This talk discusses lightweight concurrency:

Python 2.2's generators enabled semi-coroutines as a mechanism for scheduling "weightless threads."

PEP 342 allowed (easier) injection of data into coroutines, and hence made event-driven programming easier.

3rd party libraries are built around coroutines, from older GTasklet and peak.events to the current Greenlet/gevent and Twisted Reactor.

For this talk I write a terrible program about a dozen times. This is the serial version:

```
def callout_test():
    vowel, consonant, other = map(
          Counter,['vowel','consonant','other'])
    for c in BOOK:          # Each letter is an event
        if c in 'aeiouAEIOU':
            vowel(c)
        elif c in letters:
            consonant(c)
        else:
            other(c)
    return LetterTypes(vowel, consonant, other)
```

Let us pretend that *Counter* is computationally expensive or uses slow resources like network, DB, filesystem, etc.:

```
class Counter(object):
    def __init__(self, name):
        self.name = name
        self.count = 0
    def __call__(self, *args, **kws):
        self.count += 1
    def __repr__(self):
        return repr(self.count)
```

This amazing fact has little to do with *this* talk:

```
% python switch-callout.py < complete_shakespeare
Count letters with callout to counter object
vowels=1439242 consonants=2365799 others=1785152
Ten runs in 44.092643 seconds
% pypy switch-callout.py < complete_shakespeare
Count letters with callout to counter object
vowels=1439242 consonants=2365799 others=1785152
Ten runs in 5.116044 seconds
```

(much less difference if counter is inlined; this is all calling cost)

Let us make a start at parallelizing the task:

```
def process_test():
    vowel, cons, other = [MCount(name,Value('i',0))
       for name in ('vowel','consonant','other')]
    for c in BOOK:
        if c in 'aeiouAEIOU':
            p = Process(target=vowel,args=(c,))
            p.start(); p.join()
        elif c in letters:
            p = Process(target=cons,args=(c,))
            p.start(); p.join()
        else:
            p = Process(target=other,args=(c,))
            p.start(); p.join()
    return LetterTypes(vowel, cons, other)
```

(more...)

First we create a few classified counter objects:

```
def process_test():
    vowel, cons, other = [MCount(name,Value('i',0))
       for name in ('vowel','consonant','other')]
    for c in BOOK:
        if c in 'aeiouAEIOU':
            p = Process(target=vowel,args=(c,))
            p.start(); p.join()
        elif c in letters:
            p = Process(target=cons,args=(c,))
            p.start(); p.join()
        else:
            p = Process(target=other,args=(c,))
            p.start(); p.join()
    return LetterTypes(vowel, cons, other)
```

For each event type, spawn a process of that type:

```
def process_test():
    vowel, cons, other = [MCount(name,Value('i',0))
       for name in ('vowel','consonant','other')]
    for c in BOOK:
        if c in 'aeiouAEIOU':
            p = Process(target=vowel,args=(c,))
            p.start(); p.join()
        elif c in letters:
            p = Process(target=cons,args=(c,))
            p.start(); p.join()
        else:
            p = Process(target=other,args=(c,))
            p.start(); p.join()
    return LetterTypes(vowel, cons, other)
```

The *MCount* class is a lot like *Counter*, just using *multiprocessing.Value* instead of a *int* attribute:

```
class MCount(object):
    def __init__(self, name, value):
        self.name = name
        self.count = value
    def __call__(self, *args, **kws):
        self.count.value += 1
    def __repr__(self):
        return repr(self.count.value)
```

Why this version is (probably) terrible:

```
for c in BOOK:
    if c in 'aeiouAEIOU':
        p = Process(target=vowel, args=(c,))
        p.start(); p.join()
    elif c in letters:
        p = Process(target=cons, args=(c,))
        p.start(); p.join()
    else:
        p = Process(target=other, args=(c,))
        p.start(); p.join()
```

Because every event launches a new process!

Why this version might be quite good:

```
for c in BOOK:
    if c in 'aeiouAEIOU':
        p = Process(target=vowel, args=(c,))
        p.start(); p.join()
    elif c in letters:
        # ...
```

Because every event launches a new process.

If handling an event takes minutes or hours (or even multiple seconds), process creation time doesn't matter and your OS takes care of multi-cores, process isolation, and preemptive switching.

Process pooling is the obvious solution to excessive process creation (where applicable).

```
def process_test():
    vowel, cons, other = [proctuple(name)
       for name in ('vowel','consonant','other')]

    for c in BOOK:
        if c in 'aeiouAEIOU': vowel.queue.put(c)
        elif c in letters:    cons.queue.put(c)
        else:                 other.queue.put(c)

    # Kill off the processes with a sentinel value
    for p in (vowel, cons, other):
        p.queue.put(KILL); p.proc.join()

    return LetterTypes(vowel, cons, other)
```

(just a few processes will be fed events; could use *multiprocessing.Pool*)

A *namedtuple* stores the process itself, a variation on a counter object, and the queue used to feed it:

```
ProcTuple = namedtuple(
    'ProcTuple', 'queue proc counter')
def proctuple(name):
    queue = Queue()
    counter = FeedCounter(name,Value('i',0),queue)
    proc = Process(target=counter)
    proc.start()
    return ProcTuple(queue, proc, counter)
```

(Queue, Value, and Process are conveniences from *multiprocessing*)

FeedCounter is a bit different from other counters since it handles events in a loop, until killed:

```
class FeedCounter(object):
    def __init__(self, name, value, queue):
        self.name, self.count, self.queue = (
             name, value, queue)
    def __call__(self, *args, **kws):
        while True:
            s = self.queue.get() # blocks here
            if c == KILL:        # sentinel value
                break
            self.count.value += 1
```

Since *multiprocessing* has an API based on *threading*, the threaded version is nearly identical.

```
def threadtuple(name):
    queue = Queue()
    counter = FeedCounter(name,Value('i',0),queue)
    thread = Thread(target=counter)
    thread.start()
    return ThreadTuple(queue, thread, counter)

def thread_test():
    vowel = threadtuple('vowel')
    consonant = threadtuple('consonant')
    other = threadtuple('other')
    # ... the rest is same as with processtuple()
```

(But the GIL means we get less use of multi-cores)

And now for something interesting . . .

```
class FeedCounter(object):
    def __call__(self, *args, **kws):
        while True:
            c = self.queue.get() # blocking is OK
            if c == KILL:        # sentinel value
                break
            self.count.value += 1
```

Squinting the right way, this look like a generator.

We can feed the queue, and the shared *Value* object will store the total number of things placed in the queue (and will terminate once signaled).

For comparison, think of a generator like:

```
def feed_counter(queue):
    count = 0
    while True:
        if queue.empty(): # Blocking not OK
            yield count; continue
        c = queue.get()
        if c == KILL:     # sentinel value
            break
        count += 1; yield count
```

We can call into this generator as we like; it will each time return the total number of things placed in the queue (and will terminate once signaled).

Be careful with unbound generators.

```
>>> fc = feed_counter()
>>> for cnt in fc: # Bad if no KILL in queue
...     do_something(cnt)
>>> # Could safely handle a few events at a time
>>> for cnt in islice(fc, 0, FLUSH_SIZE):
...     do_something(cnt)
>>> # Or explicitly pull .next() values
>>> while some_condition():
...     cnt = fc.next()
...     put_stuff_in(queue)
```

One might also check whether *cnt* is unchanged as a *break* condition, return a special *EMPTY* signal instead of *count*, etc.

Let us take a breath here…

Most of you see the punchline coming: We can often use generators as "weightless threads" in place of OS threads or processes. This is *almost* the same thing as "coroutines."

Pros: Creation, entry, and return are nearly free; code is simpler to write and understand; encourages asynchronous style of waiting for background events.

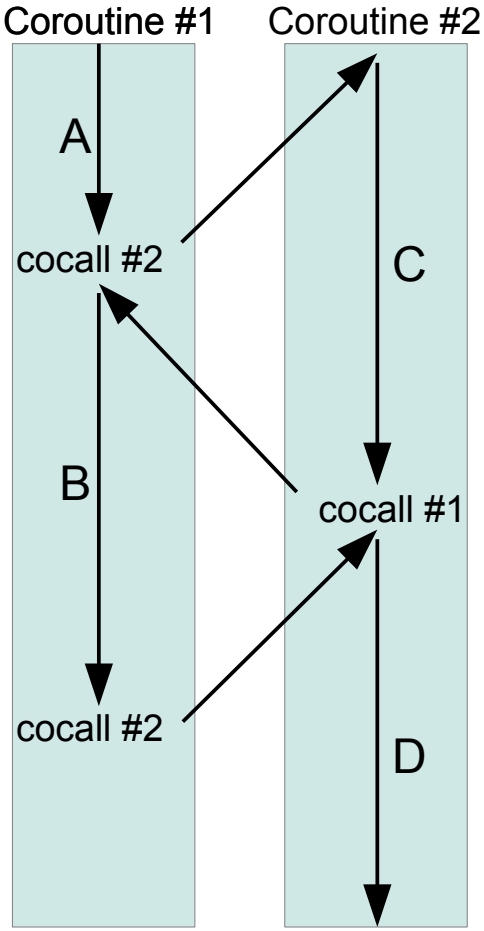
Cons: Generator "threads" are strictly single core; if your problem is computation-bound, asynchronous style is hard to manage; multi-tasking is cooperative.

Since Python 2.2 added generators, we have been able to write coroutines using a simple trampoline:

```
def routine1():
    print "A",; yield CR2
    print "B",; yield CR2

def routine2():
    print "C",; yield CR1
    print "D",; yield "DONE"

CR1, CR2 = routine1(), routine2()
current = CR1
while True:
    if current is "DONE": break
    current = current.next()
# Prints-> A C B D
```



You might think: *"Python didn't have coroutines until PEP 342 implemented them for 2.5."*

You would be wrong, despite the PEP 342 title "Coroutines via Enhanced Generators." You still need a trampoline nowadays, and "coroutines" are still really only semi-coroutines.

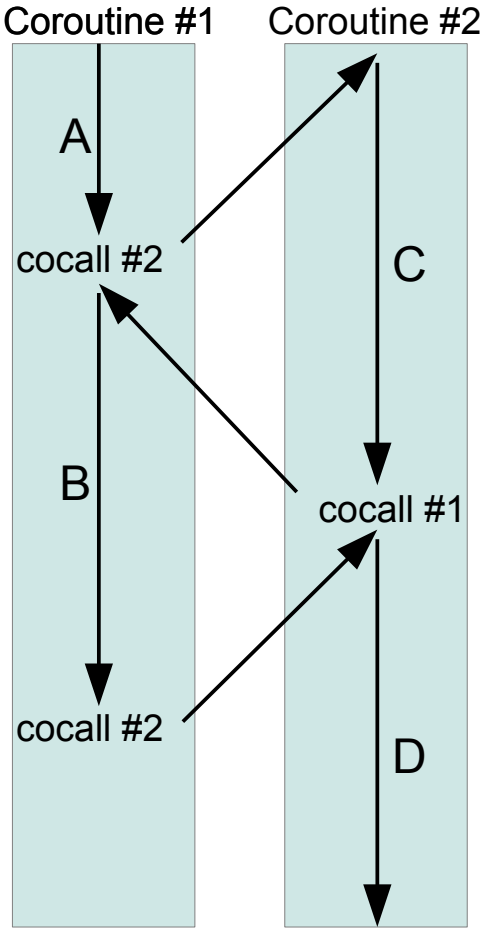
Quibbles aside, it is most certainly nicer to inject data into a generator than it is to pass around a shared data structure like a *Queue.Queue* or a mutable type that needs housekeeping.

What if a language similar to Python had direct coroutines (including data injection)?

```
def routine1(data):
    print "A", data,
    data = switch to routine2 send "a"
    print "B", data,
    data = switch to routine2 send "b"

def routine2(data):
    print "C", data,
    data = switch to routine1 send "c"
    print "D", data,

routine1('x') #-> A x C a B c D b
```



This *does* avoid trampolines.

In actual Python we still need a thin trampoline:

```
def routine1(data): # START routine passed data
    print "A",data,; data = yield (R2,'a')
    print "B",data,; data = yield (R2,'b')

def routine2():
    data = yield  # Others primed with throwaway
    print "C",data,; data = yield (R1,'c')
    print "D",data,

R1, R2 = routine1('x'), routine2()
(current,data), _ = R1.next(), R2.next()
while True:
    try: current, data = current.send(data)
    except StopIteration: break

# Prints -> A x C a B c D b
```

(more...)

Each *yield* selects a target and injects some data:

```
def routine1(data): # START routine passed data
    print "A",data,; data = yield (R2,'a')
    print "B",data,; data = yield (R2,'b')

def routine2():
    data = yield  # Others primed with throwaway
    print "C",data,; data = yield (R1,'c')
    print "D",data,

R1, R2 = routine1('x'), routine2()
(current,data), _ = R1.next(), R2.next()
while True:
    try: current, data = current.send(data)
    except StopIteration: break

# Prints -> A x C a B c D b
```

Coroutines need to be "primed" first:

```
def routine1(data): # START routine passed data
    print "A",data,; data = yield (R2,'a')
    print "B",data,; data = yield (R2,'b')

def routine2():
    data = yield  # Others primed with throwaway
    print "C",data,; data = yield (R1,'c')
    print "D",data,

R1, R2 = routine1('x'), routine2()
(current,data), _ = R1.next(), R2.next()
while True:
    try: current, data = current.send(data)
    except StopIteration: break

# Prints -> A x C a B c D b
```

(Priming with a decorator would be more elegant)

The trampoline loop itself is trivially simple:

```
def routine1(data): # START routine passed data
    print "A",data,; data = yield (R2,'a')
    print "B",data,; data = yield (R2,'b')

def routine2():
    data = yield  # Others primed with throwaway
    print "C",data,; data = yield (R1,'c')
    print "D",data,

R1, R2 = routine1('x'), routine2()
(current,data), _ = R1.next(), R2.next()
while True:
    try: current, data = current.send(data)
    except StopIteration: break

# Prints -> A x C a B c D b
```

In a more fleshed-out example, each generator is likely to loop indefinitely, and cocall based on conditional tests. E.g. (but still hypothetical):

```
def stage1():
    while True:
        data = get_nowait() # DB,socket,queue,etc.
        crunch(data)        # crunch() ignores None
        if data is None:
            yield ('newtask', None)
        elif isType(data, this):
            data = yield ('stage2', data)
        elif isType(data, that):
            data = yield ('stage3', data)
        else:
            break
```

Coroutines and "weightless threads" just need slightly different schedulers/trampolines:

```
def co_trampoline(generators, start, init_data):
    CRs = {}
    for g in generators:
        if g is start:
            coroutine = g(init_data)
            curr, data = coroutine.next()
        else:
            coroutine = g(); coroutine.next()
        CRs[coroutine.__name__] = coroutine

    while True:
        try: curr, data = CRs[curr].send(data)
        except StopIteration: break
```

(more...)

The start generator gets initialized with data, while the others are merely primed:

```
def co_trampoline(generators, start, init_data):
    CRs = {}
    for g in generators:
        if g is start:
            coroutine = g(init_data)
            curr, data = coroutine.next()
        else:
            coroutine = g(); coroutine.next()
        CRs[coroutine.__name__] = coroutine

    while True:
        try: curr, data = CRs[curr].send(data)
        except StopIteration: break
```

The trampoline loop itself is, as in the prior examples, essentially trivial:

```
def co_trampoline(generators, start, init_data):
    CRs = {}
    for g in generators:
        if g is start:
            coroutine = g(init_data)
            curr, data = coroutine.next()
        else:
            coroutine = g(); coroutine.next()
        CRs[coroutine.__name__] = coroutine
    while True:
        try: curr, data = CRs[curr].send(data)
        except StopIteration: break
```

A scheduler for weightless threads won't switch to a specific place, but will round-robin or the like:

```
def thread_scheduler(generators):
    thread_data = defaultdict(list)
    threads = [g() for g in generators()]
    while True:
        try:
            for t in threads:
                data = thread_data[t.__name__]
                consumer, data = t.send(data)
                thread_data[consumer].append(data)
        except StopIteration: break
```

Here consumer threads cannot be switched into directly, but once reached will get injected data.

Putting it all together into a "weightless" counter; first the loop itself, which looks much like earlier versions:

```
def weightless_test():
    vowel_t, consonant_t, other_t = [
          f() for f in [generator_counter]*3]
    for c in BOOK:
        if c in 'aeiouAEIOU':
            vowel = vowel_t.send(c)
        elif c in letters:
            consonant = consonant_t.send(c)
        else:
            other = other_t.send(c)
    return LetterTypes(vowel, consonant, other)
```

The counter is the simplest we have seen, but uses a decorator to "prime" the coroutine:

```
def coroutine(func):
    def start(*args, **kwargs):
        cr = func(*args, **kwargs)
        cr.next()
        return cr
    return start

@coroutine
def generator_counter():
    count = 0
    while True:
        c = yield count  # could actually use 'c'!
        count += 1
```

(The decorator lifted from a presentation by David Beazley)

The generator-injection style here is neither weightless threads nor coroutine switching. Each generator has no say in where to inject event data :

```
for c in BOOK:
    if c in 'aeiouAEIOU':
        vowel = vowel_t.send(c)
    elif c in letters:
        # ...
```

This is just standard event dispatch, really.

It is also free of overhead, even faster than the *callout_test()* version at the beginning of the talk (on CPython anyway, PyPy makes calling cheap).

PEP 380 (Syntax for Delegating to Subgenerator).

This feature will be in Python 3.3 (alpha 1 was released a couple days ago). It allows easier refactoring of generators by assigning work to subgenerators where:

```
for v in g:
    yield v
```

Will become expressible as:

```
yield from g
```

With PEP 380, the subgenerator also handles *.send()*, *.throw()* and *.close()* calls into the outside generator without extra scaffolding.

PEP 380: Simple refactoring example.

```
def pairs():
    for i in iter1:      # Some logic about 'i' here
        for j in iter2: # Complex stuff goes here
            yield i, j
```

```
def pairs():
    for i in iter1:      # Some logic about 'i' here
        yield from subloop(i)

def subloop(i):
    for j in iter2:      # Complex stuff goes here
        yield i, j
```

Third party stuff (greenlets, gevent, Twisted, etc.)

gevent is a coroutine-based Python networking library that uses *greenlet* to provide a high-level synchronous API on top of the *libevent* event loop

A "greenlet" is a micro-thread with no implicit scheduling; coroutines, in other words. You can build custom scheduled micro-threads on top of *greenlet*.

Twisted uses the *Deferred* object to manage the callback sequence. A series of functions are attached to the *deferred* to be called when the results of the asychronous request are available

Third party stuff (greenlets, gevent, Twisted, etc.)

Documentation on these libraries at:

- *http://twistedmatrix.com/trac/*
- *http://readthedocs.org/docs/greenlet/*
- *http://www.gevent.org/contents.html*

The APIs for these tools are different from the examples in this talk, but in ultimate effect are similar to the skeletons shown herein.

In particular, greenlets solve the issue addressed by PEP 380 in a somewhat different style.

Wrap-up / Questions?

Simple function calls, processes, process pools, and threads compared for event loops.

A fed process looks (a little bit) like a generator.

Coroutines as a mechanism of flow control and as event handlers. Data injection into coroutines.

Coroutines as a mechanism for weightless threads. Trampolines, schedulers and event loops.

PEP 380 and third party abstractions of generator-based threads.