Modern Python Patterns and Idioms

http://www.aleax.it/pyconit15_mppi_en.pdf

©2015 Google -- aleax@google.com

Patterns vs Idioms (1)

- Patterns: a very general term
  - Architecture
  - Design
  - Development
  - Deployment
    - ...
  - can nevertheless be technology-specific
    - building architecture w/wooden beams
    - vs bricks
    - vs reinforced concrete

Patterns vs Idioms (2)

- Idioms: a rather specific term
  - in natural languages, "a phrase or fixed expression that has a figurative, or sometimes literal, meaning" (from Greek ἴδιος, "one's own")
  - a distinct style/character (music, art, &c)
  - in artificial languages (for programming, markup, configuration, &c), "a means of expressing a recurring construct" typical of the specific language

Today's hottest key patterns

- are mostly architectural ones
  - for distributed, scalable, reliable systems
  - farther away from coding than DPs are
- load balancing (inc. the elastic kind)
  - stateless or sticky
  - health checking, traffic splitting
    - canarying, A/B testing, ...
- microservices (w/REST and/or RPC APIs)
- caching (esp. the distributed kind)
  - oldie but goldie!-)

Load Balancing

- all load goes to a single system...
  - ...which balances it across the servers
  - always considering their "health"
    - ? considering their "load" ?
  - maybe adding servers (elastic)
    - ...and removing them when feasible...
  - health checking, traffic splitting
    - canarying, A/B testing, ...
  - ? track "state" (sessions)... ?
- unequal split x canarying and A/B testing

LB in Python

- txLoadBalancer 1.1.0
  - twisted, norm. stateless, at TCP level
  - scheduling configurabile (a tad of state)
  - ex http://pythondirector.sourceforge.net/
- http://zguide.zeromq.org/py:lbbroker
  - "example" (usable) in/for ZeroMQ
- ...

μ services

- instead of libraries (always in-process)
  - with HTTP+REST+JSON (or other RPC)
  - better if "clothed" with libraries
- explode network "scalability"
  - perhaps with internal load-balancing!-)
- easier to maintain, upgrade, test, ...
- can be multi-language (but Python...:-)
- only likely problem: overhead
- e.g: http://gilliam.github.io/

Distributed caching

- Beaker → dogpile.cache
- memcached

- problem #1, always: data freshness
- problem #2, sometimes: serialization format
- problem #3, sometimes: atomicity issues
- problem #4: overhead of distributed comms
  - vs a local cache alternative

PL progress swallows idioms

- …and sometimes patterns too (a fine line!)
  - in BAL/360: BALR r14, r15 … BR r14
    - subroutine-call as an idiom/pattern
  - in ARM: BL address … MOV pc, lr
    - dedicated link-register
  - in x86: explicit CALL/RET (using stack)
  - in HLL: explicit/implicit CALL/RETURN (stack somewhat hidden/parameters too)

Python swallows, too:-)

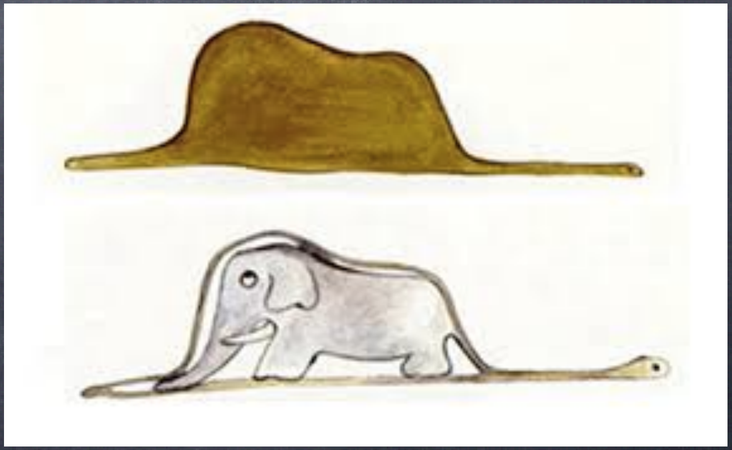
- once upon a time, DSU

```
decorated = [(f(x),x) for x in xs]
decorated.sort()
xs[:] = [x for _, x in decorated]
```

- nowadays, key=... most everywhere

```
xs.sort(key=f)
```

- ...but not quite everywhere, so DSU still worth knowing!-)



DSU and heapq

```
class keyed_heapq(object):
  def __init__(self, seq, key):
    self.h = [(key(s), s) for s in seq]
    heapq.heapify(self.h)
    self.key = key
  def __len__(self):
    return len(self.h)
  def push(self, x):
    decorated = (self.key(x), x)
    heapq.heappush(self.h, decorated)
  def pop(self):
    return heapq.heappop(self.h)[1]
  def peek(self):
    return self.h[0][1]
```

Python Containers

a long time ago
in a version far, far away,
there were only `list`, `dict`, and `tuple`…

what a journey it has been since!-)

Container Idioms

- `set` is a built-in: are you using it right?
    - and what about other "new" built-ins?
    - `frozenset, bytearray, memoryview, enumerate, reversed, buffer`...?
    - ever used built-in `object` idiomatically?
- `collections` has 5 containers obsoleting ("swallowing") many good old idioms
    - and 16 abc's -- even bigger potential!
    - (plus, more abc's -- the `numbers` module)

Your Honor, I `object`!

```
_sentinel = object()
def f(optional=_sentinel):
    if optional is _sentinel: ...
x = d.get(k, _sentinel)
```

Other Sentinel Pattern variants: +/- Infinity, EqualsAll, PredicateSatisfier decorator...:

```
def predicate_satisfier(predicate):
  def wrapper(x):
    if x is _sentinel: return True
    return predicate(x)
  return wrapper
```

Some Swallowed Idioms

```
d.setdefault(x, []).append(y)
```
...?

- nevermore! use, instead:

```
d = collections.defaultdict(list)
d[x].append(y)
```

- and for some idioms, generations passed:

```
if x in d: d[x] += 1
else: d[x] = 1

d[x] = 1 + d.get(x, 0)

d = collections.defaultdict(int)
d[x] += 1

d = collections.Counter(xs)
```

Not just dicts -- I/O, too...

```
while True:
    line = afile.readline()
    if line == '': break
    ...
```

- <u>nevermore</u>! **use:** `for line in afile:`

```
f = open(...)
try: ...
finally: f.close()
```

- <u>nevermore</u>! **use:** `with open(...) as f:`

collections.Counter

- not just a multiset (though mostly that:-)
  - as it can have zero/negative counts too!
- e.g: "items seen more often in `xs` than in `ys`"

```
a = collections.Counter(xs)
a.subtract(collections.Counter(ys))
return (a+collections.Counter()).keys()
```

- and don't forget `.elements` and `.most_common`!-)
- exercise: implement union, intersection, and symmetric difference, between counter multisets!

```
>>> xs = 'tanto va la gatta al lardo'
>>> ys = 'four score and seven years ago'
>>> a = collections.Counter(xs)
>>> a
Counter({'a':7, ' ':5, 't':4, 'l':3, 'o':2,
'd':1, 'g':1, 'n':1, 'r':1, 'v':1})
>>> a.subtract(collections.Counter(ys))
>>> a
Counter({'a': 4, 't': 4, 'l': 3, ' ': 0,
'd': 0, 'g': 0, 'v': 0, 'c': -1, 'f': -1,
'o': -1, 'n': -1, 'u': -1, 'y': -1, 'r': -2,
's': -3, 'e': -4})
>>> xx = a + collections.Counter()
>>> xx
Counter({'a': 4, 't': 4, 'l': 3})
>>> xx.keys()
['a', 'l', 't']
```

collections.deque

- not just "2-e queue" (though mostly that:-)
- as it can have <u>constrained length</u> too!
- perfect for a "ring buffer" ("last n items"):
  `d = collections.deque(iter, maxlen=n)`
  (`itertools.islice` can't support negative args!-)
- caveat for C++ers: general `d[x]` is O(N), not O(1)!

namedtuple

- namedtuple: mostly cosmetic, but, <u>readability counts!</u>
    - a "factory of container types"!

```
>>> Person =
collections.namedtuple('Person', 'name
phone email')
>>> x = Person('Alex', '555-5555',
'a@lex')
>>> x
Person(name='Alex', phone='555-5555',
email='a@lex')
>>> type(x)
<class '__main__.Person'>
```

OrderedDict

- OrderedDict: good, but *take care*!
  - bad anti-idiom alas often observed:

```
od = collections.OrderedDict(somedict)
```

  - see why it's totally useless...?
  - and similarly:

```
>>> collections.OrderedDict(b=1, a=2)
OrderedDict([('a', 2), ('b', 1)])
```

must be, instead:

```
od = OrderedDict([(b, 1), (a, 2)])
```

Do you *need* a container?

- Traditionally, you built up a list with interesting items, then looped over it for further processing

```
mylist = []
for rawitem in container:
    if interesting(rawitem):
        mylist.append(process(rawitem))
for x in mylist: ...
```

- then, list comprehensions appeared...:

```
mylist = [process(r) for r in container
          if interesting(r)]
for x in mylist: ...
```

Turns out you often *don't*!

- *generator expressions saved us a lot of memory...:

```
mygenex = (process(r) for r in container
           if interesting(r))
for x in mygenex: ...
```

- ...and the rush to iterators/generators was on!
- `itertools` raised it to a craze w/*performance*
  - & cool recipes@ `https://docs.python.org/2/library/itertools.html#recipes`
- generators also begat co-routines
  - w/`send` and `throw` methods, `yield` as an expr
  - then `yield from`, making `asyncio` possible

Iterator idioms

"First item > 25" (raise if no item is > 25)

```
fi = next(x for x in iter if x > 25)
```

Ditto, but, a sentinel of 0 rather than raising

```
fi = next((x for x in iter if x > 25), 0)
```

Is iterator empty?

```
_sentinel == next(iter, _sentinel)
```

How many items in iterator?

```
hmi = sum(1 for _ in iter)
```

Do remember each such idiom (itertools too!) advances/consumes the iterator! Cfr `itertools.tee` if appropriate...

Duck typing...?

- once upon a time...

```
def work(x):
    try: x + 0
    except TypeError: raise
```

...NEVER

```
if not isinstance(x, int):
    raise TypeError
```

isinstance rehabilitated

- ...thanks to Abstract Base Classes!
- so nowadays...:

```
if not isinstance(x, numbers.Number):
    raise TypeError
```

...GOOSE typing!

- and tomorrow...:
- (PEP 3107, 484, ...)

`def work(x: numbers.Number):` (SWAN typing?)

Note you can still easily get it wrong...

`def work(x: int):` (CUCKOO typing?-)

Q & A

http://www.aleax.it/pyconit15_mppi_en.pdf

? !