Flavors of PyPy

CPython: You can have it in any color, provided it‘s green. No longer ☺

Choose Your Flavor and combine:

- Reference counting, Boehm, …
- Stackless support
- Thunk object space
- C, LLVM, JS
- More options to come

Variations on a theme

- Stackless Python on PyPy
  - Just a compilation option
  - Very minimalistic layer on top of coroutines
- Low-level coroutines
  - Create high-speed, non-blocking sockets
- App-level coroutines and more
  - Use whatever API you like
  - coroutines, greenlets, tasklets included

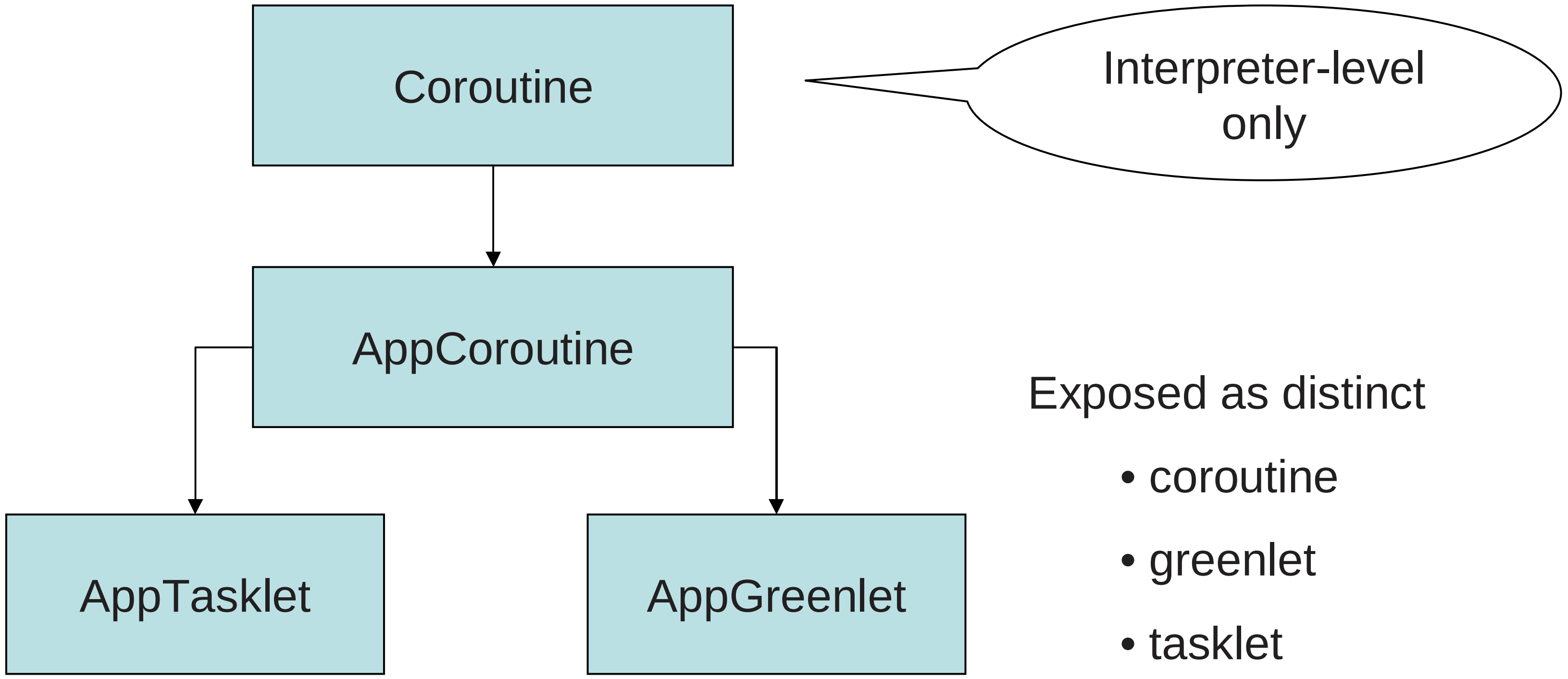
Taking the chance for a Redesign

- Minimalistic base implementation
  - Exposing high-speed coroutines to RPython, usable in other low-level modules
- Exposing multiple interfaces to application level
  - Coroutines, tasklets/channels, greenlets
  - Peaceful co-existence of concepts
  - Extensibility on both levels

What is a coroutine?

- Coroutines can „switch“ to each other
- There is always one „current“ coroutine monitored in a costate structure
- Current‘s state is on the machine stack
- Others are stored as a structure
- By switching, we replace „current“ by a different coroutine and update.
- This works multiple times

Class Hierarchy



(!) Inheritance just for implementation brevity, not exposed to the user

Exposed as distinct

- coroutine
- greenlet
- tasklet
- … others as needed

Simple API

- c = coroutine()
- c.bind(func, args)
- c.switch()

- c.alive
- c.kill()
- coroutine.getcurrent()

Enough to build everything else on top

How can they co-exist?

- Every coro-class has its own costate singleton instance (could be a class variable if RPython supported it)
- Coro-classes are created with an active instance representing the whole program
- A coro-class‘ current is by definition active until we change this coro-class‘ costate
- Coroutines don‘t see greenlets don‘t see tasklets don‘t see what has a different costate.

Implementation notes

- Extremely compact! Why?
  - Interp-level coroutines are 150 lines
  - Application level 100 extra lines
- PyPy is very cooperative
  - Built-in stack unwinding does all work
  - Writing support code in C is automated
  - Extremely tedious doing manually (sigh)
- Little input from me, mostly Armin‘s design™

Advantages

- No compromizes necessary, no fiddling with the C stack at all
  - Might become difficult again when we support calling into external functions
- No longer fighting the mainstream with patches
- Flexible modelling of different interfaces. It is Python, after all.

Future

- Create more APIs tailored for special needs
- Back-port the new design to CPython?