vLLM code 0.6.3

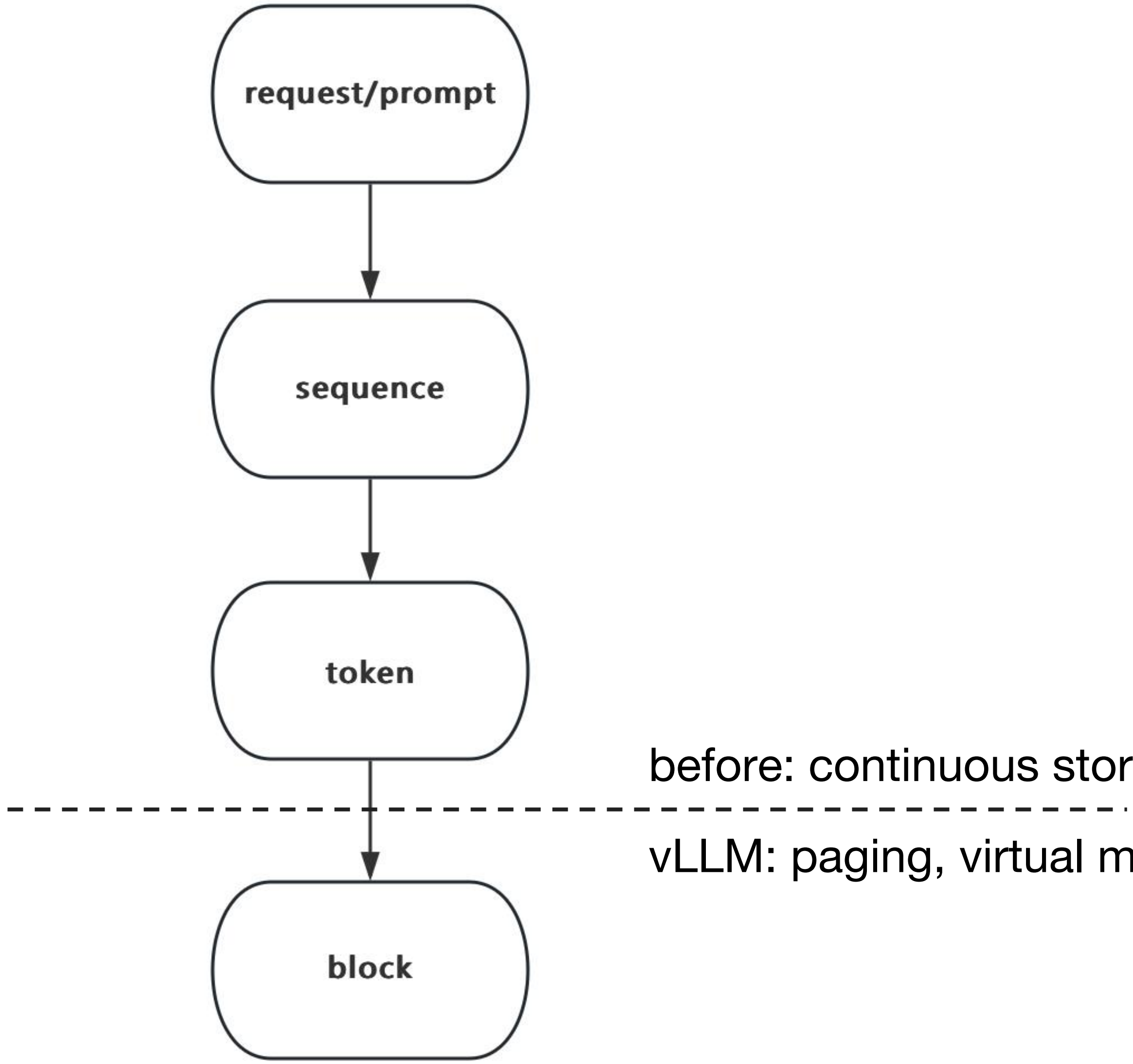
NaiveBlockAllocator

Chenye Wang

Oct 15, 2024

[1] https://github.com/vllm-project/vllm/tree/main/vllm/core/block

Preknowledge



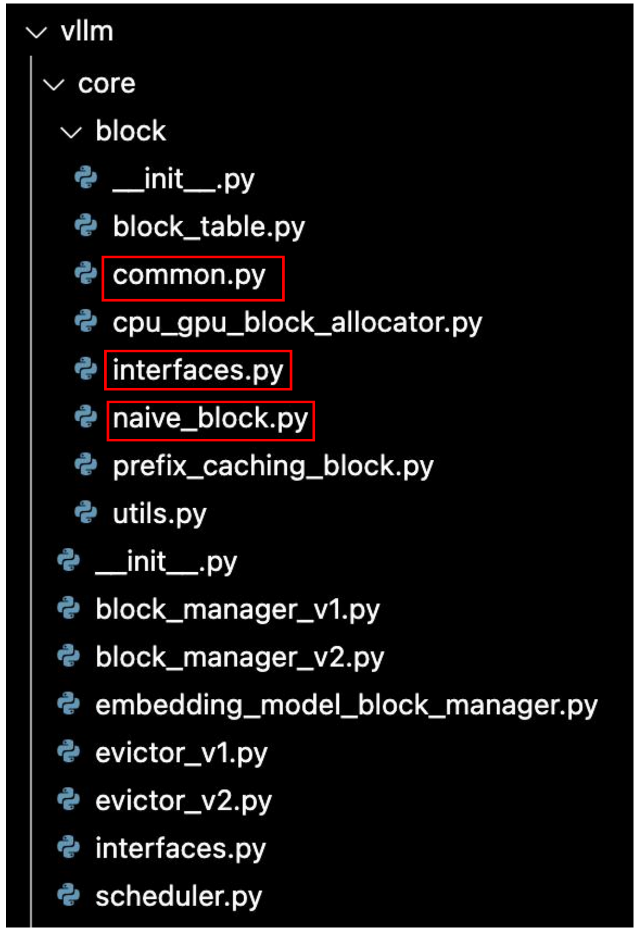
before: continuous storage

vLLM: paging, virtual memory

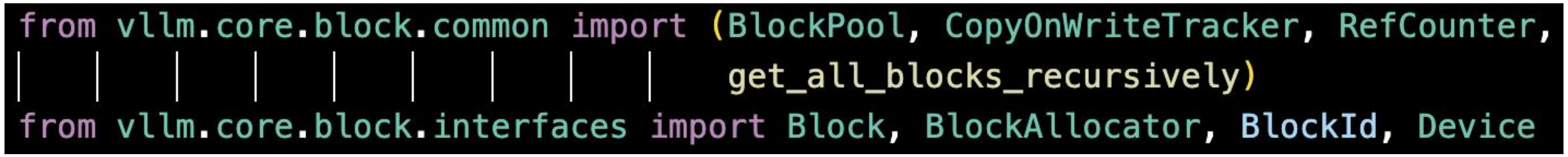
Start from block

- In python, we can discover module dependencies by "from ... import ..." statement.

- example
  - naive_block.py

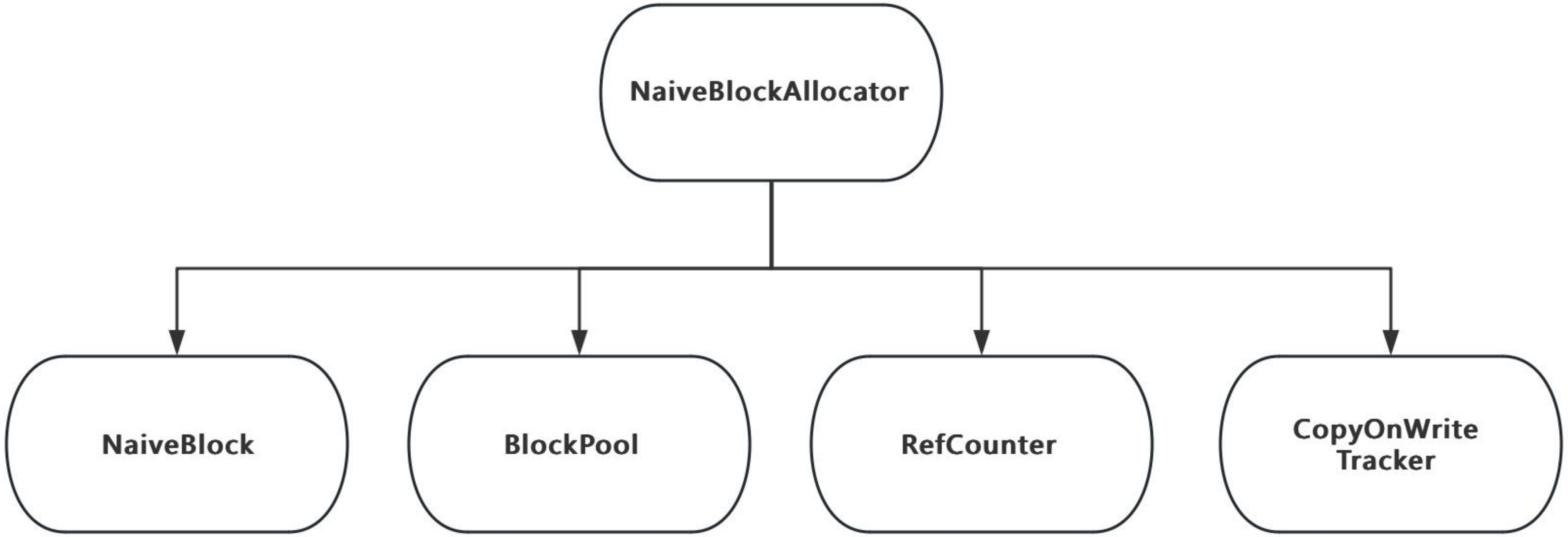


```
vllm
  core
    block
      __init__.py
      block_table.py
      common.py
      cpu_gpu_block_allocator.py
      interfaces.py
      naive_block.py
      prefix_caching_block.py
      utils.py
    __init__.py
    block_manager_v1.py
    block_manager_v2.py
    embedding_model_block_manager.py
    evictor_v1.py
    evictor_v2.py
    interfaces.py
    scheduler.py
```

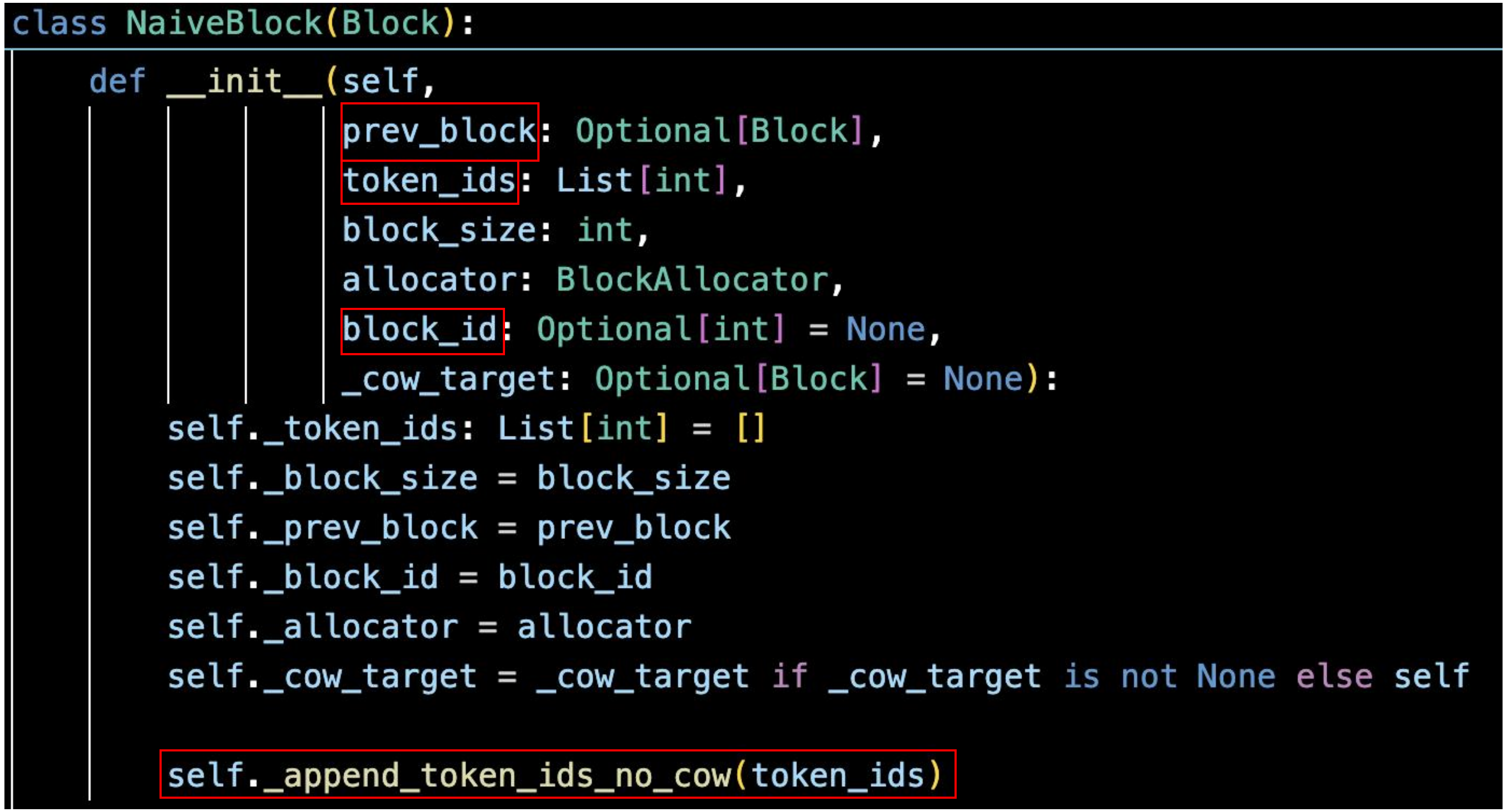


```
from vllm.core.block.common import (BlockPool, CopyOnWriteTracker, RefCounter,
                                    get_all_blocks_recursively)
from vllm.core.block.interfaces import Block, BlockAllocator, BlockId, Device
```

Module dependency

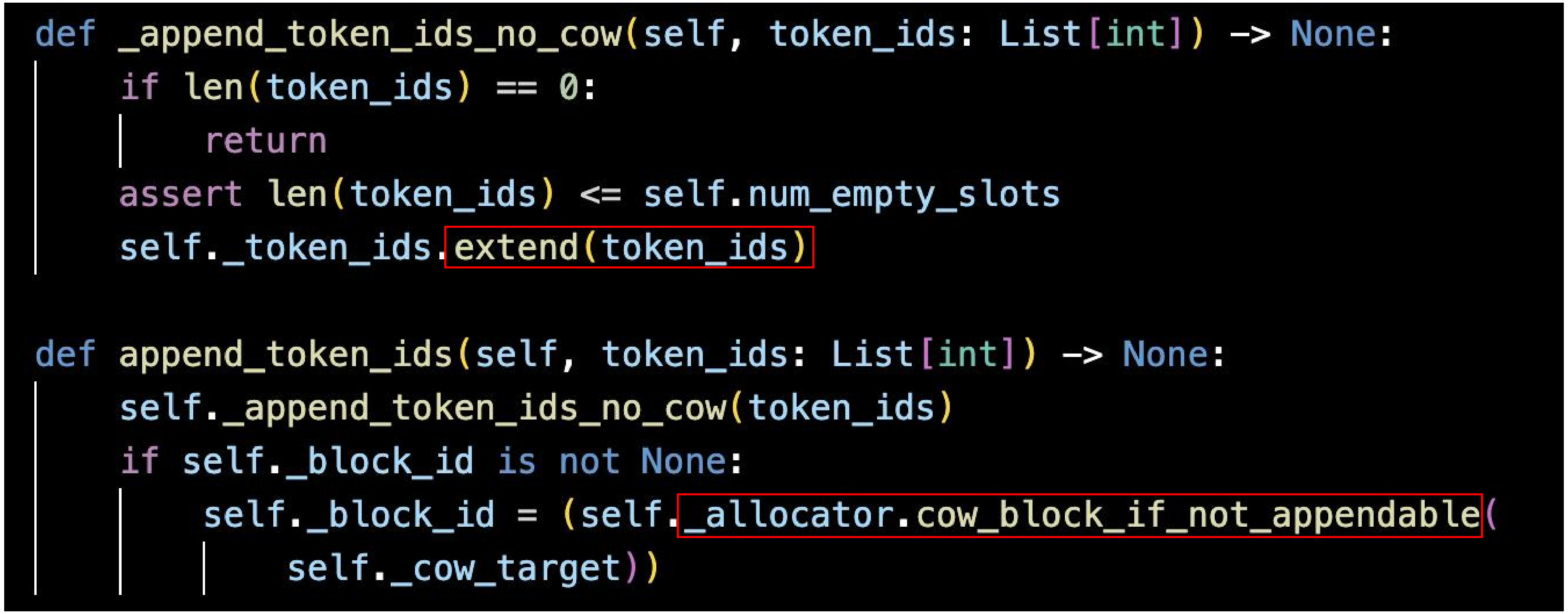


Block Initialization



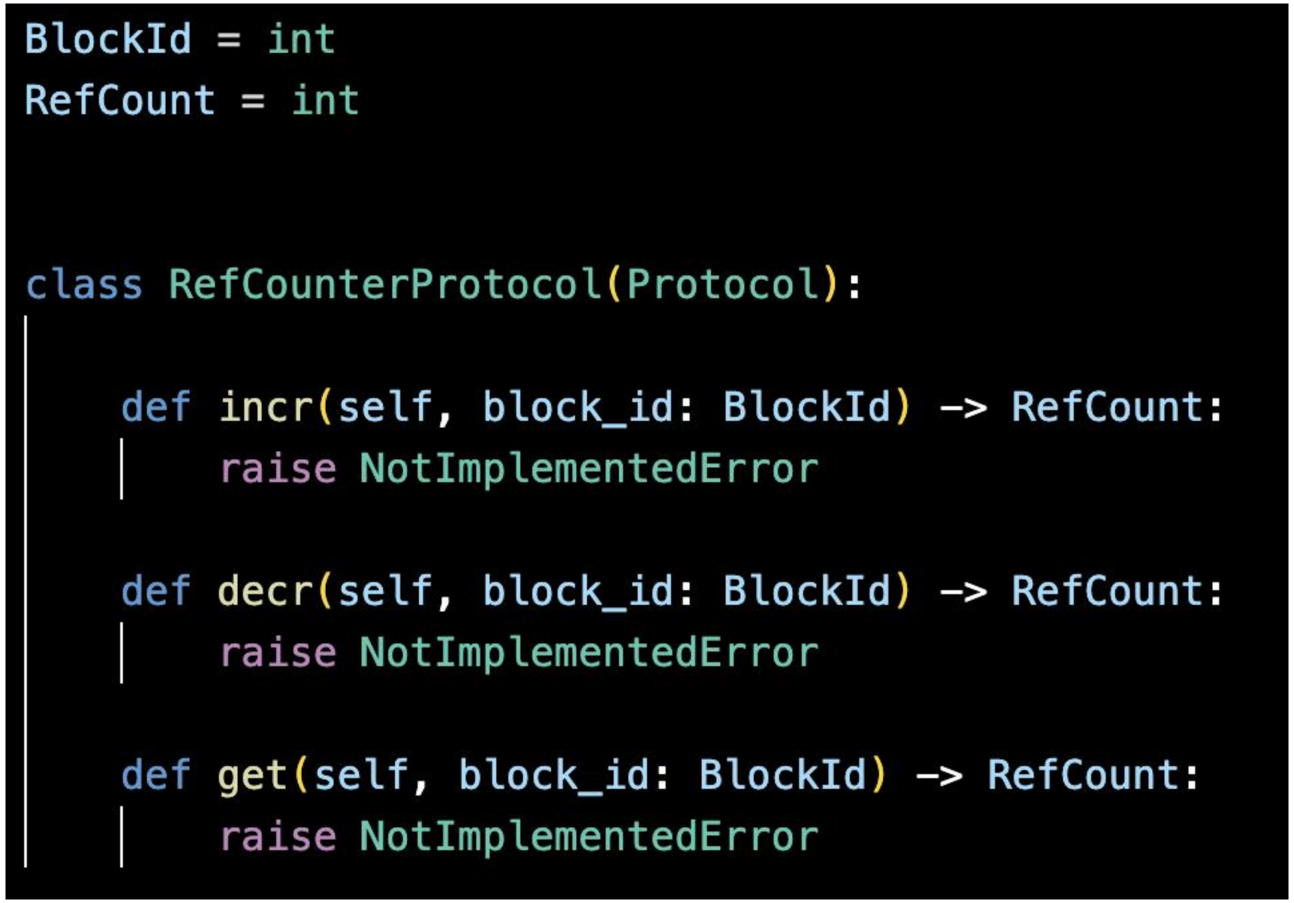
```python
class NaiveBlock(Block):
    def __init__(self,
                 prev_block: Optional[Block],
                 token_ids: List[int],
                 block_size: int,
                 allocator: BlockAllocator,
                 block_id: Optional[int] = None,
                 _cow_target: Optional[Block] = None):
        self._token_ids: List[int] = []
        self._block_size = block_size
        self._prev_block = prev_block
        self._block_id = block_id
        self._allocator = allocator
        self._cow_target = _cow_target if _cow_target is not None else self

        self._append_token_ids_no_cow(token_ids)
```

How to store tokens?



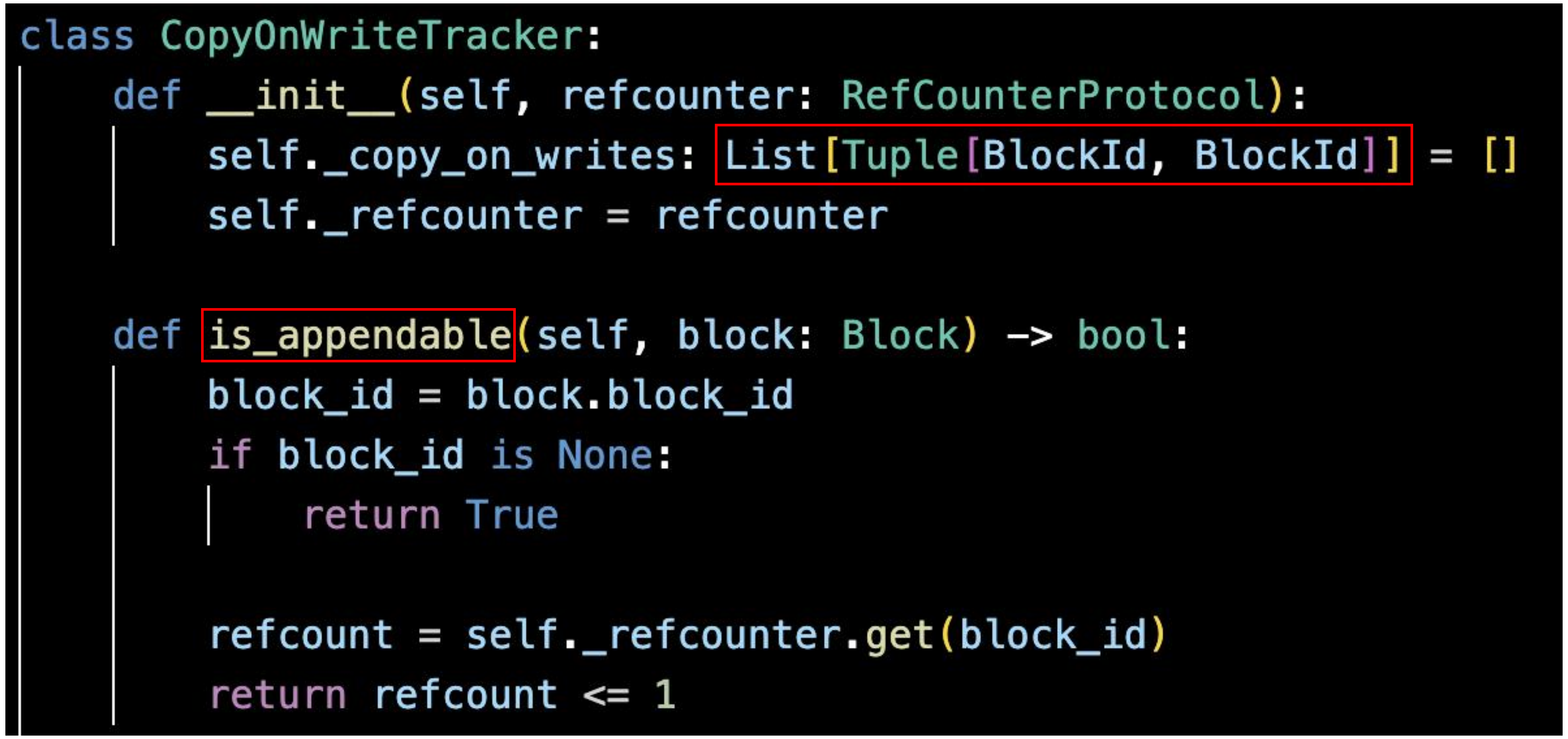
```python
def _append_token_ids_no_cow(self, token_ids: List[int]) -> None:
    if len(token_ids) == 0:
        return
    assert len(token_ids) <= self.num_empty_slots
    self._token_ids.extend(token_ids)

def append_token_ids(self, token_ids: List[int]) -> None:
    self._append_token_ids_no_cow(token_ids)
    if self._block_id is not None:
        self._block_id = (self._allocator.cow_block_if_not_appendable(
            self._cow_target))
```

Reference count



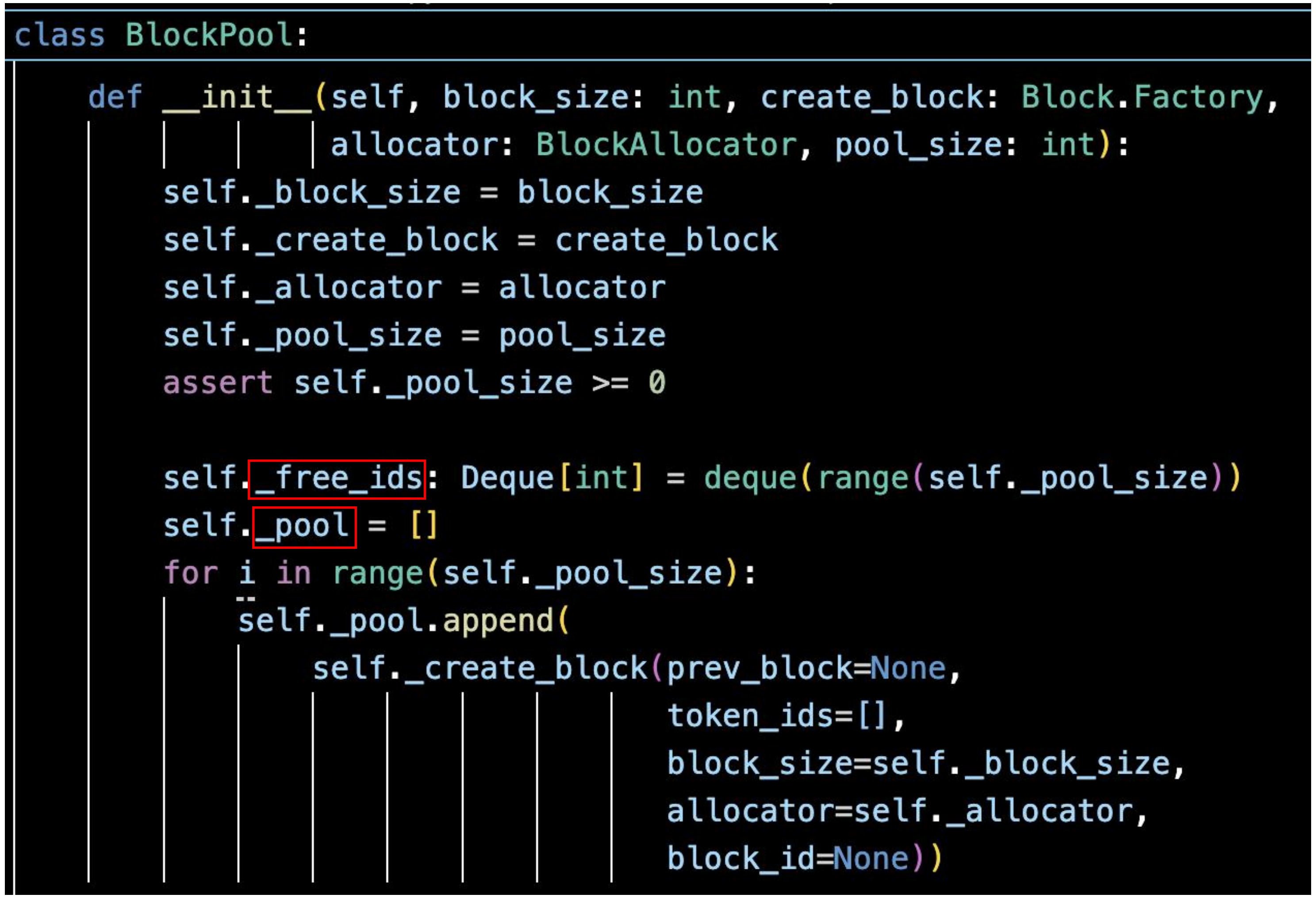
```python
BlockId = int
RefCount = int


class RefCounterProtocol(Protocol):

    def incr(self, block_id: BlockId) -> RefCount:
        raise NotImplementedError

    def decr(self, block_id: BlockId) -> RefCount:
        raise NotImplementedError

    def get(self, block_id: BlockId) -> RefCount:
        raise NotImplementedError
```

Copy-on-write tracker



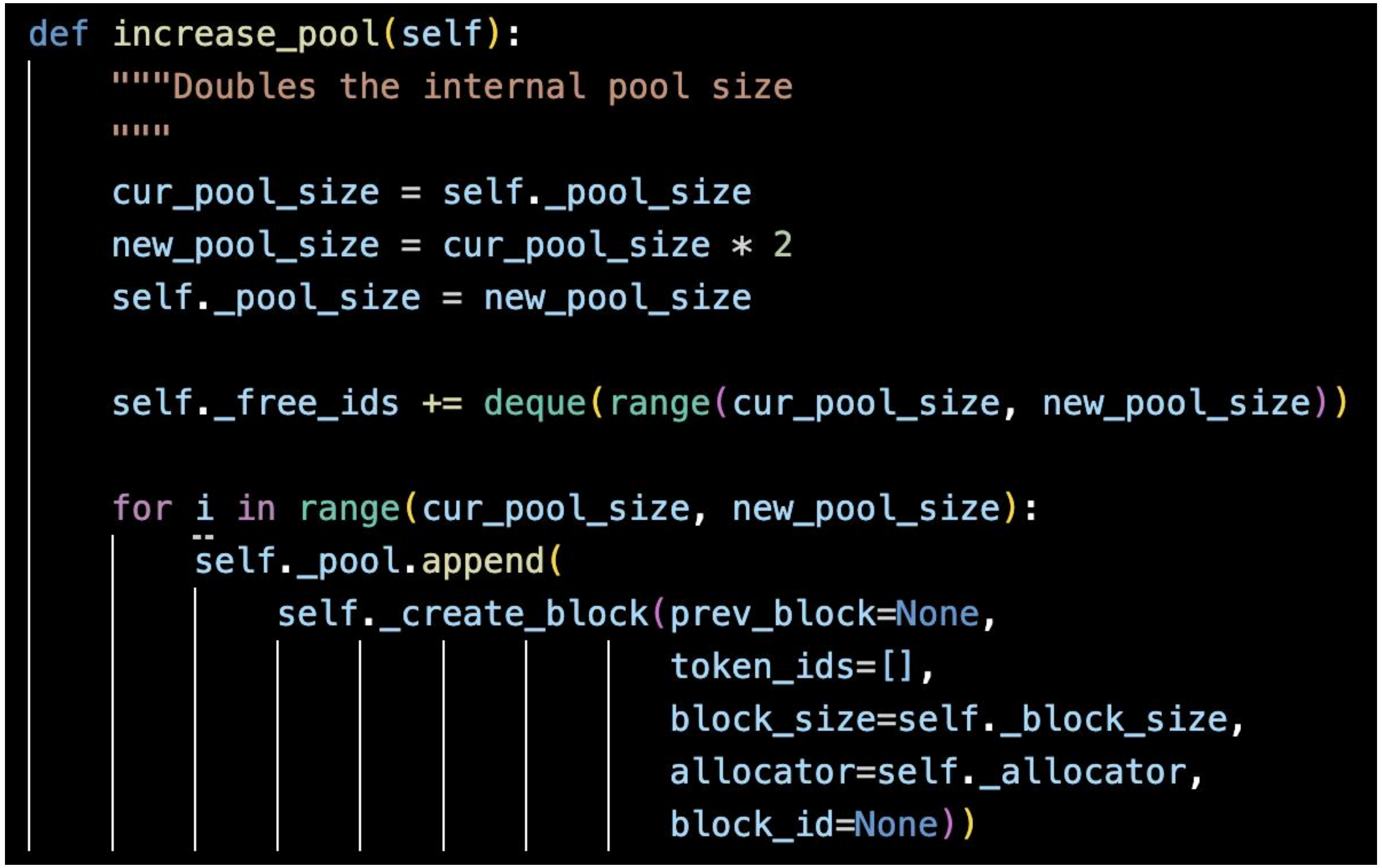
```python
class CopyOnWriteTracker:
    def __init__(self, refcounter: RefCounterProtocol):
        self._copy_on_writes: List[Tuple[BlockId, BlockId]] = []
        self._refcounter = refcounter

    def is_appendable(self, block: Block) -> bool:
        block_id = block.block_id
        if block_id is None:
            return True

        refcount = self._refcounter.get(block_id)
        return refcount <= 1
```

Block pool



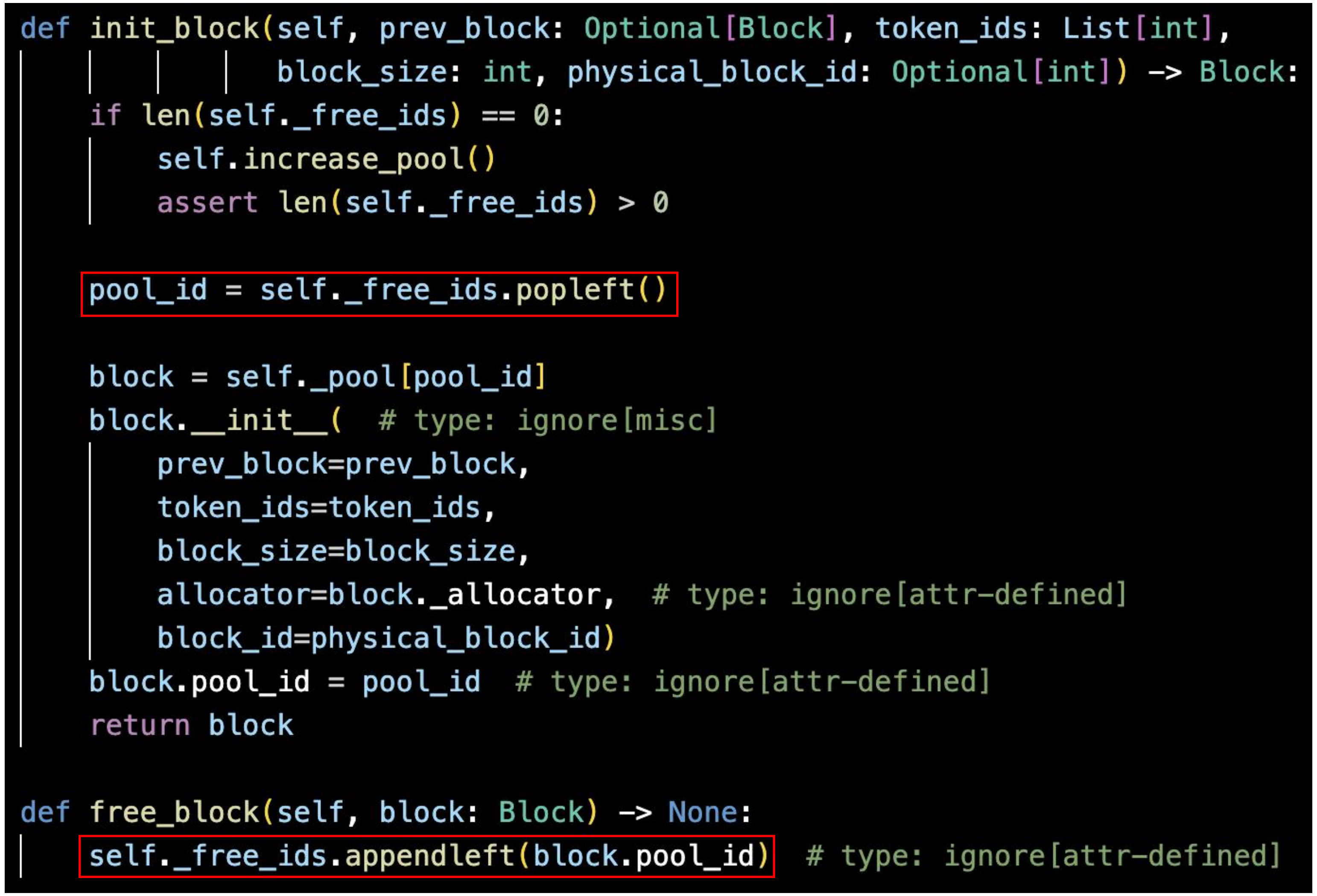
```
class BlockPool:

    def __init__(self, block_size: int, create_block: Block.Factory,
                 allocator: BlockAllocator, pool_size: int):
        self._block_size = block_size
        self._create_block = create_block
        self._allocator = allocator
        self._pool_size = pool_size
        assert self._pool_size >= 0

        self._free_ids: Deque[int] = deque(range(self._pool_size))
        self._pool = []
        for i in range(self._pool_size):
            self._pool.append(
                self._create_block(prev_block=None,
                                   token_ids=[],
                                   block_size=self._block_size,
                                   allocator=self._allocator,
                                   block_id=None))
```

What if there are not enough blocks in the pool?



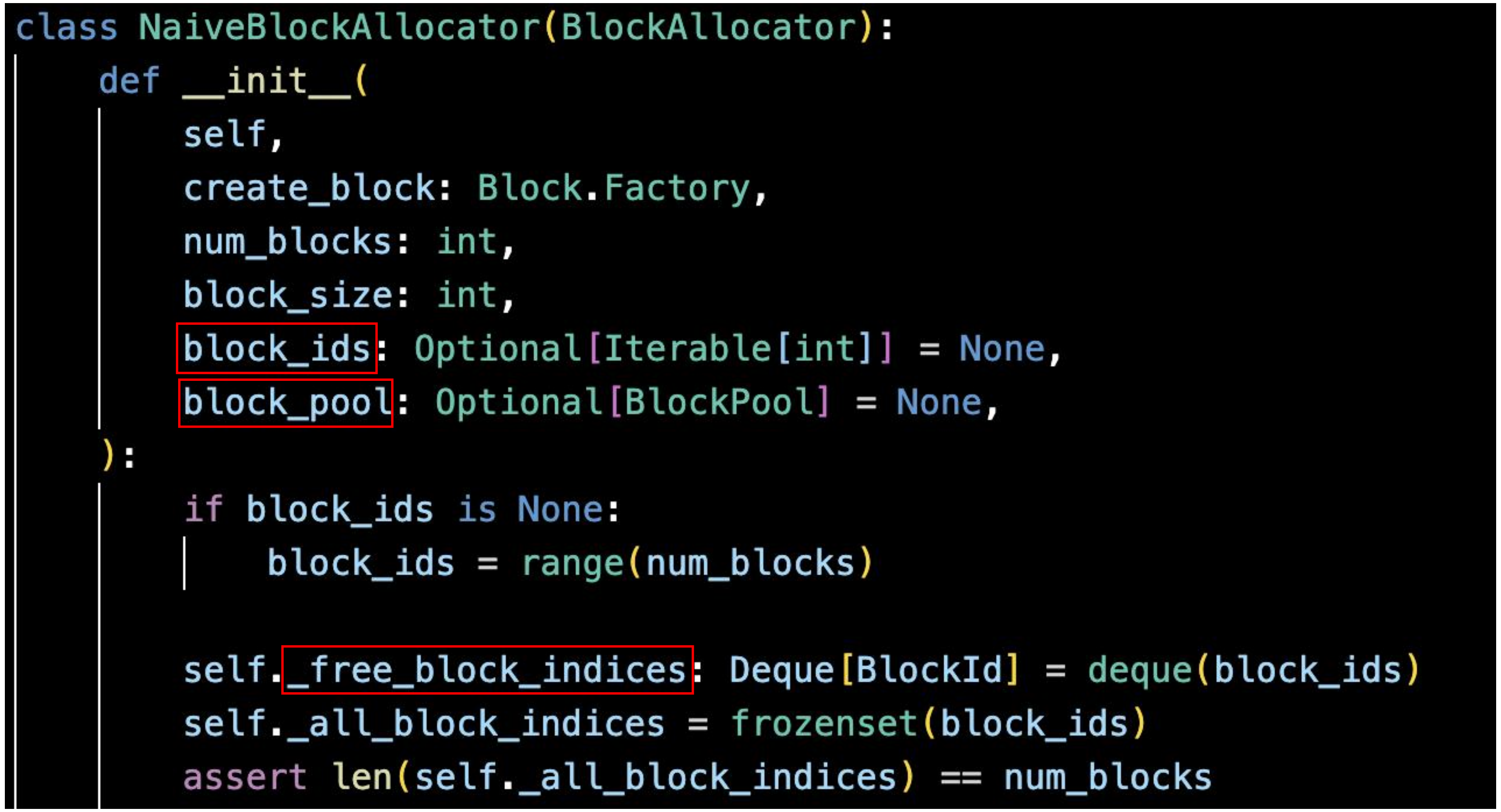
```python
def increase_pool(self):
    """Doubles the internal pool size
    """
    cur_pool_size = self._pool_size
    new_pool_size = cur_pool_size * 2
    self._pool_size = new_pool_size

    self._free_ids += deque(range(cur_pool_size, new_pool_size))

    for i in range(cur_pool_size, new_pool_size):
        self._pool.append(
            self._create_block(prev_block=None,
                               token_ids=[],
                               block_size=self._block_size,
                               allocator=self._allocator,
                               block_id=None))
```

Add/Free block in pool



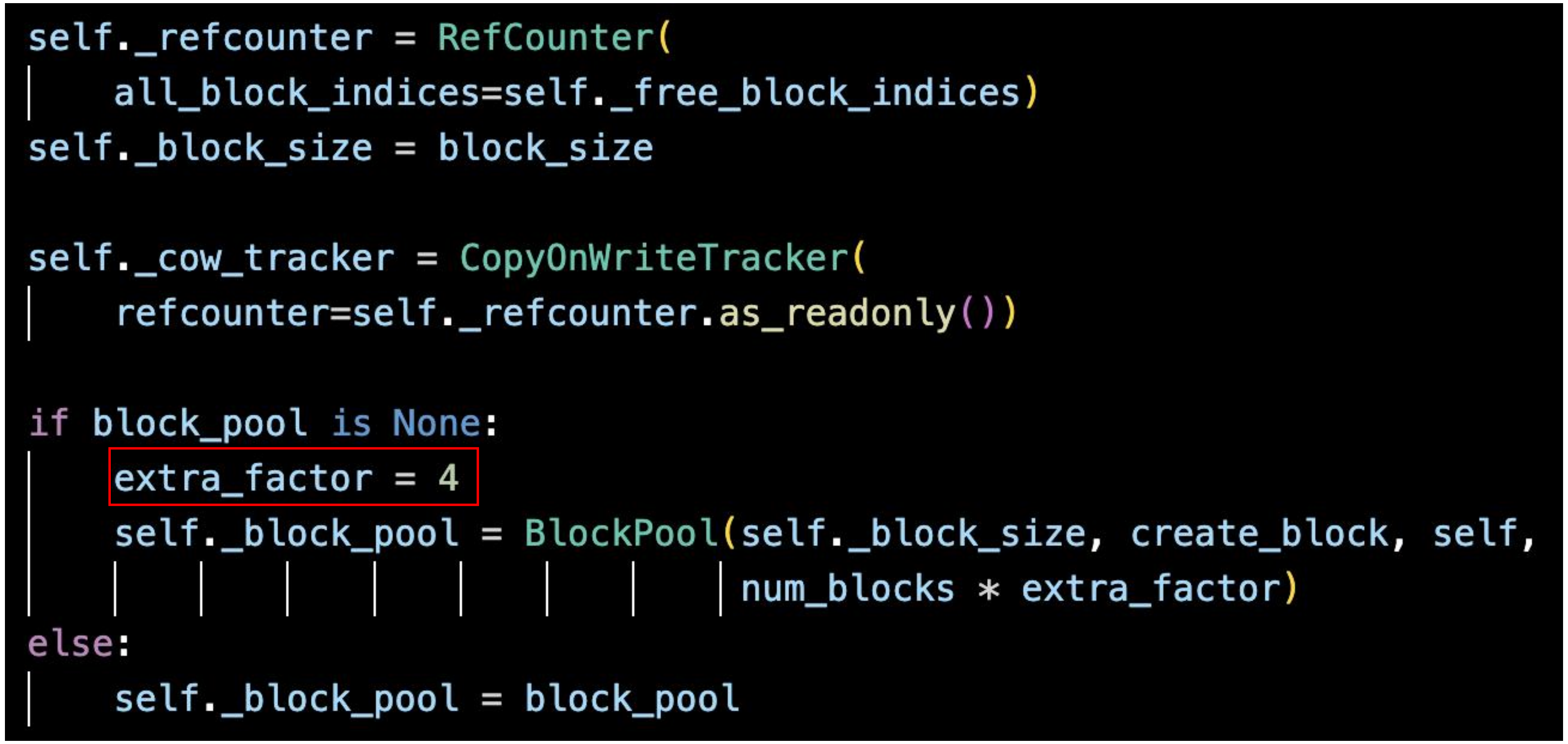
```
def init_block(self, prev_block: Optional[Block], token_ids: List[int],
               block_size: int, physical_block_id: Optional[int]) -> Block:
    if len(self._free_ids) == 0:
        self.increase_pool()
        assert len(self._free_ids) > 0

    pool_id = self._free_ids.popleft()

    block = self._pool[pool_id]
    block.__init__(  # type: ignore[misc]
        prev_block=prev_block,
        token_ids=token_ids,
        block_size=block_size,
        allocator=block._allocator,  # type: ignore[attr-defined]
        block_id=physical_block_id)
    block.pool_id = pool_id  # type: ignore[attr-defined]
    return block

def free_block(self, block: Block) -> None:
    self._free_ids.appendleft(block.pool_id)  # type: ignore[attr-defined]
```

BlockAllocator Initialization



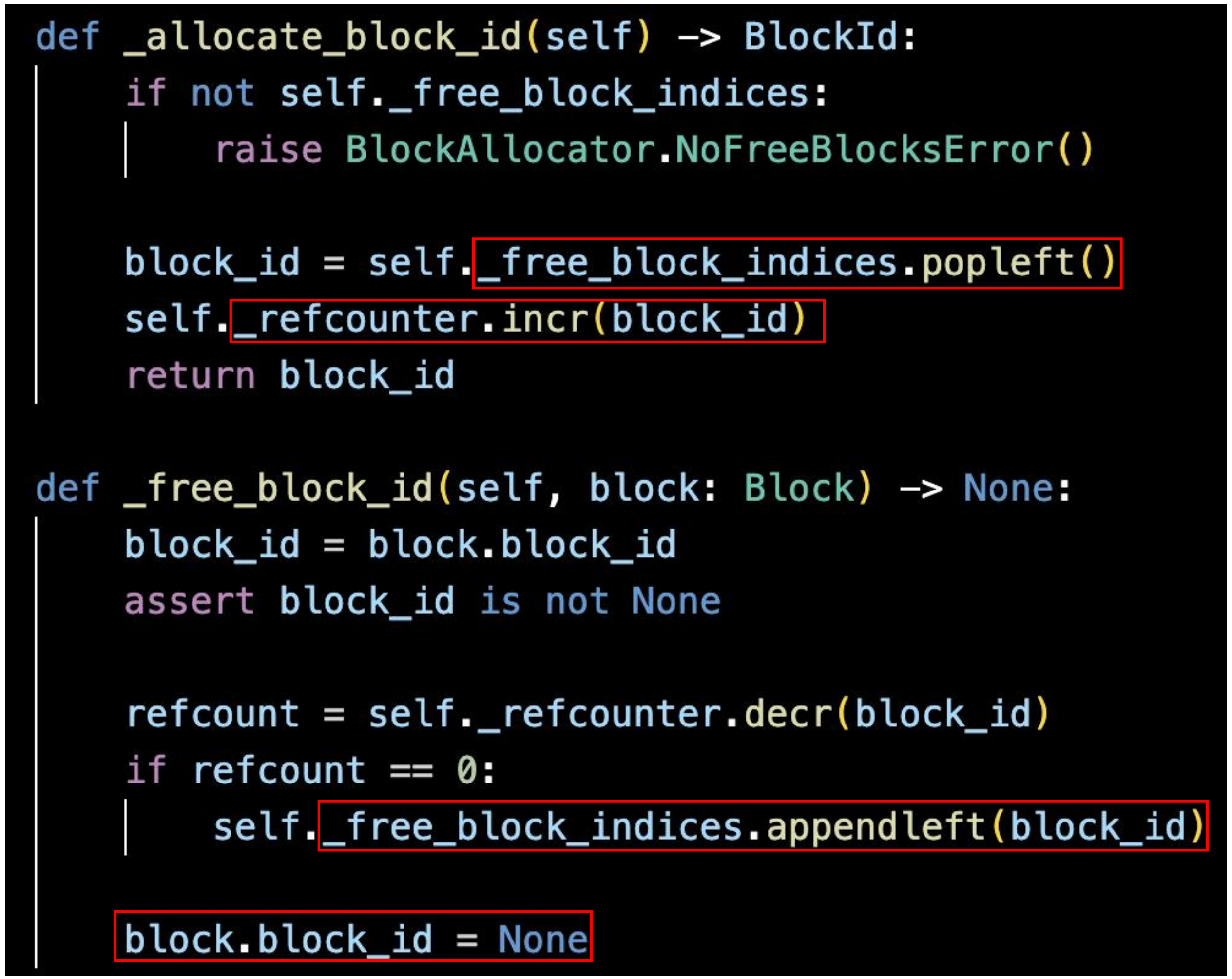
```
class NaiveBlockAllocator(BlockAllocator):
    def __init__(
        self,
        create_block: Block.Factory,
        num_blocks: int,
        block_size: int,
        block_ids: Optional[Iterable[int]] = None,
        block_pool: Optional[BlockPool] = None,
    ):
        if block_ids is None:
            block_ids = range(num_blocks)

        self._free_block_indices: Deque[BlockId] = deque(block_ids)
        self._all_block_indices = frozenset(block_ids)
        assert len(self._all_block_indices) == num_blocks
```

BlockAllocator Initialization



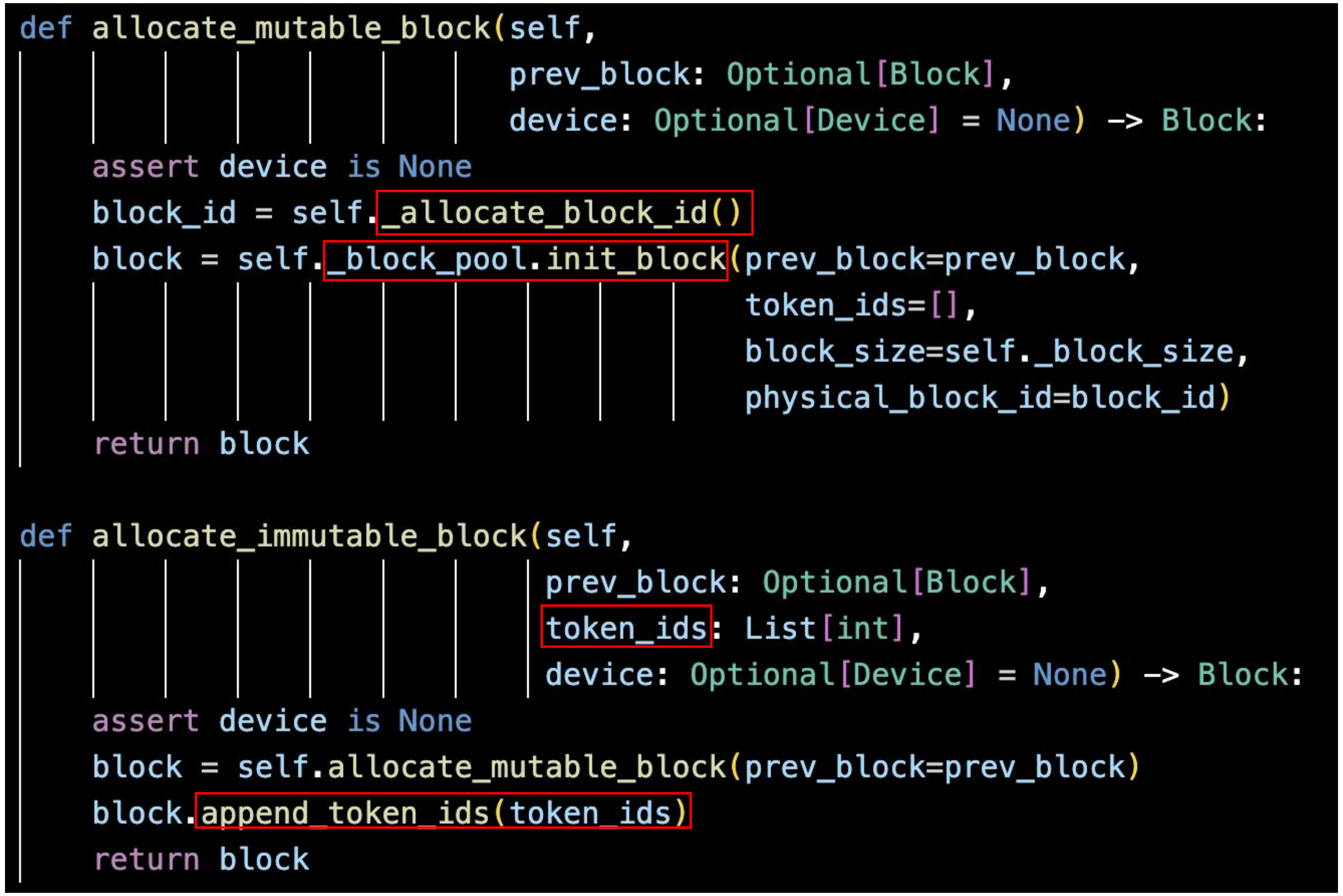
```python
self._refcounter = RefCounter(
    all_block_indices=self._free_block_indices)
self._block_size = block_size

self._cow_tracker = CopyOnWriteTracker(
    refcounter=self._refcounter.as_readonly())

if block_pool is None:
    extra_factor = 4
    self._block_pool = BlockPool(self._block_size, create_block, self,
                                 num_blocks * extra_factor)
else:
    self._block_pool = block_pool
```

Block id operation



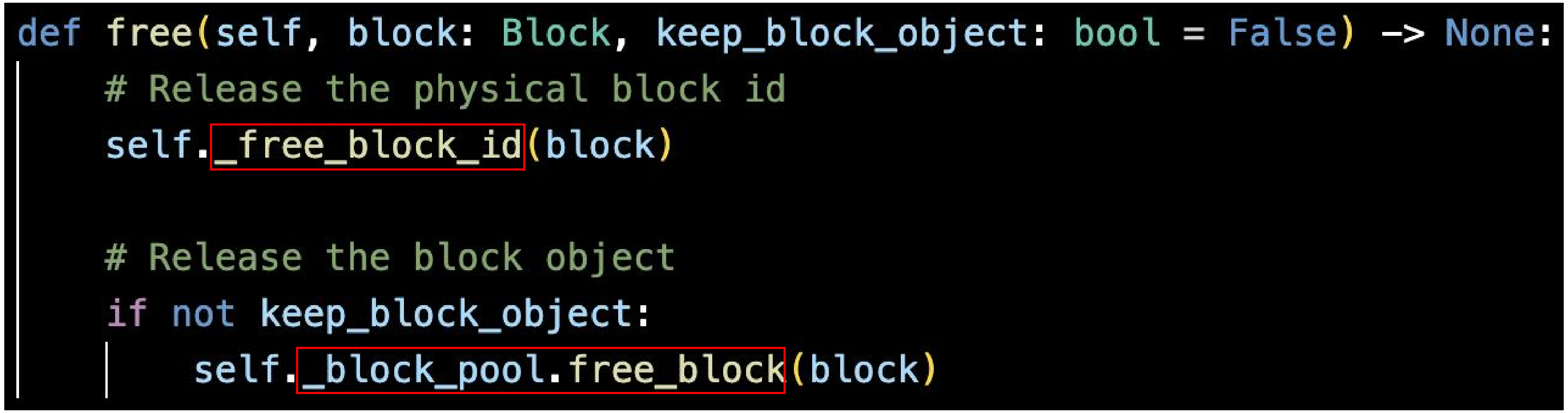
```python
def _allocate_block_id(self) -> BlockId:
    if not self._free_block_indices:
        raise BlockAllocator.NoFreeBlocksError()

    block_id = self._free_block_indices.popleft()
    self._refcounter.incr(block_id)
    return block_id

def _free_block_id(self, block: Block) -> None:
    block_id = block.block_id
    assert block_id is not None

    refcount = self._refcounter.decr(block_id)
    if refcount == 0:
        self._free_block_indices.appendleft(block_id)

    block.block_id = None
```

Allocate block



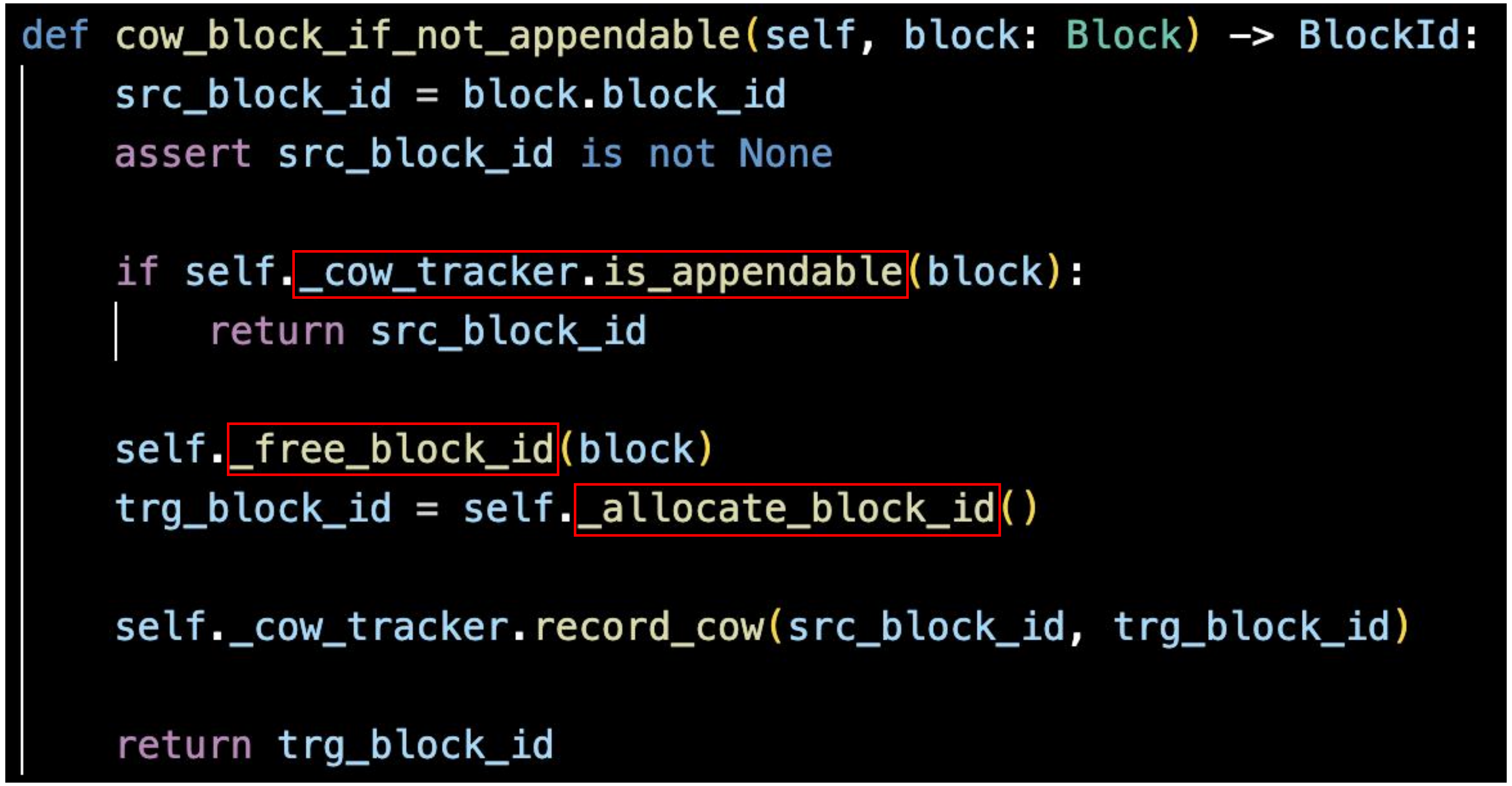
```
def allocate_mutable_block(self,
                           prev_block: Optional[Block],
                           device: Optional[Device] = None) -> Block:
    assert device is None
    block_id = self._allocate_block_id()
    block = self._block_pool.init_block(prev_block=prev_block,
                                        token_ids=[],
                                        block_size=self._block_size,
                                        physical_block_id=block_id)
    return block

def allocate_immutable_block(self,
                             prev_block: Optional[Block],
                             token_ids: List[int],
                             device: Optional[Device] = None) -> Block:
    assert device is None
    block = self.allocate_mutable_block(prev_block=prev_block)
    block.append_token_ids(token_ids)
    return block
```

Free block



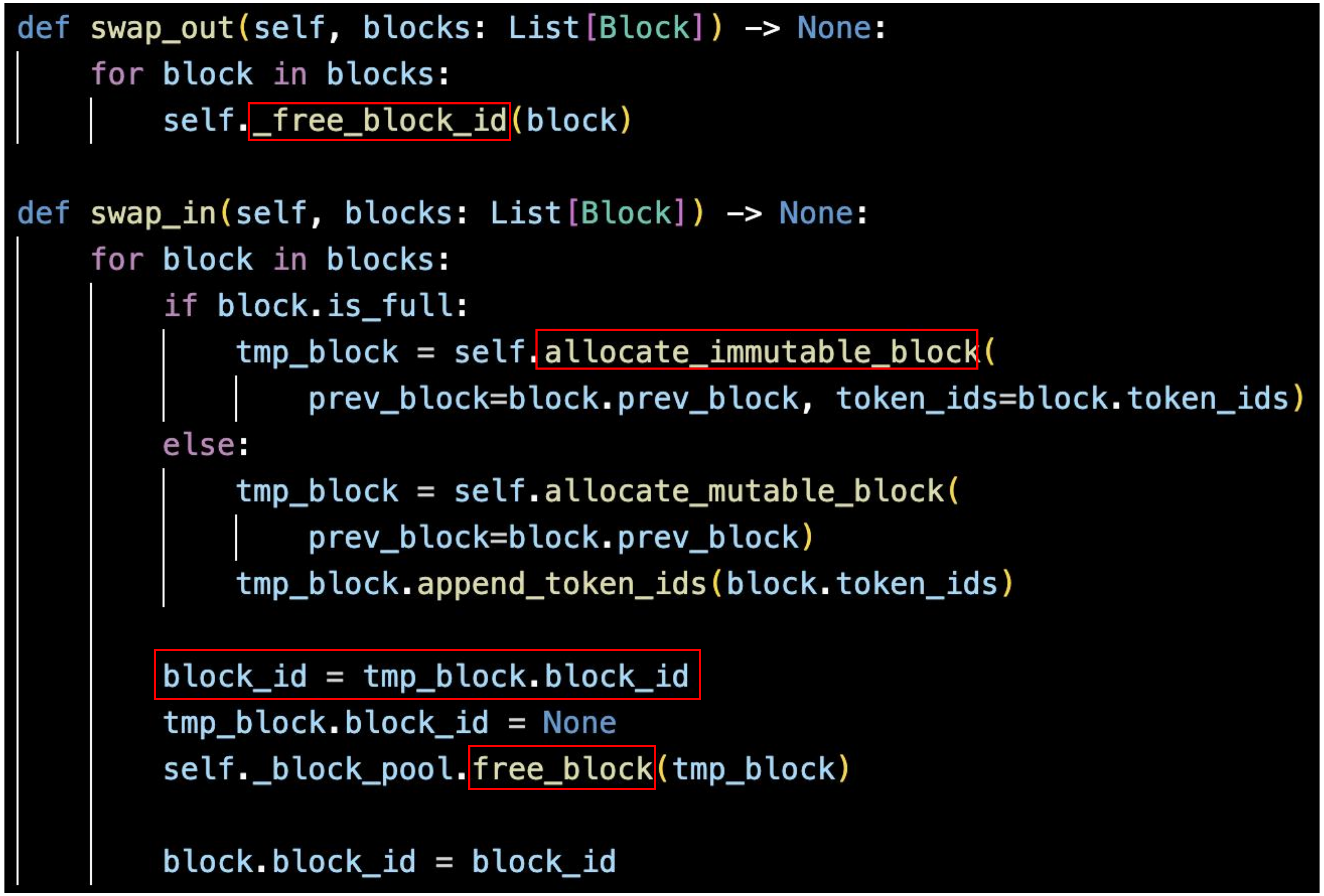
```python
def free(self, block: Block, keep_block_object: bool = False) -> None:
    # Release the physical block id
    self._free_block_id(block)

    # Release the block object
    if not keep_block_object:
        self._block_pool.free_block(block)
```

Copy on write



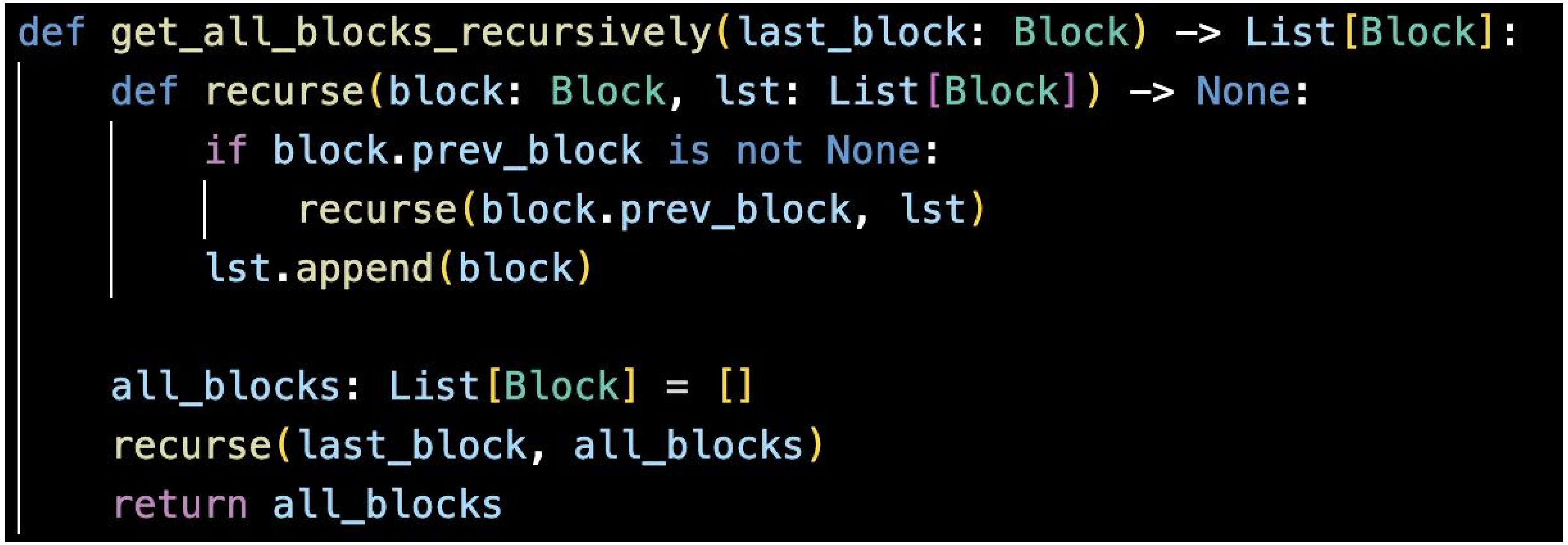
```python
def cow_block_if_not_appendable(self, block: Block) -> BlockId:
    src_block_id = block.block_id
    assert src_block_id is not None

    if self._cow_tracker.is_appendable(block):
        return src_block_id

    self._free_block_id(block)
    trg_block_id = self._allocate_block_id()

    self._cow_tracker.record_cow(src_block_id, trg_block_id)

    return trg_block_id
```

Swap in/out



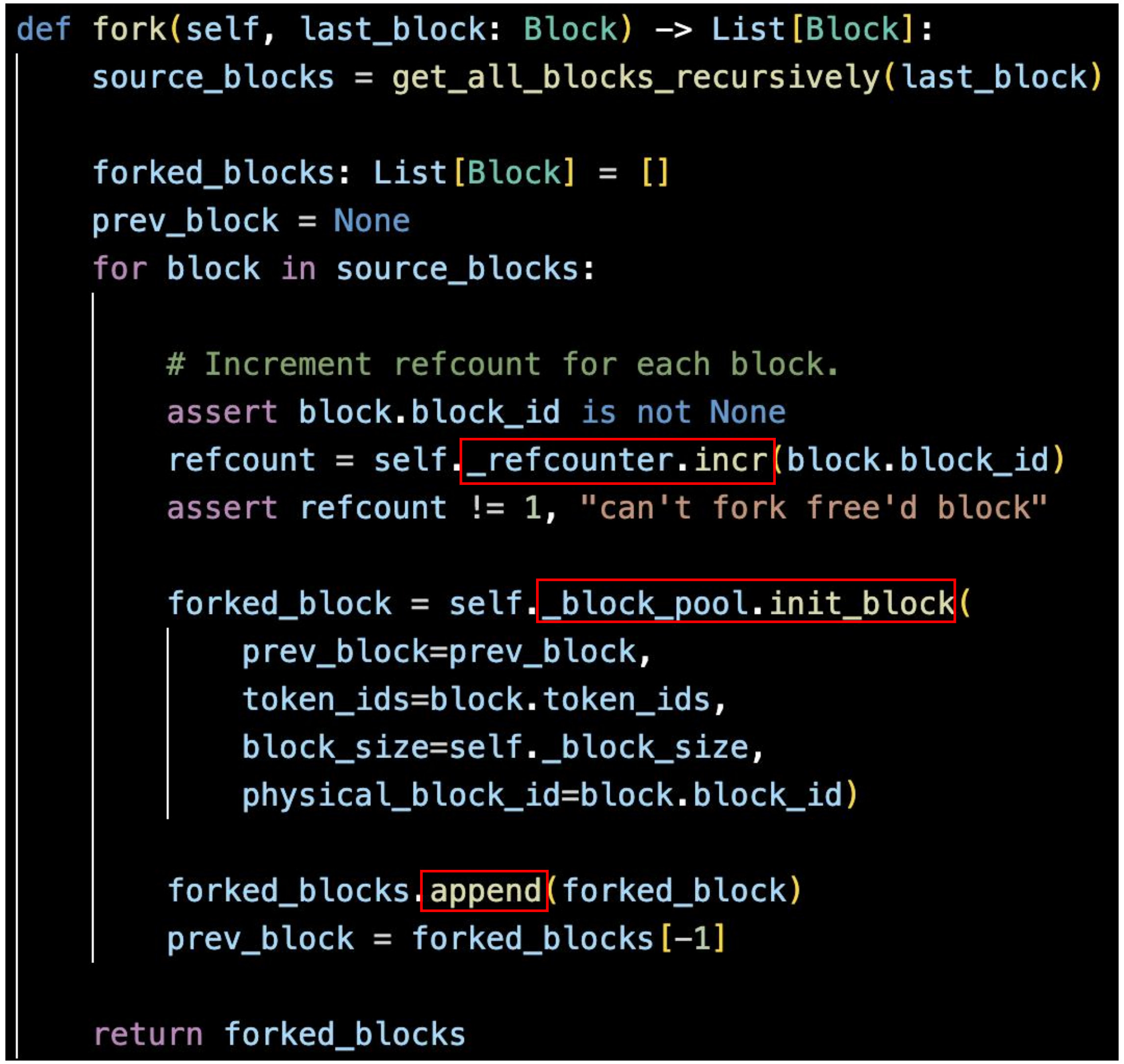
```python
def swap_out(self, blocks: List[Block]) -> None:
    for block in blocks:
        self._free_block_id(block)

def swap_in(self, blocks: List[Block]) -> None:
    for block in blocks:
        if block.is_full:
            tmp_block = self.allocate_immutable_block(
                prev_block=block.prev_block, token_ids=block.token_ids)
        else:
            tmp_block = self.allocate_mutable_block(
                prev_block=block.prev_block)
            tmp_block.append_token_ids(block.token_ids)

        block_id = tmp_block.block_id
        tmp_block.block_id = None
        self._block_pool.free_block(tmp_block)

        block.block_id = block_id
```

How to copy sequence?



```python
def get_all_blocks_recursively(last_block: Block) -> List[Block]:
    def recurse(block: Block, lst: List[Block]) -> None:
        if block.prev_block is not None:
            recurse(block.prev_block, lst)
        lst.append(block)

    all_blocks: List[Block] = []
    recurse(last_block, all_blocks)
    return all_blocks
```

Fork sequence



```python
def fork(self, last_block: Block) -> List[Block]:
    source_blocks = get_all_blocks_recursively(last_block)

    forked_blocks: List[Block] = []
    prev_block = None
    for block in source_blocks:

        # Increment refcount for each block.
        assert block.block_id is not None
        refcount = self._refcounter.incr(block.block_id)
        assert refcount != 1, "can't fork free'd block"

        forked_block = self._block_pool.init_block(
            prev_block=prev_block,
            token_ids=block.token_ids,
            block_size=self._block_size,
            physical_block_id=block.block_id)

        forked_blocks.append(forked_block)
        prev_block = forked_blocks[-1]

    return forked_blocks
```

Thanks!

Q & A