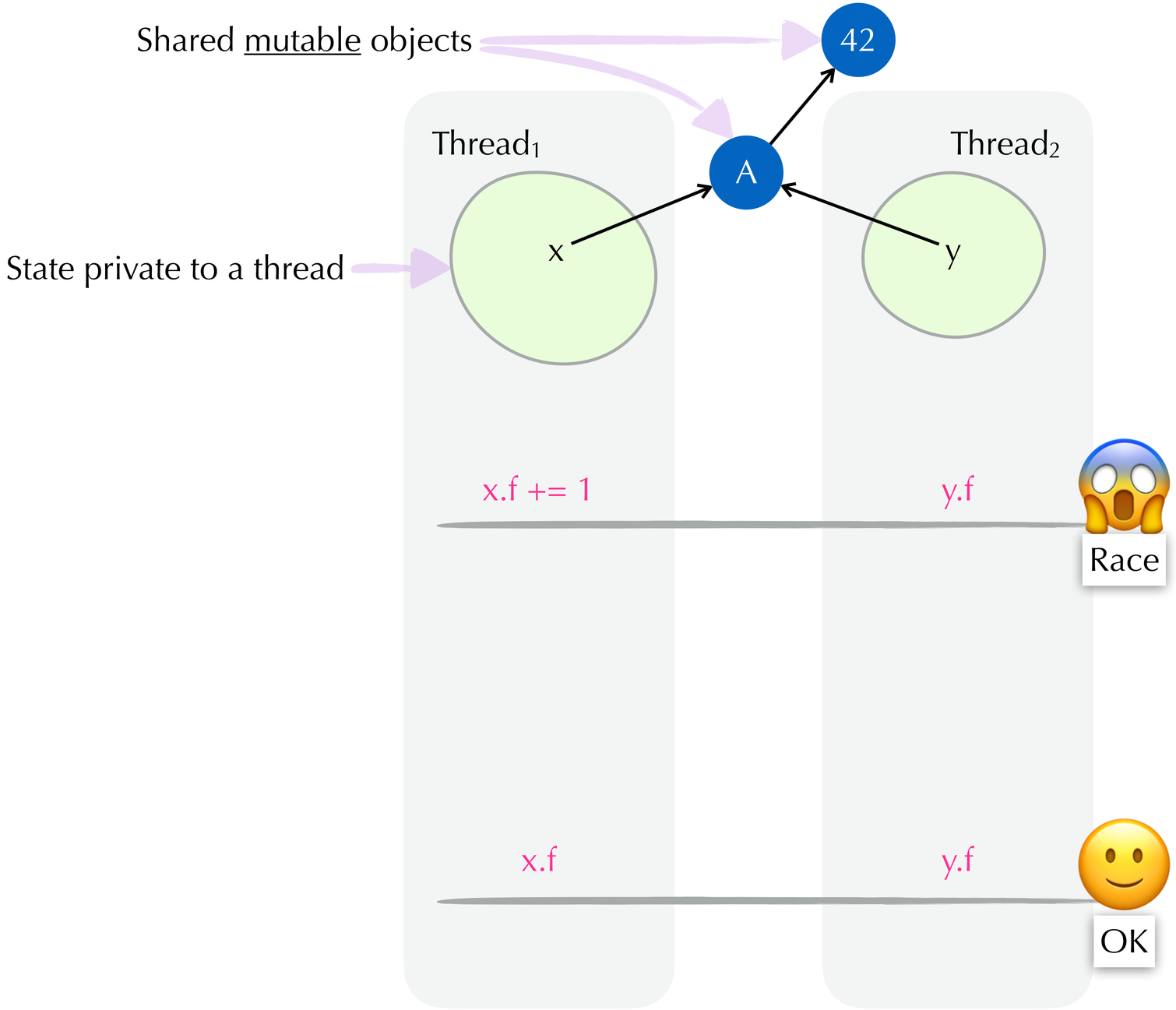
Fearless Concurrency in Python

Sylvan Clebsch (Azure Research, US)
Matt Johnson (Azure Research, UK)
Matt Parkinson (Azure Research, UK)
Fridtjof Stoldt (Uppsala University, Sweden)
Tobias Wrigstad (Uppsala University, Sweden)

This is the future for free-threaded Python programmers

Data races — ″heisenbugs″



Off to the data races

- The meaning of x.f += 1 depends on whether or not the **object** x **points to** is shared between threads

  There is **no easy way** to tell if x **points to** a shared object

  No easy way to ask on Stack Overflow / ChatGPT

- Data race bugs are heisenbugs

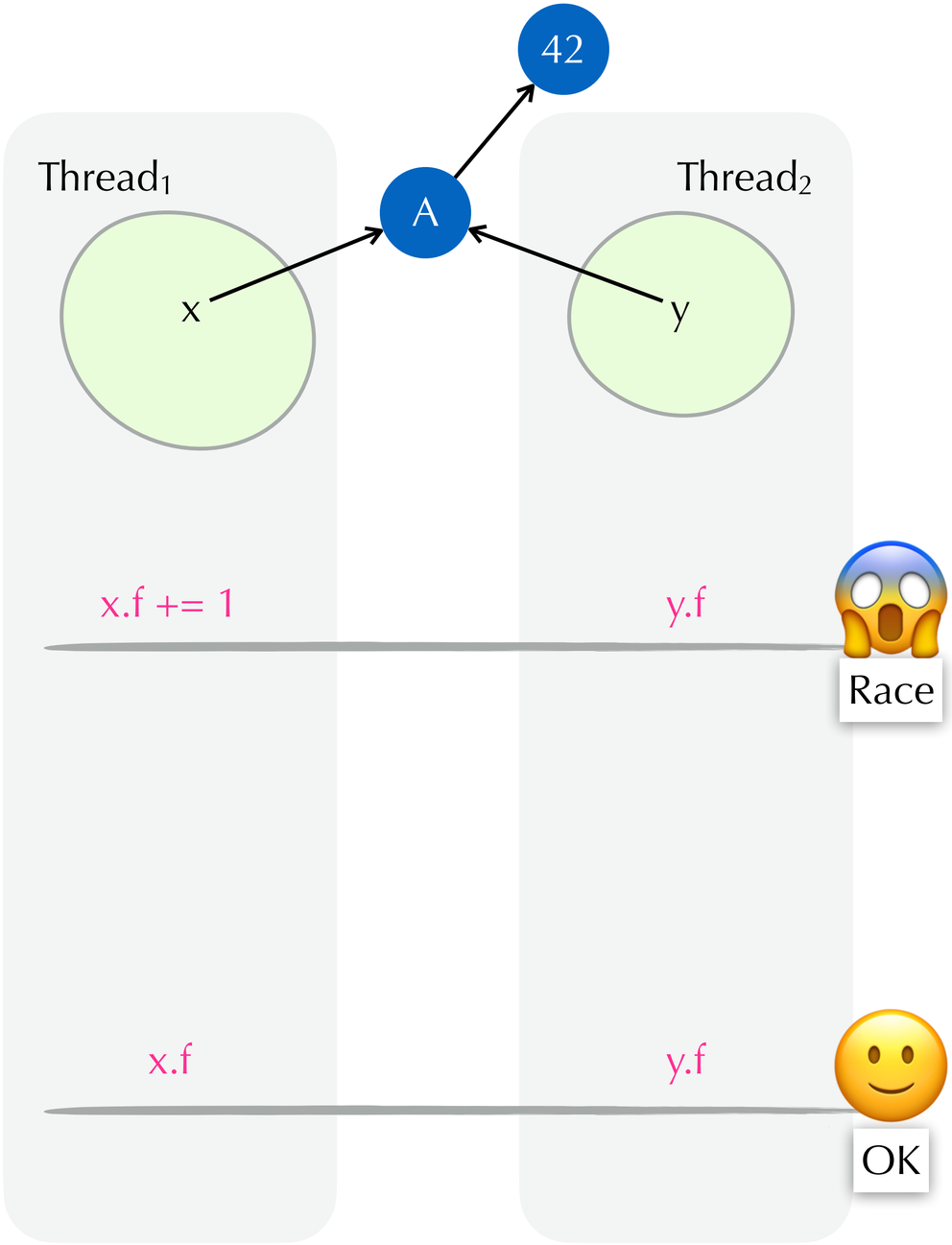
  Often silent wrong results — not a crash

  Hard to reproduce, debug, and fix

- Tools can help — if you realise you need them

  e.g. TSAN — but not familiar to Python programmers

- Costly for the runtime to protect itself against bad code



Off to the data races

- The meaning of `x.f += 1` depends on whether or not the **object** x **points to** is shared between threads

  There is **no easy way** to tell if x **points to** a shared object

  No easy way to ask on Stack Overflow / ChatGPT

- Data race bugs are heis

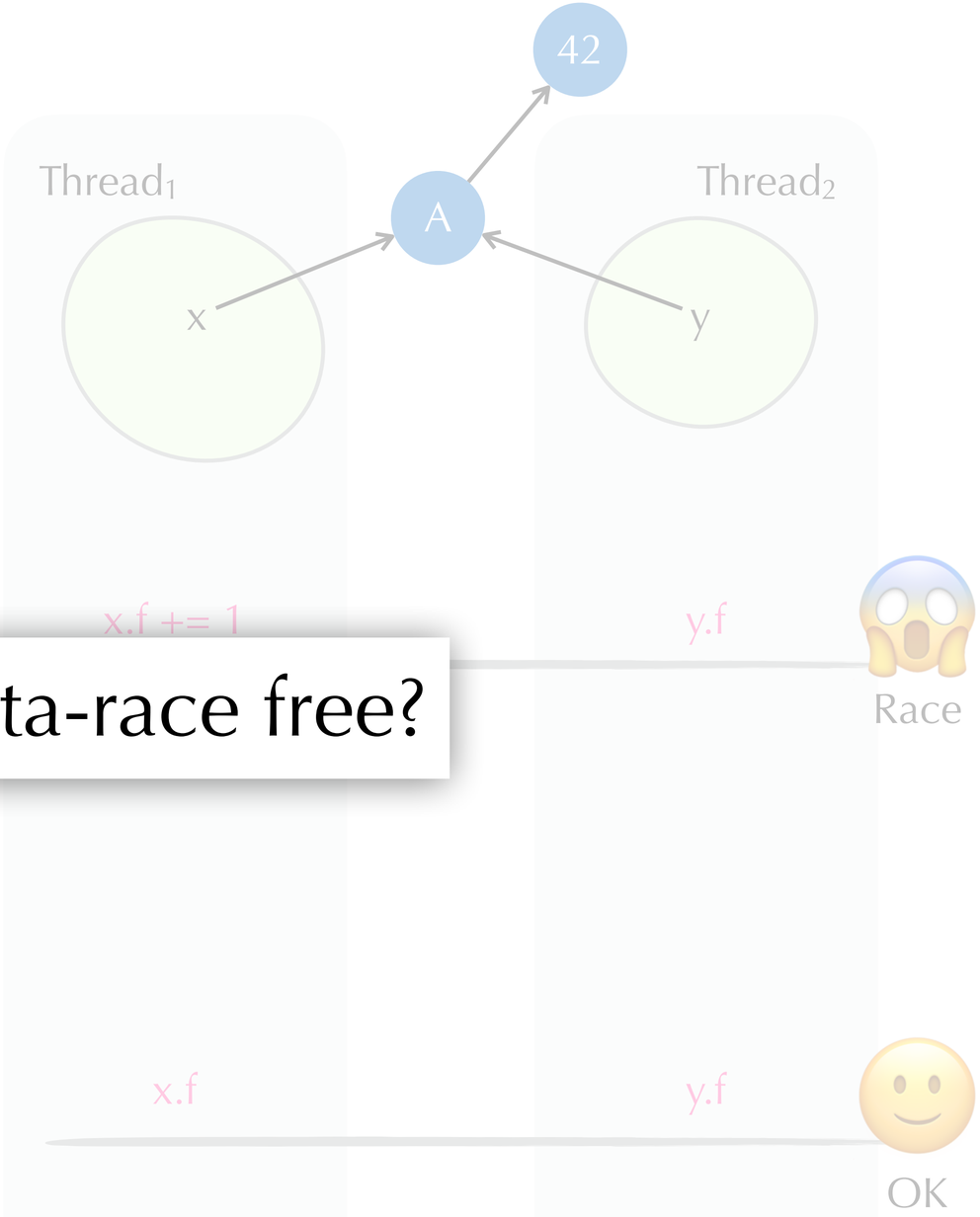
  Often silent wrong re

  Hard to reproduce, debug, and fix

- Tools can help — if you realise you need them

  e.g. TSAN — but not familiar to Python programmers

- Costly for the runtime to protect itself against bad code

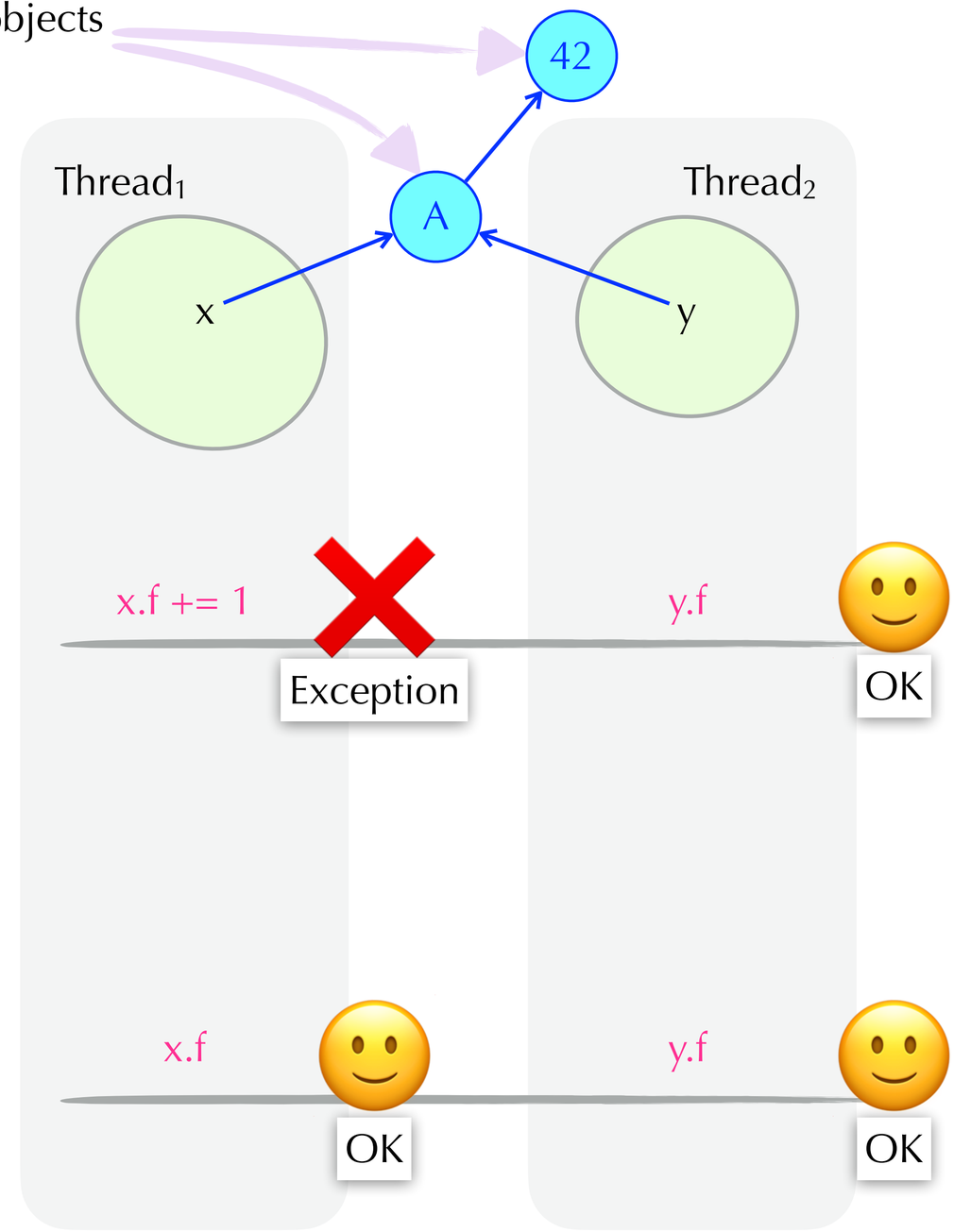


Can we make Python data-race free?

Shared immutable objects

What about deeply immutable state?

- **Model**
  - All shared memory is immutable — so there can be no data races
  - Can be checked efficiently at run-time
- **Advantages**
  - Mutating shared memory becomes an error — not a data race
  - Explainability goes up
  - Enables optimisations for the runtime
- **However**
  - Rules out mutation (which is common in Python)
  - Cannot transfer data without making it immutable



What about ownership á la Rust?

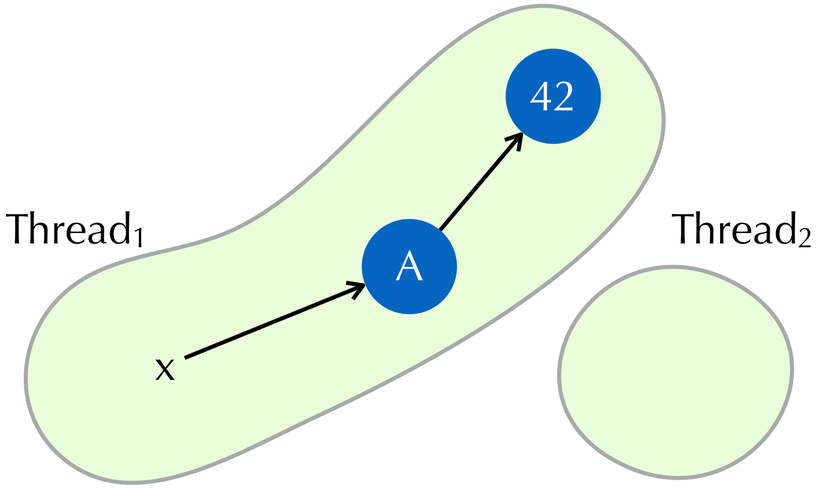
- **Model**

  Object with RC > 1 cannot be mutated

- **Advantages**

  Safe transfer of mutable objects (if RC = 1)

  Can construct safe locking protocols



x.f += 1 🙂 Safe, OK

What about ownership á la Rust?

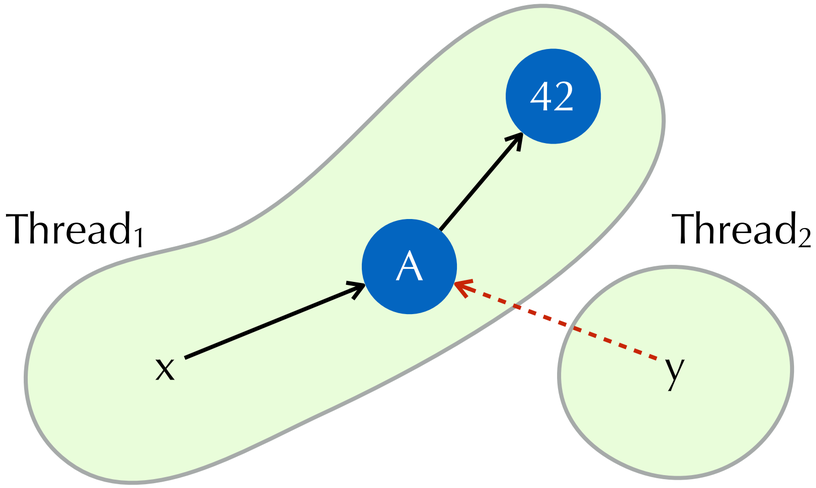
- **Model**

  Object with RC > 1 cannot be mutated

- **Advantages**

  Safe transfer of mutable objects (if RC = 1)

  Can construct safe locking protocols



x.f += 1 🙂 Safe, OK

y = x 🙂 Not allowed, prevents race

What about ownership á la Rust?

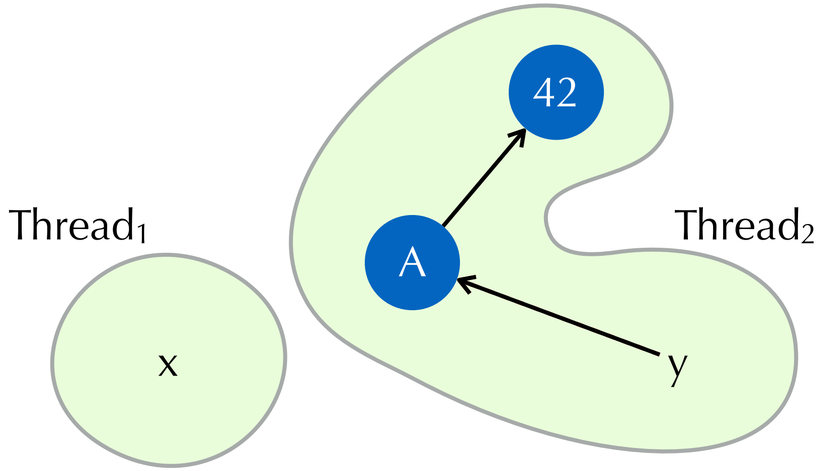
- **Model**

  Object with RC > 1 cannot be mutated

- **Advantages**

  Safe transfer of mutable objects (if RC = 1)

  Can construct safe locking protocols

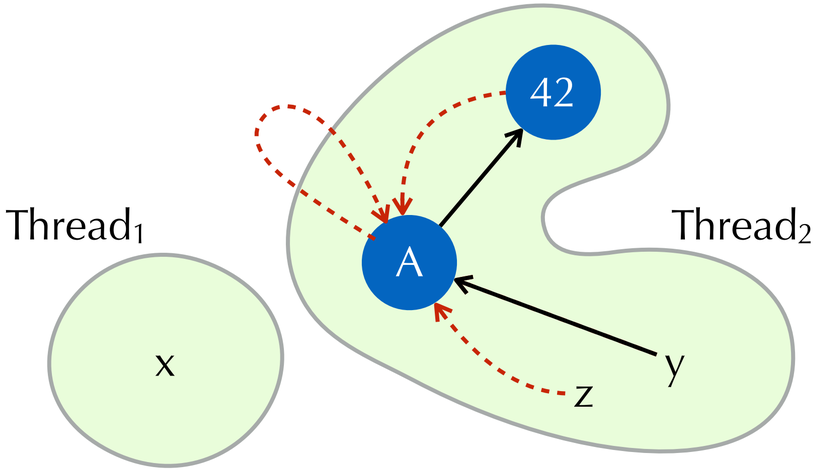


x.f += 1 🙂 Safe, OK

y = x 🙂 Not allowed, prevents race

y = **move** x 🙂 Safe, OK and sets x to **None**

What about ownership á la Rust?



- **Model**

  Object with RC > 1 cannot be mutated

- **Advantages**

  Safe transfer of mutable objects (if RC = 1)

  Can construct safe locking protocols

- **However**

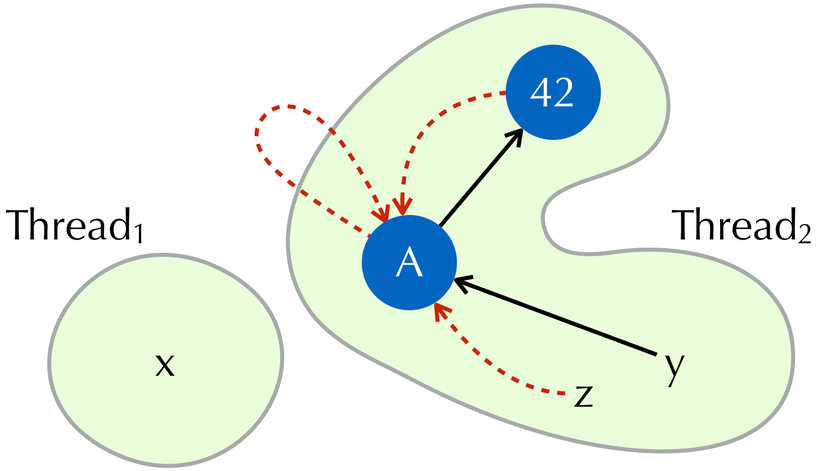
  Built on static types + static analysis (borrow checker)

  Massively restricts shape of object graphs

Would force rewriting most existing Python programs!

x.f += 1	🙂	Safe, OK
z = y	🙁	Safe but not allowed
y = x	🙂	Not allowed, prevents race
y = **move** x	🙂	Safe, OK and sets x to **None**

What about ownership á la Rust?



- **Model**

  Object with RC > 1 cannot be mutated

- **Advantages**

  Safe transfer of mutable objects (if RC = 1)

  Can construct safe locking protocols

- **However**

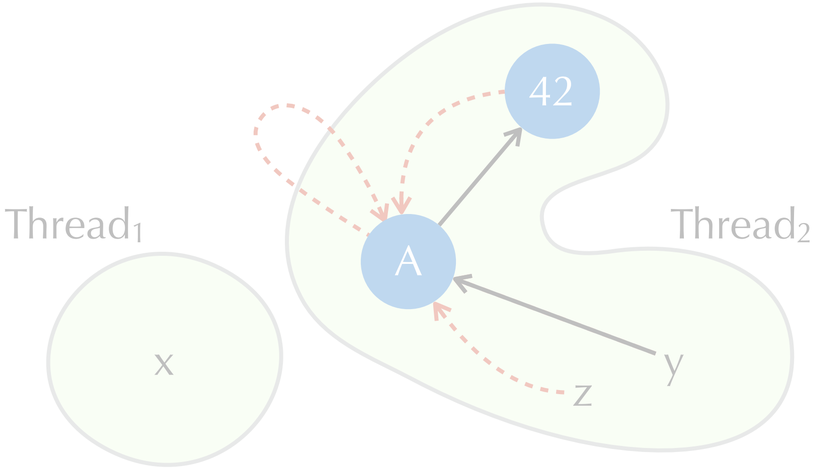
  Built on static types + static analysis (borrow checker)

  Massively restricts shape of object graphs

Would force rewriting most existing Python programs!

x.f += 1	🙂	Safe, OK
z = y	😕	Safe but not allowed
y = x	🙂	Not allowed, prevents race
y = **move** x	🙂	Safe, OK and sets x to **None**

What about ownership á la Rust?



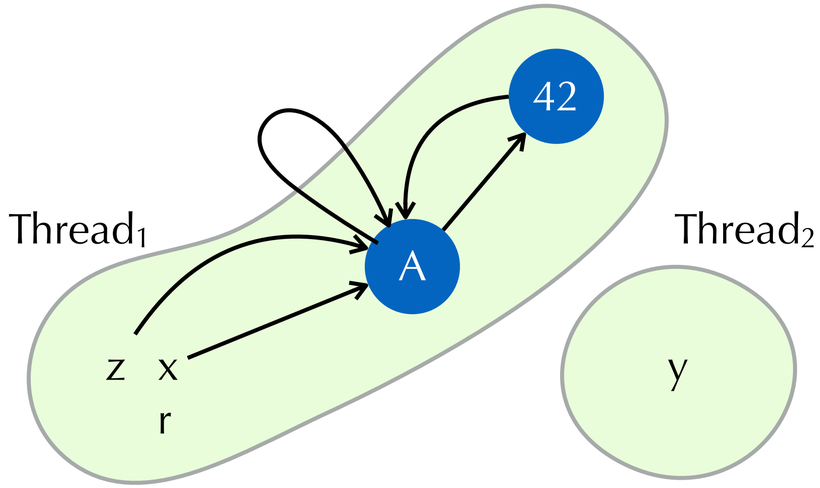
- **Model**
  - Object with RC > 1 cannot be mutated
- **Advantages**
  - Safe transfer of mutab
  - Can construct safe locking protocols
- **However**
  - Built on static types + static analysis (borrow checker)
  - Massively restricts shape of object graphs

Does a more Pythonic solution exist?

	🙂	Safe, OK
z = y	🙁	Safe but not allowed
y = x	🙂	Not allowed, prevents race
y = **move** x	🙂	Safe, OK and sets x to **None**

💀 **Would force rewriting most existing Python programs!** 💀

Lungfish — Pythonic ownership!



- **Region-based ownership**
  - More permissive than Rust — yet data-race free
  - Can be checked efficiently at run-time
  - Sensible error messages
- Region = group of mutable objects
  - Isolated
  - Share, transfer, make immutable — together
  - Can be nested
- References into region tracked dynamically using a ”Borrow Count”

Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

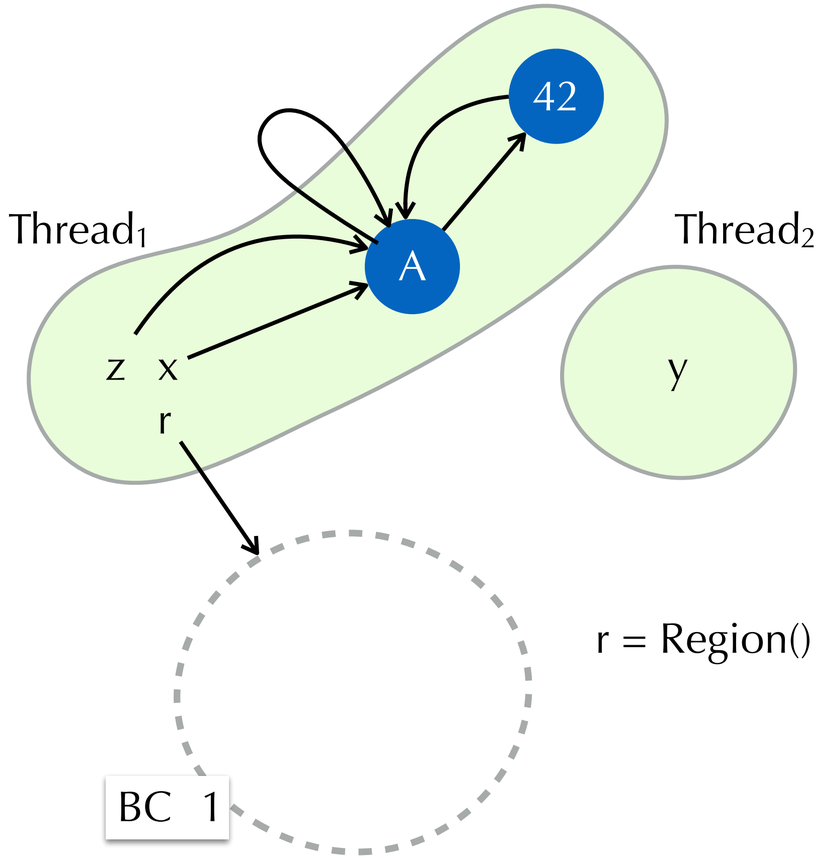
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using a ”Borrow Count”



Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

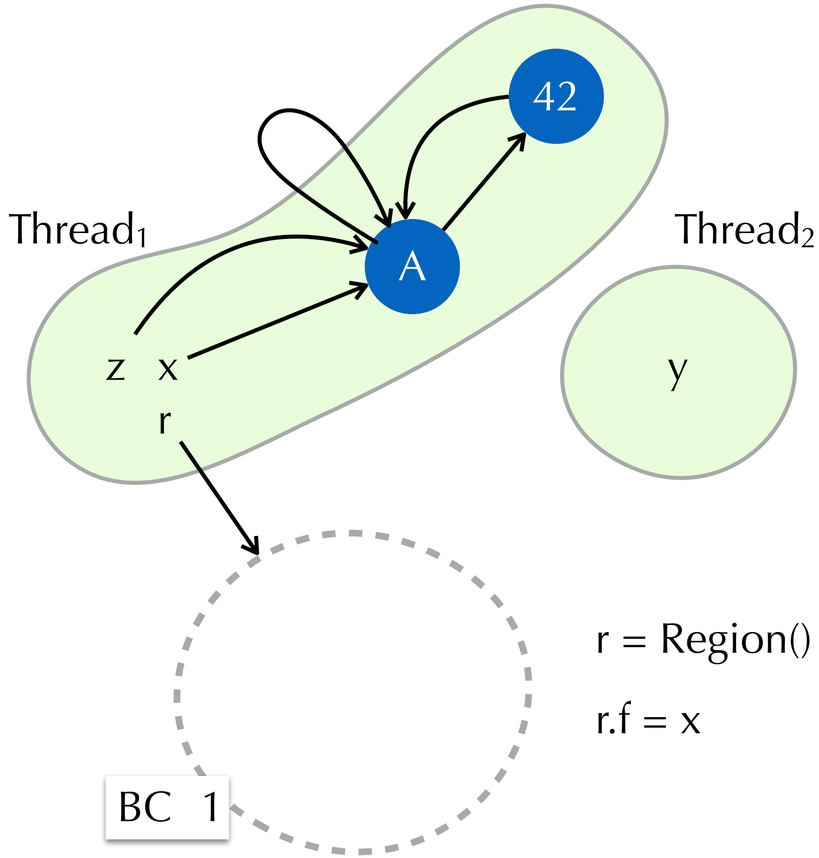
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using a ”Borrow Count”



Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

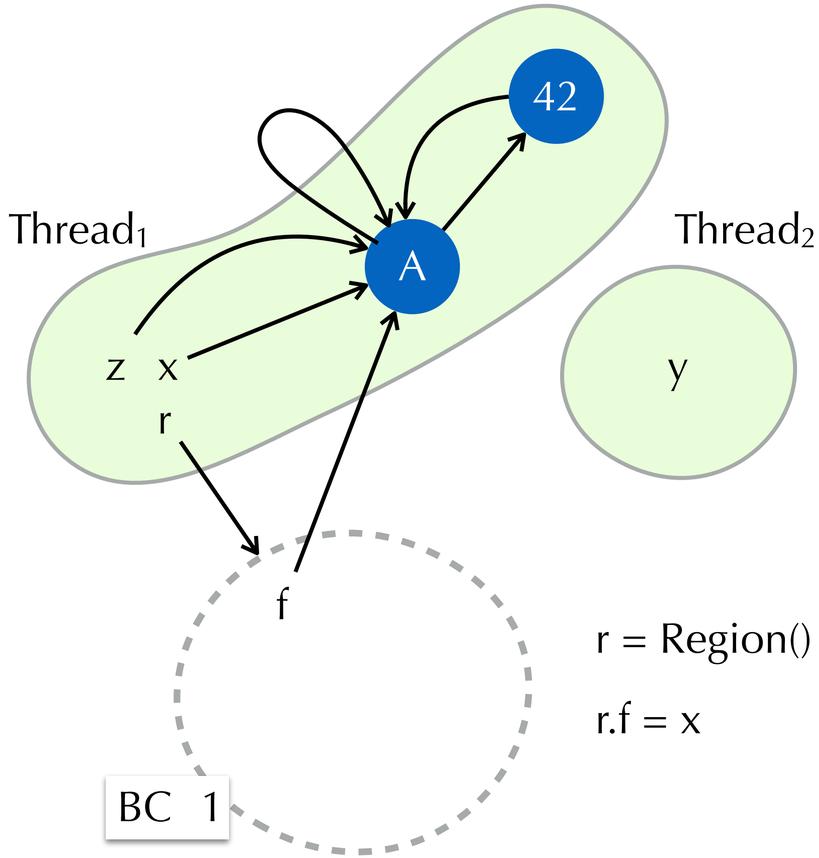
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using a ”Borrow Count”



Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

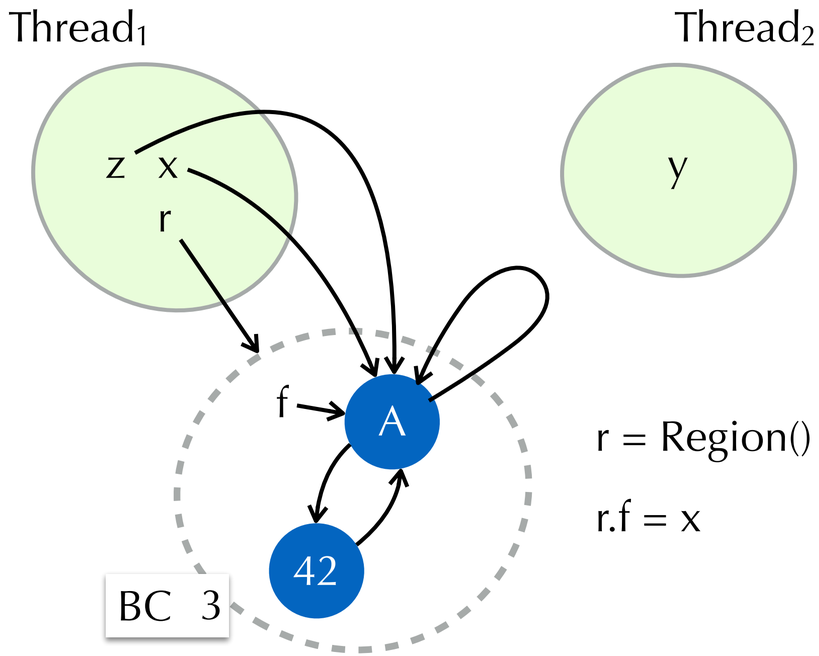
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using a ”Borrow Count”



Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

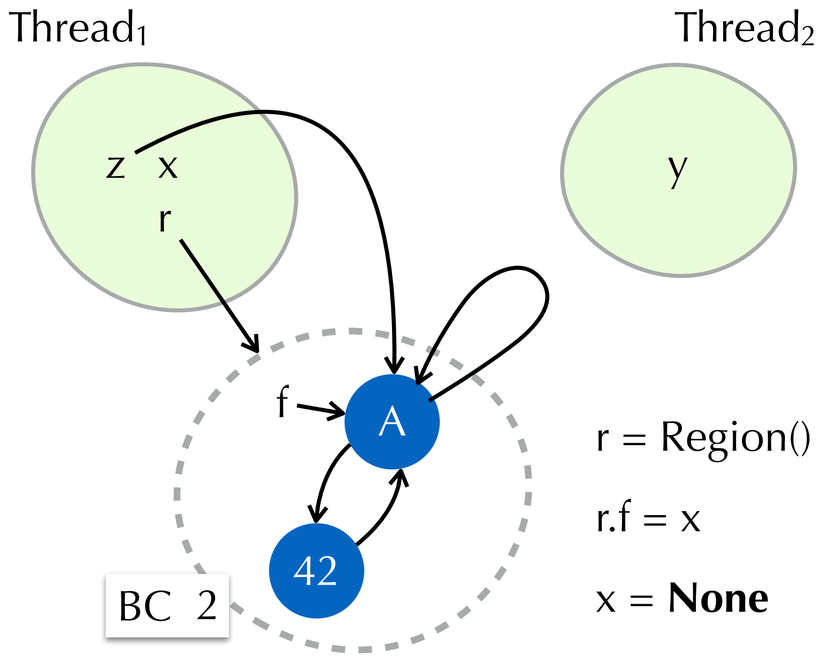
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using a ”Borrow Count”



Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

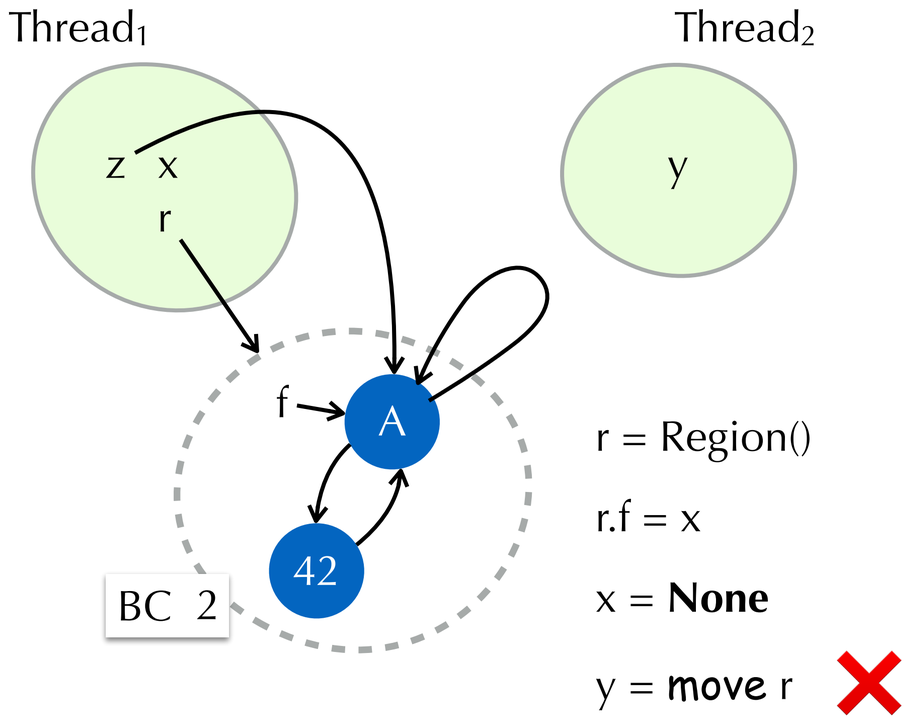
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using a ”Borrow Count”



Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

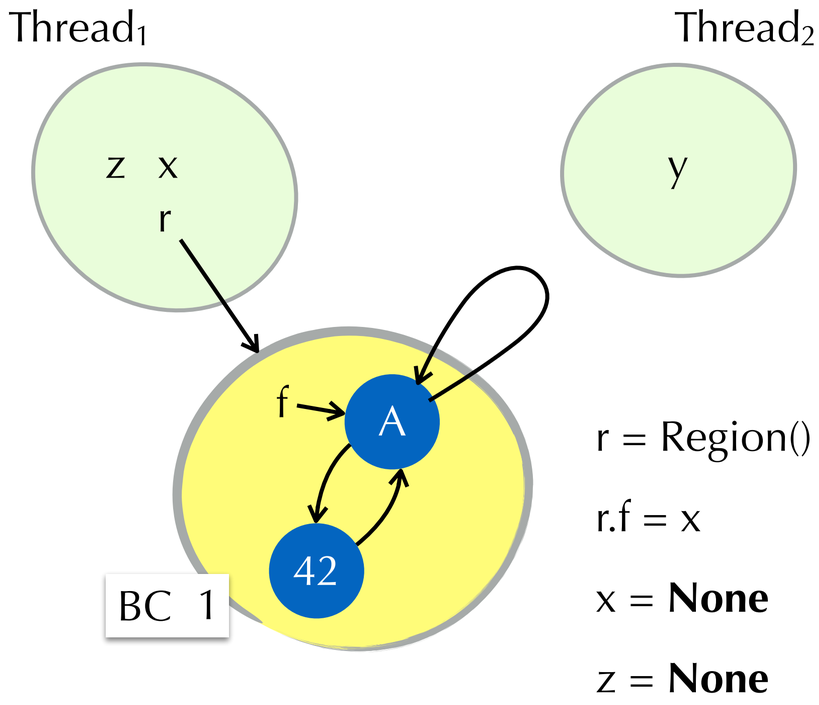
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using a ”Borrow Count”



Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

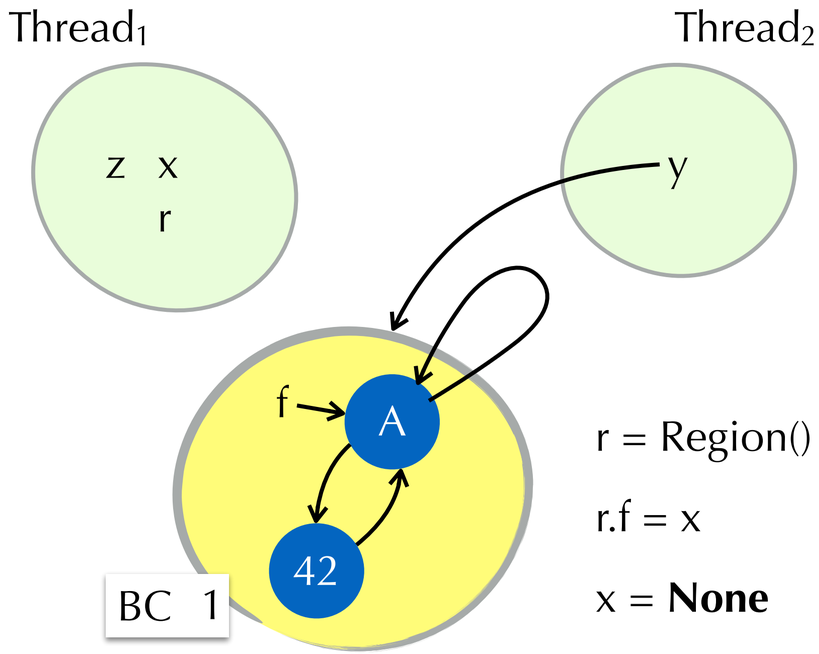
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using a ”Borrow Count”



Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

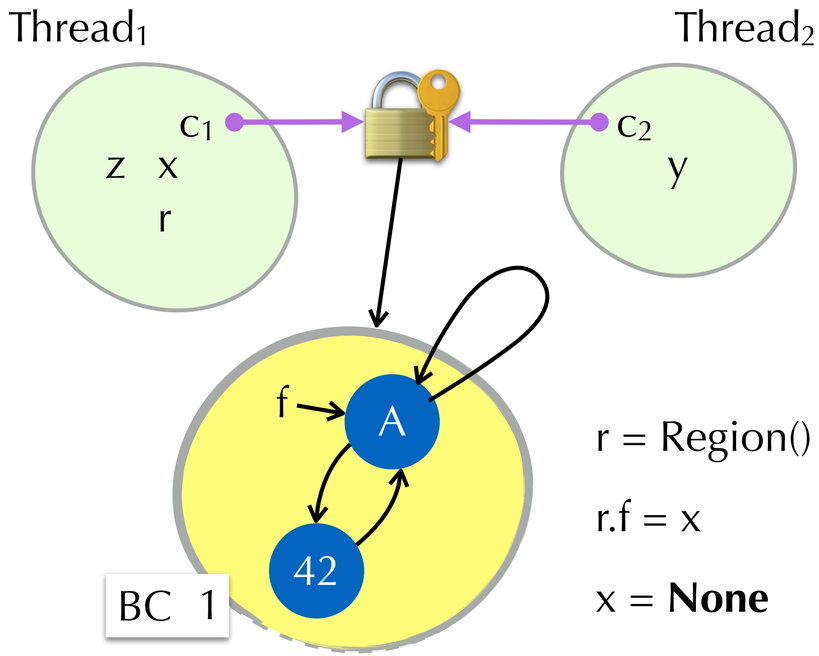
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using a ”Borrow Count”



r = Region()

r.f = x

x = **None**

z = **None**

y = **move** r

c1, c2 = **share** y

OK

Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

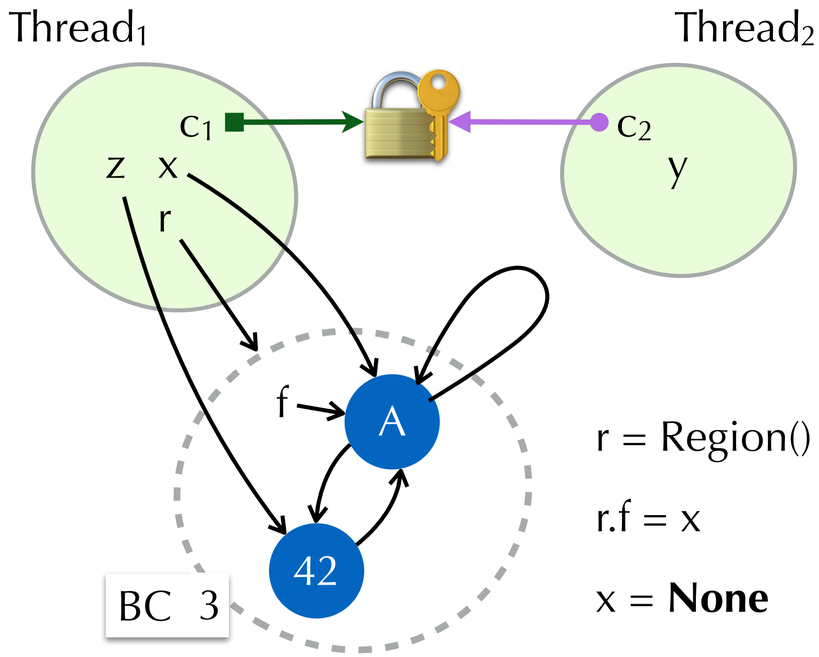
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using a ”Borrow Count”



r = Region()

r.f = x

x = **None**

z = **None**

y = **move** r

c1, c2 = **share** y

OK

Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

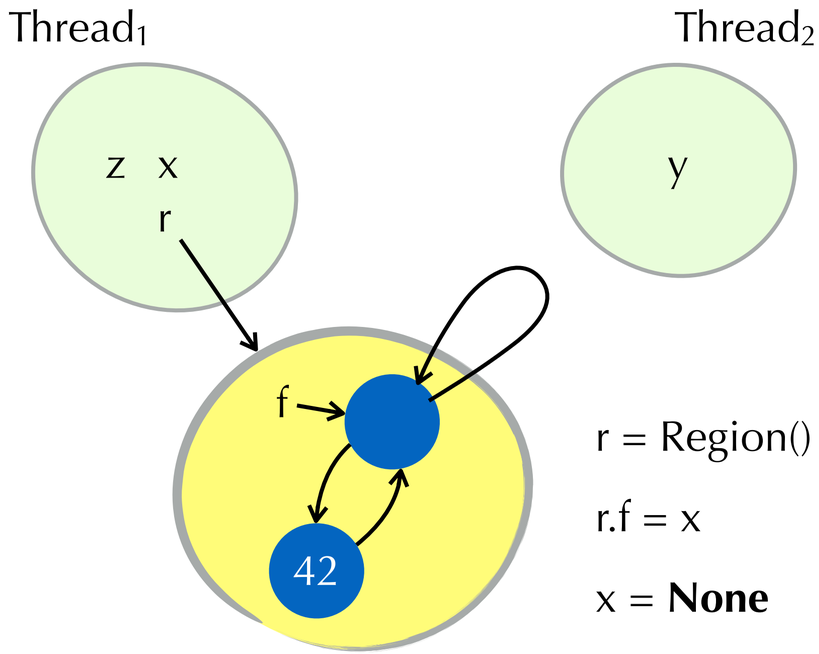
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using a ”Borrow Count”




```
r = Region()
r.f = x
x = None
z = None
freeze(r)
# x,  y = …
```

Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

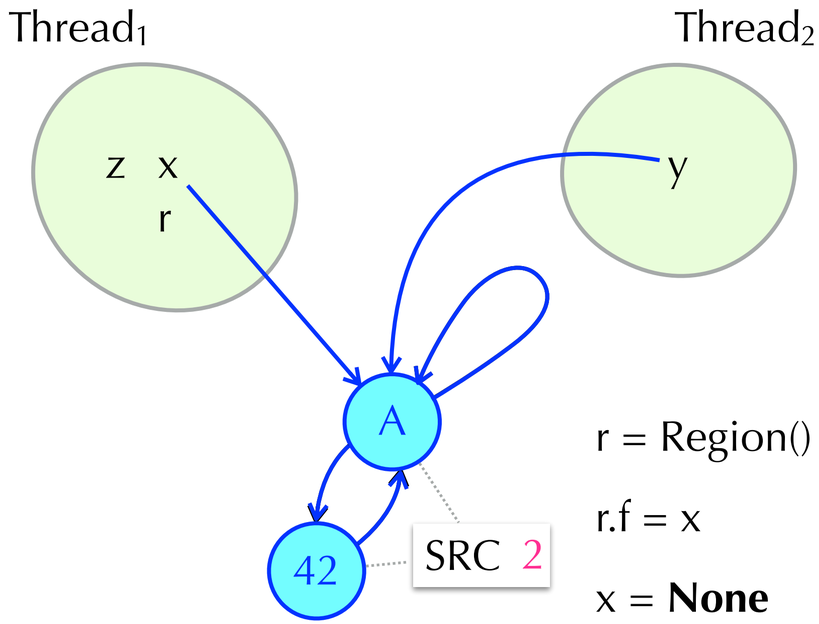
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using a ”Borrow Count”




```
r = Region()
r.f = x
x = None
z = None
freeze(r)
# x,  y = …
```

Lungfish — Pythonic ownership!

- **Region-based ownership**

  More permissive than Rust — yet data-race free

  Can be checked efficiently at run-time

  Sensible error messages

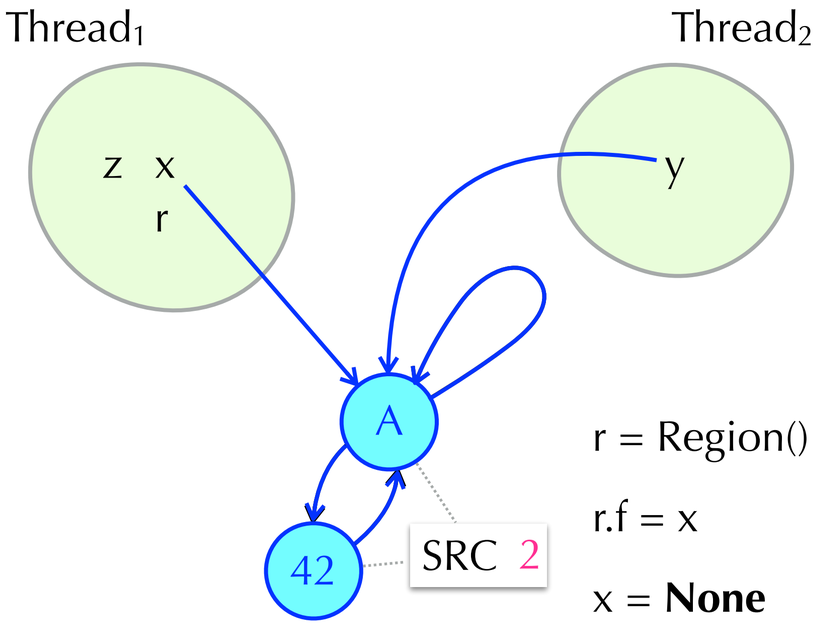
- Region = group of mutable objects

  Isolated

  Share, transfer, make immutable — together

  Can be nested

- References into region tracked dynamically using




```
r = Region()
r.f = x
x = None
z = None
freeze(r)
# x,  y = …
```

✅ Transfer

✅ Mutable sharing

✅ Immutable sharing

✅ Enables optimisations

– Efficient GC

– Immutable obj. rep.

A Plan for "Fearless Python"

- We propose to make Python data-race free through a series of Python Enhancement Proposals:

  PEP A: Deep Immutability

  Implementation status (**note:** on-top of 3.12): 99% ✅

  PEP B: Manage cyclic immutable garbage with reference counting + atomic reference counting of immutable objects

  Implementation status: 80% 🚧

  PEP C: Share **immutable** data between threads and subinterpreters (PEP734)

  Implementation status: ≈50% 🚧

  PEP D: Share **mutable data** between threads and subinterpreters using regions

  Implementation status: 🤷 🚧

- Lungfish offers fearless concurrency both with and without GIL(s) — free-threaded or with subinterpreters

We thank the **Faster CPython team** for helping us refine these ideas for ~2 years

Now we are looking forward to **working with all of you** to explore the possibility of "Fearless Python"!


Matt J


Sylvan


Matt P

Go here for more info and a copy of this presentation:

https://microsoft.github.io/verona/pyrona.html


Fridtjof


Lungfish


Tobias