l)
```


Fine, but...

- ...how do you **test** ss without long waits?
- ...how do you **integrate** it with other subsystems' event loops/simulations?

The core issue is that ss "concretely depends" on some specific objects (here, callables `time.sleep` and `time.time`).

We'll discuss 3 approaches to solve this...:

1. the Template Method DP
2. "Monkey Patching"
3. the Dependency Injection DP

The Template Method DP

One classic answer ("Template Method" DP):

```
...  
when, n, c, a, k = self.q.pop()  
self.WaitFor(when)  
c(*a, **k)
```

```
...  
def WaitFor(self, when):  
    time.sleep(when - time.time())
```

(to customize: subclass `ss`, override `WaitFor`)

TM DP example

```
class sq(ss):
    def __init__(self):
        ss.__init__(self)
        ss.mtq = Queue.Queue()
    def WaitFor(self, when):
        try:
            while when > time.time():
                c, a, k = self.mtq.get(True,
                                       time.time() - when)
                c(*a, **k)
        except Queue.Empty:
            return
```


Some issues with TM

- inheritance gives strong, inflexible coupling
 - a customized-scheduler has complex, specialized extra logic
- far from ideal for either unit-testing or simulated-time system testing
 - e.g.: if another subsystem makes a scheduler, how does it know to make a test-scheduler instance vs a simple-scheduler one? (shades of recursion...)
- multiple integrations even harder than need be (but, there's no magic bullet for those!-)

Monkey-patching...

```
import ss
class faker(object): pass
fake = faker()
ss.time = fake
fake.sleep = ...
fake.time = ...
```



- 👁 extremely handy in emergencies, but...
- 👁 ...too often abused for NON-emergencies!
 - 👁 "gives dynamic languages a bad name"!-)
- 👁 subtle, hidden "communication" via secret, obscure pathways (explicit is better!-)

The general DI idea

```
class ss(object):  
    def __init__(self, tm=time.time,  
                 sl=time.sleep):  
        self.tm = tm  
        self.sl = sl  
    ...  
        self.sl(when - self.tm())
```

👁 a known use: standard library sched module!

DI makes it easy to mock

```
class faketime(object):  
    def __init__(self, t=0.0): self.t = t  
    def time(self): return self.t  
    def sleep(self, t): self.t += t
```

```
f = faketime()  
s = ss(f.time, f.sleep)  
...
```


DI/TM orthogonality

Not at all mutually exclusive...:

```
class ss(object):  
    def __init__(self, tm=time.time,  
                 sl=time.sleep):  
  
    ...  
    def WaitFor(self, when):  
        self.sl(when-self.tm())
```

then may use either injection, or subclassing and overriding, (or both!-), for testing, integration, &c

DI design-choice details

- inject by constructor (as shown)
 - with, or without, default dep. values?
 - ensure just-made instance is consistent
 - choose how "visible" to make the inject..
- inject by setter
 - automatic in Python (use non-__ names)
 - very flexible (sometimes too much;-)
- "inject by interface" (AKA "IoC type 1")
 - not very relevant to Python
- DI: by code or by config-file/flags?

DI and factories

```
class ts(object):
```

```
...
```

```
def Delegate(self, c, a, k):
```

```
    q = Queue.Queue()
```

```
    def f(): q.put(c(*a,**k))
```

```
    t = threading.Thread(target=f)
```


```
    t.start()
```

```
    return q
```

- 👁 each call to Delegate needs a new Queue and a new Thread; how do we DI these objects...?
- 👁 easy solution: inject **factories** for them!

DI and factories

```
class ts(object):
    def __init__(self, q=Queue.Queue,
                 t=threading.Thread):
        self.q = q
        self.t = t
    ...
    def Delegate(self, c, a, k):
        q = self.q()
        ...
        t = self.t(target=f)
```

👁 pretty obvious/trivial solution when each class is a factory for its instances, of course;-) 

Questions & Answers

Q?

A!