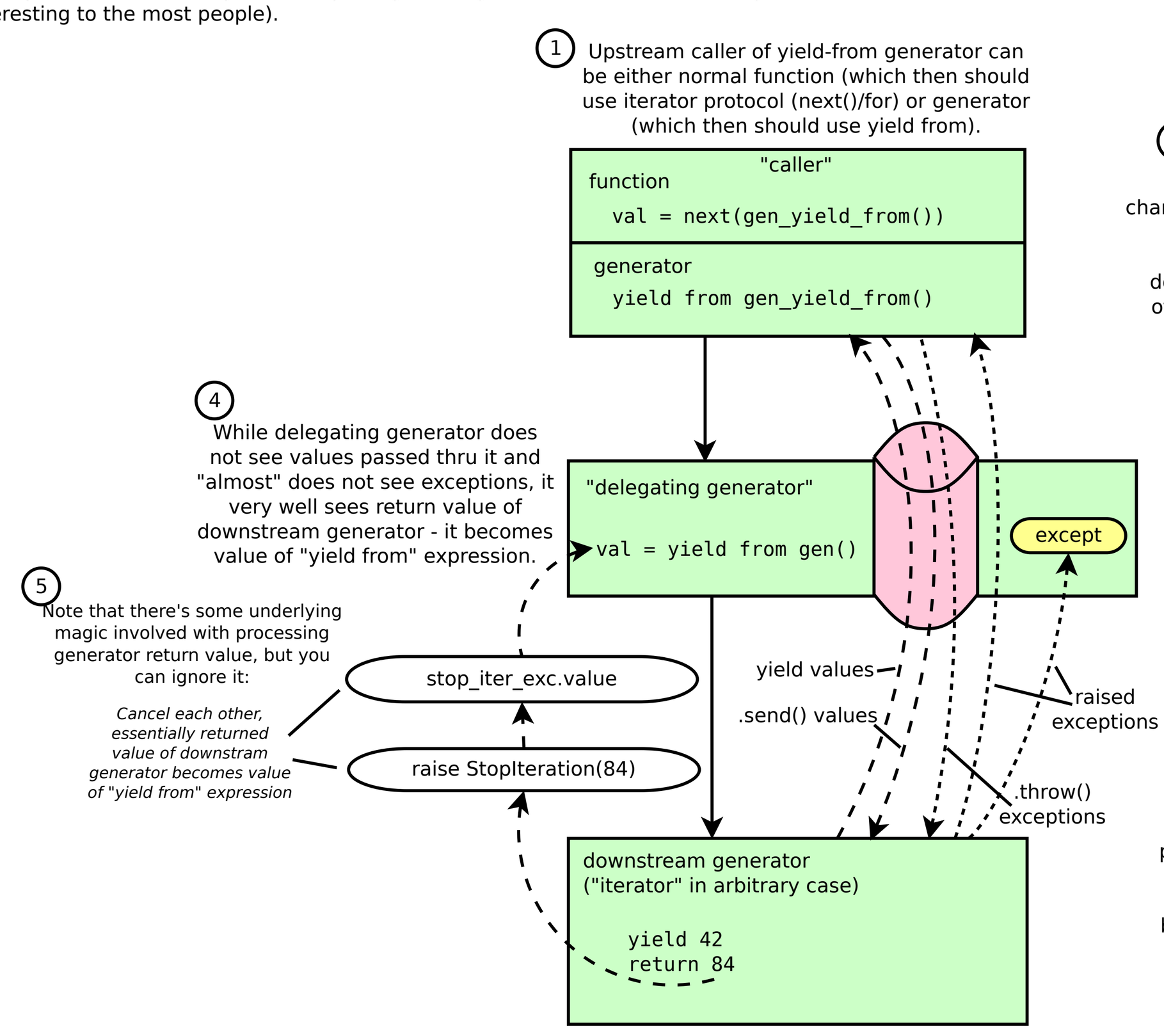
How Python 3.3 "yield from" construct works

Note to people not familiar with generators: By a definition, a generator is a function which uses yield or yield from. Text below always uses term "generator" to refer to such functions to be formally correct. But don't let that confuse you: generators are first of all just functions, like the ones you always used.

Note to Python language laweyrs who read PEP380: this diagram tries to use terminilogy consistent with PEP380, but a bit more self-describing, avoiding (implied) "hereafter called XXX". So, "upstream caller" is what PEP calls "caller", "delegating generator" is used just the same, and "downstream generator" corresponds to "iterator" (yes, to keep things manageable and focused, the diagram considers a case when downstream is a generator, which is the case interesting to the most people).



2 "yield from" establishes a transparent bidirectional channel for values between delegating generator's upstream caller (can be func or gen) and downstream generator (argument of "yield from"). Note that current iterator does not see any values yielded by downstream generator - again, they are transparently piped upstream. The channel also works in downstream direction for exceptions (as injected by .throw()).

3 Yes, you guessed it - magic unicorn pipe does not have any provisions for transporting ascending exceptions. Why is so? One obvious answer is that it is natural direction of exception propagation, already governed by Python semantics. In particular, exceptions will be transparently passed from downstream generator to upstream caller if delegating generator does not use try/except blocks. Or if except clause does not match exception raised. But with appropriate except clause, delegating iterator can easily intercept downstream exceptions, breaking free exception flow between downstream and upstream.

So, everything seems logical. But still, Greg, Guido, why? Why not let magic pipe do nice symetrical magic instead of black tangled exception-ridden magic? The biggest concern is that this inconsistency precludes *easy* optimization of "yield from" chains, and optimization now needs to be more complicated and timid. Was that the reason why Greg's "somewhat optimizing" patches did not make it to the releases, and "yield from" implementation in CPython 3.3&3.4 is non-optimized at all?

6 Note that as "yield from" establishes a transparent channel, there can be any number of "yield from" generators in a chain.

7 As Python uses duck typing, a downstream object can be something else than a generator:
1) It can be conventional iterable (e.g. a list). This is an easy case, it just yields values, does not use more advanced features like .send() or .throw().
2) Arbitrary object which implements __next__(), send(), throw(), close() methods (or their subset). In this case, these methods will be called as if it was a generator.

8 Summing up:

val = yield from func()

behaves just as

val = func()

In particular, it is *synchronous* call, returning all data in one big chunk at the end. The only difference of "yield from" is that it allows communication of downstream async code (data producers/consumers) with upstream async code (scheduler) transparently thru your func().

9 It's not enough to use only "yield from" to write well-behaving async apps. Read it again: when you write something like:

```
data = yield from sock.read(1000*1000*1000)
return "<b>" + data + "</b>"
```

- you are writing horrendous synchronous blocking code.

To write truly async code, one must use yield - sensibly.
Rough example:

```
def pump(ins, outs):
for chunk in gen(ins):
yield from outs.write(chunk)

def gen(ins):
yield "<b>"
yield from ins.read_in_chunks(1000*1000*1000)
yield "</b>"
```