CS 61A Lecture 9

Friday, September 16

The Sequence Abstraction

The Sequence Abstraction

red, orange, yellow, green, blue, indigo, violet.

The Sequence Abstraction

red, orange, yellow, green, blue, indigo, violet.

There isn't just one sequence type (in Python or in general)

The Sequence Abstraction

red, orange, yellow, green, blue, indigo, violet.

There isn't just one sequence type (in Python or in general)

This abstraction is a collection of behaviors:

The Sequence Abstraction

red, orange, yellow, green, blue, indigo, violet.

There isn't just one sequence type (in Python or in general)

This abstraction is a collection of behaviors:

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

The Sequence Abstraction

red, orange, yellow, green, blue, indigo, violet.

0 , 1 , 2 , 3 , 4 , 5 , 6 .

There isn't just one sequence type (in Python or in general)

This abstraction is a collection of behaviors:

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

The Sequence Abstraction

red, orange, yellow, green, blue, indigo, violet.

0 , 1 , 2 , 3 , 4 , 5 , 6 .

There isn't just one sequence type (in Python or in general)

This abstraction is a collection of behaviors:

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

The sequence abstraction is shared among several types.

Tuples are Sequences

(Demo)

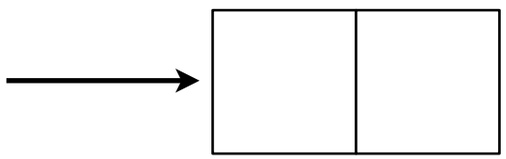
Box-and-Pointer Notation for Nested Pairs

Box-and-Pointer Notation for Nested Pairs

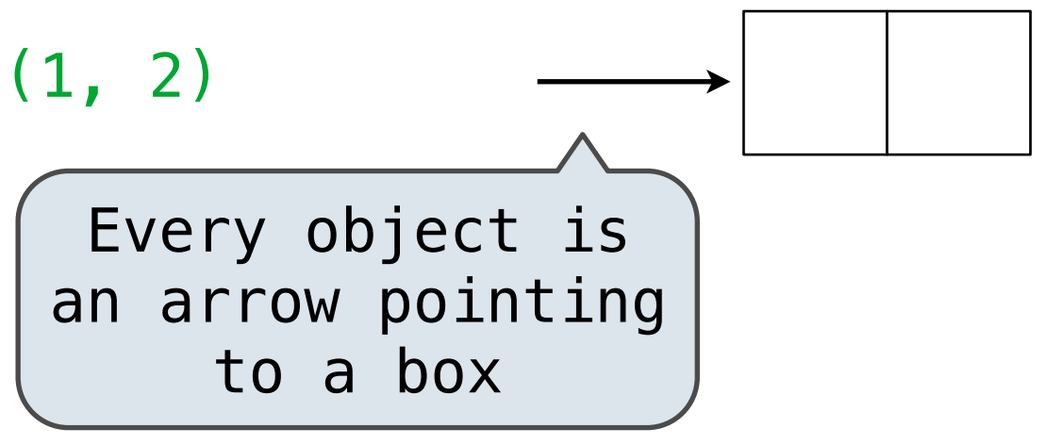
```
(1, 2)
```

Box-and-Pointer Notation for Nested Pairs

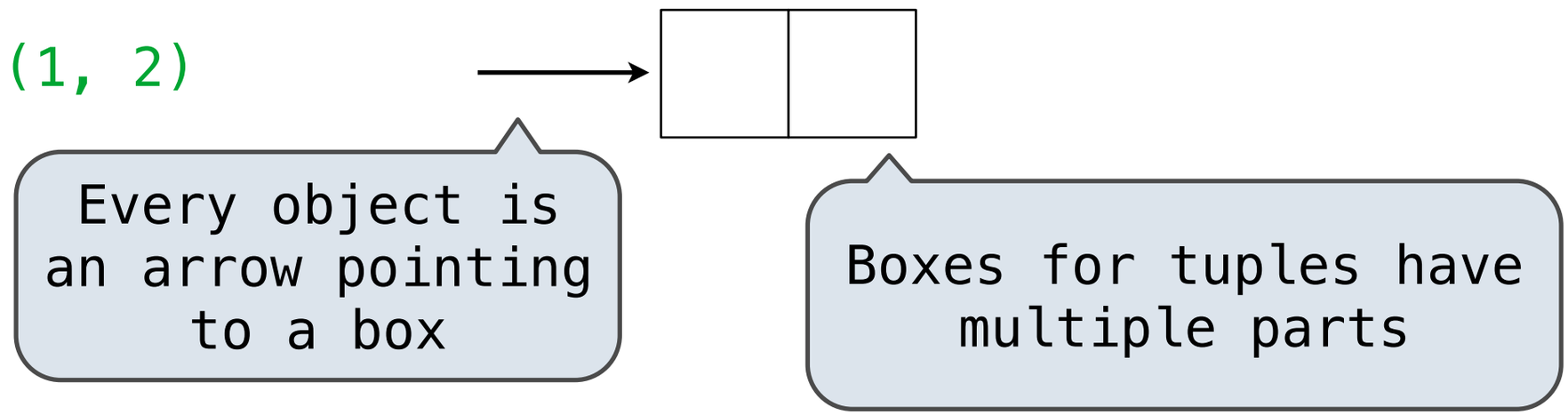
```
(1, 2)
```



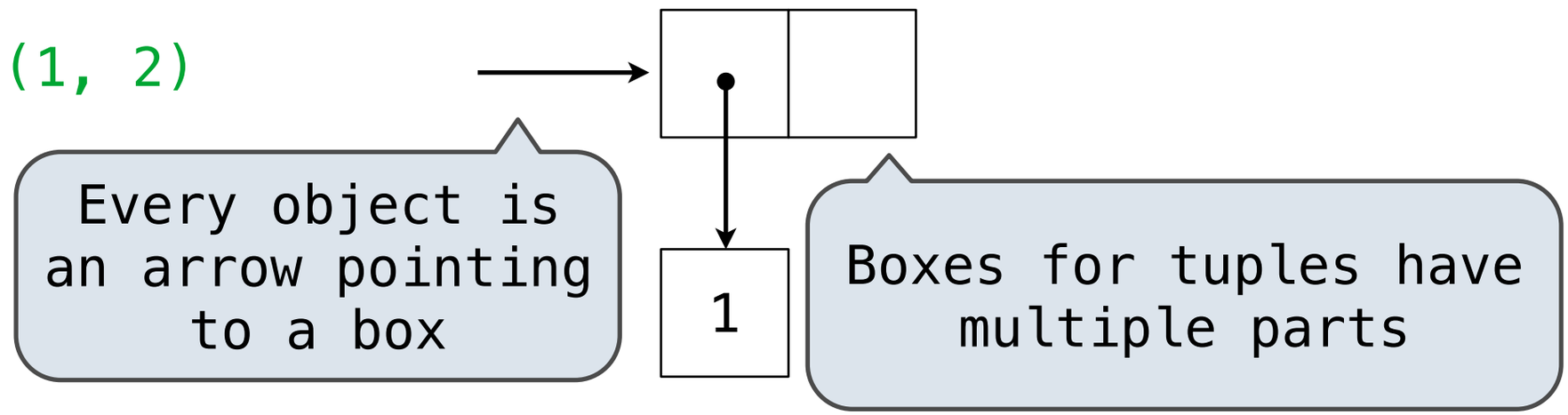
Box-and-Pointer Notation for Nested Pairs



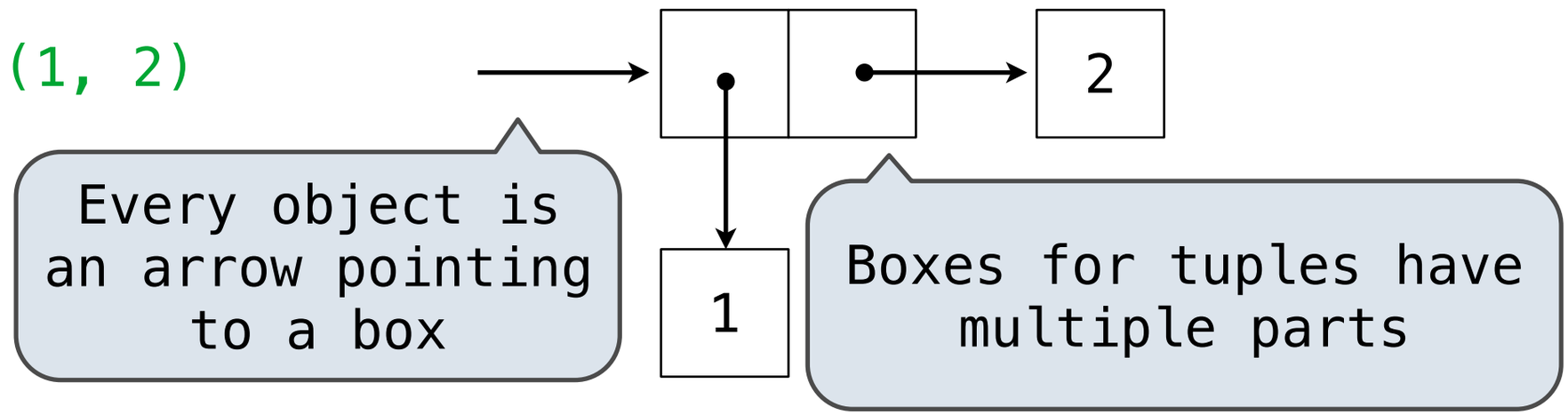
Box-and-Pointer Notation for Nested Pairs



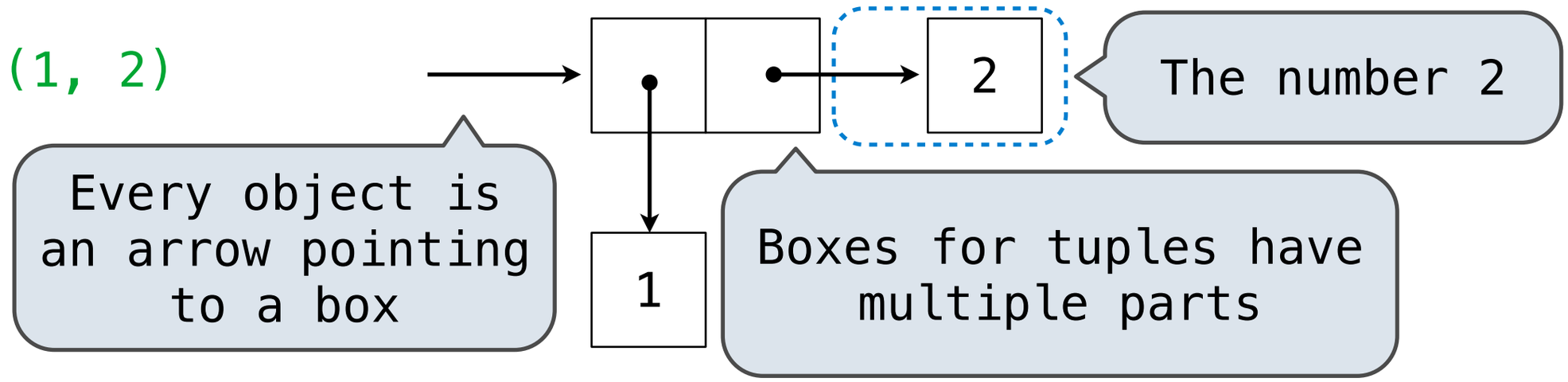
Box-and-Pointer Notation for Nested Pairs



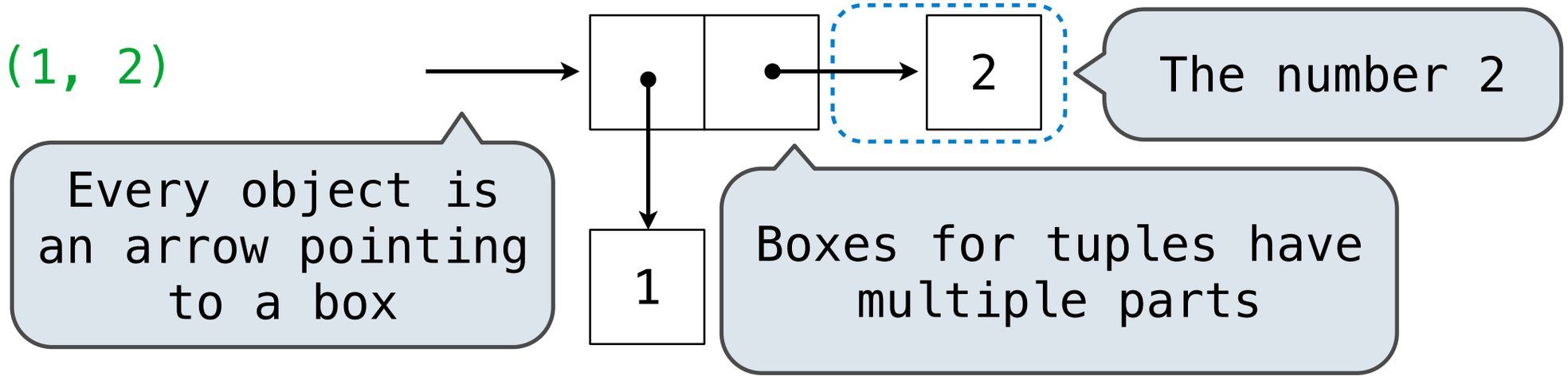
Box-and-Pointer Notation for Nested Pairs



Box-and-Pointer Notation for Nested Pairs

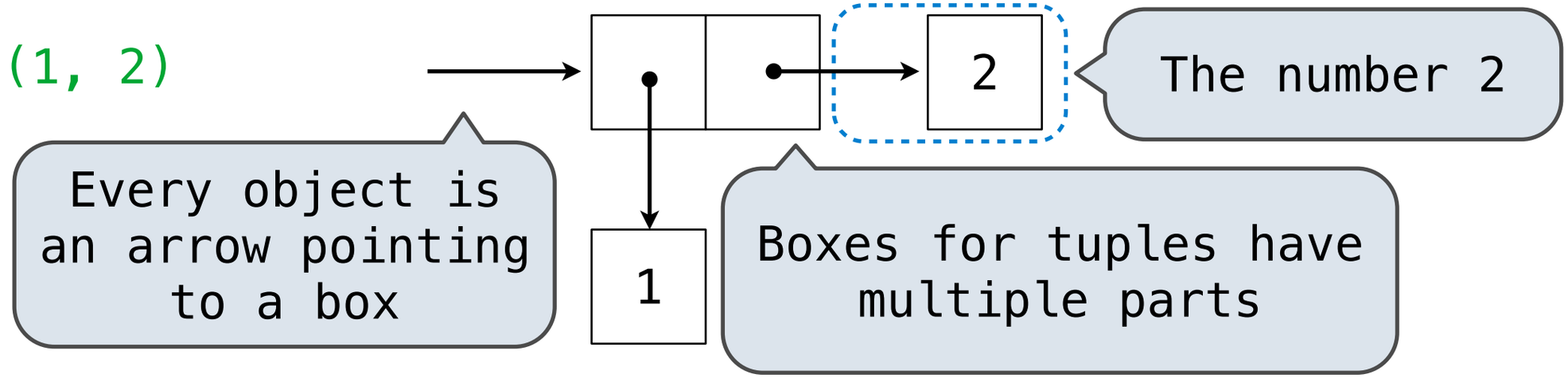


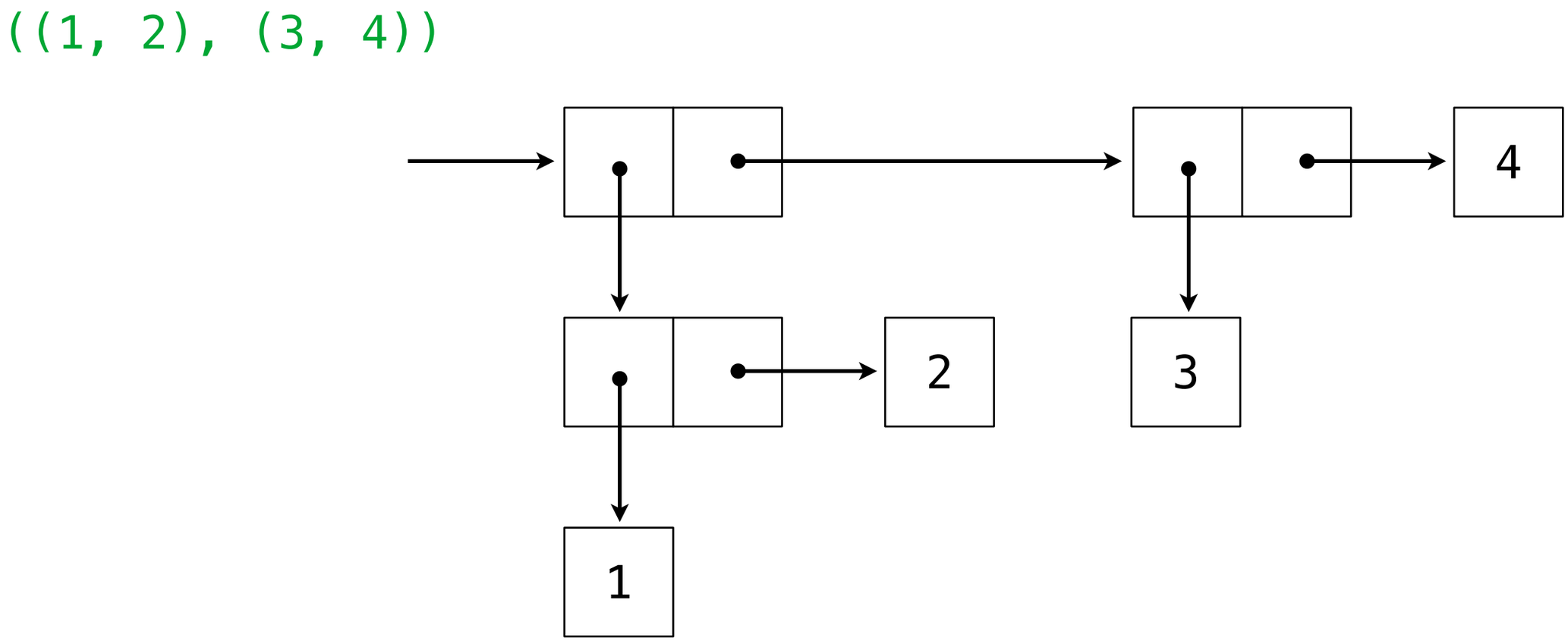
Box-and-Pointer Notation for Nested Pairs



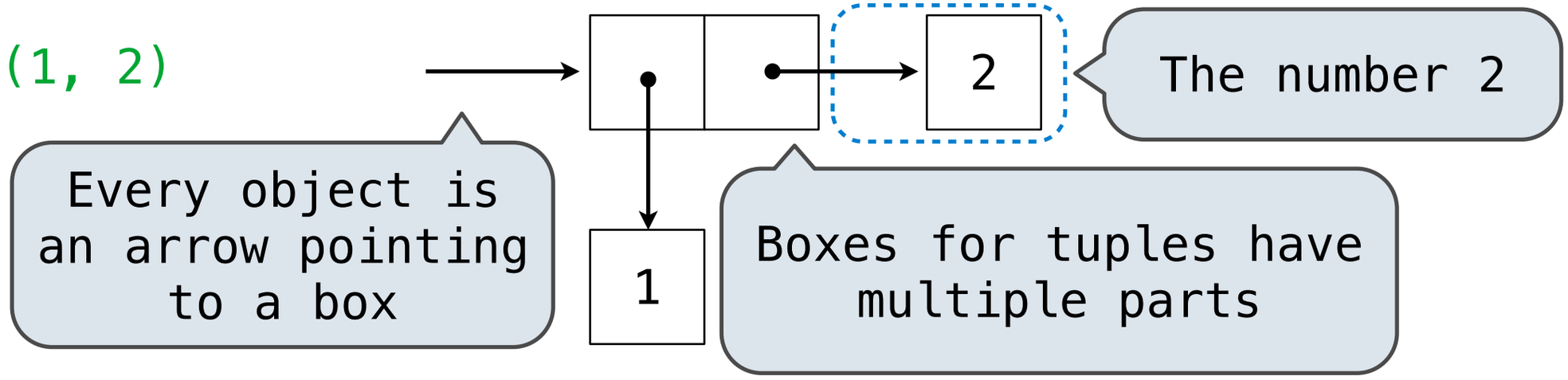

```
((1, 2), (3, 4))
```

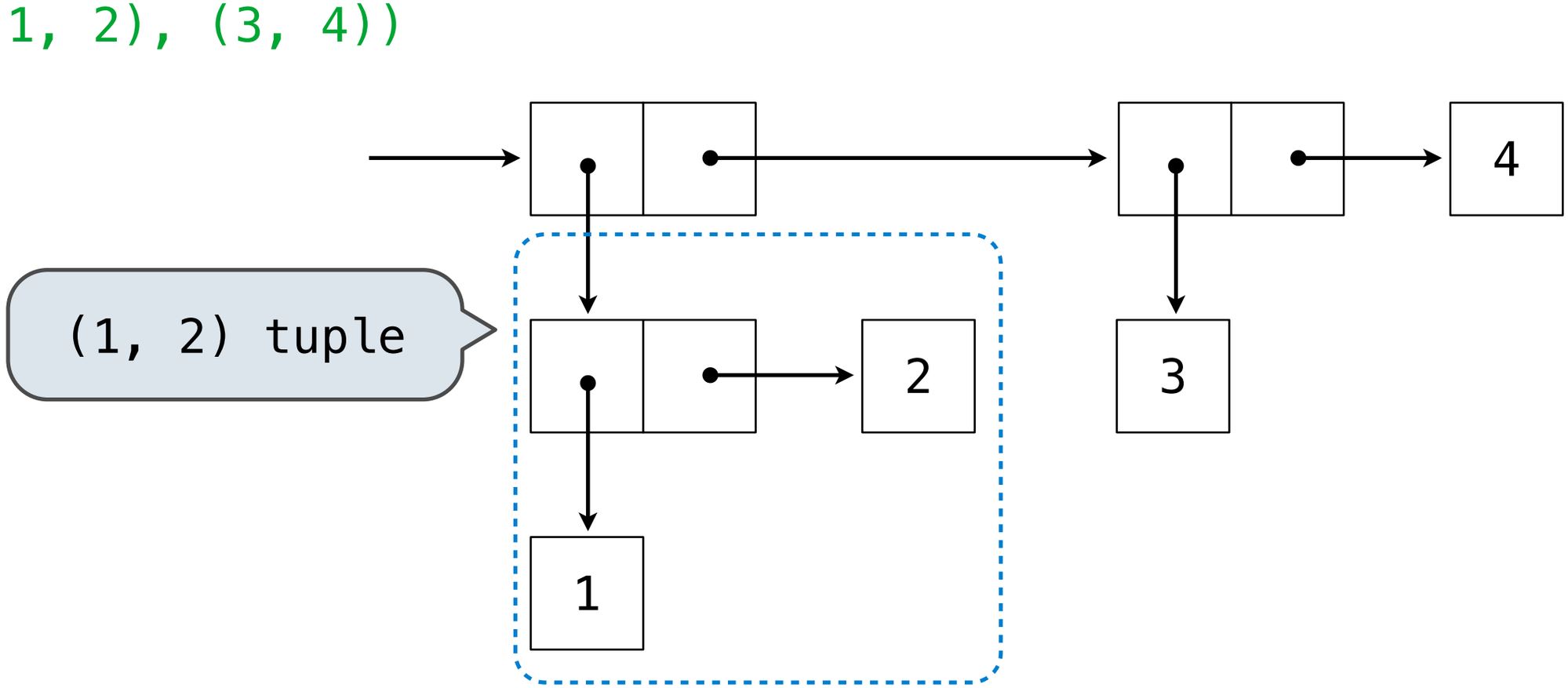
Box-and-Pointer Notation for Nested Pairs





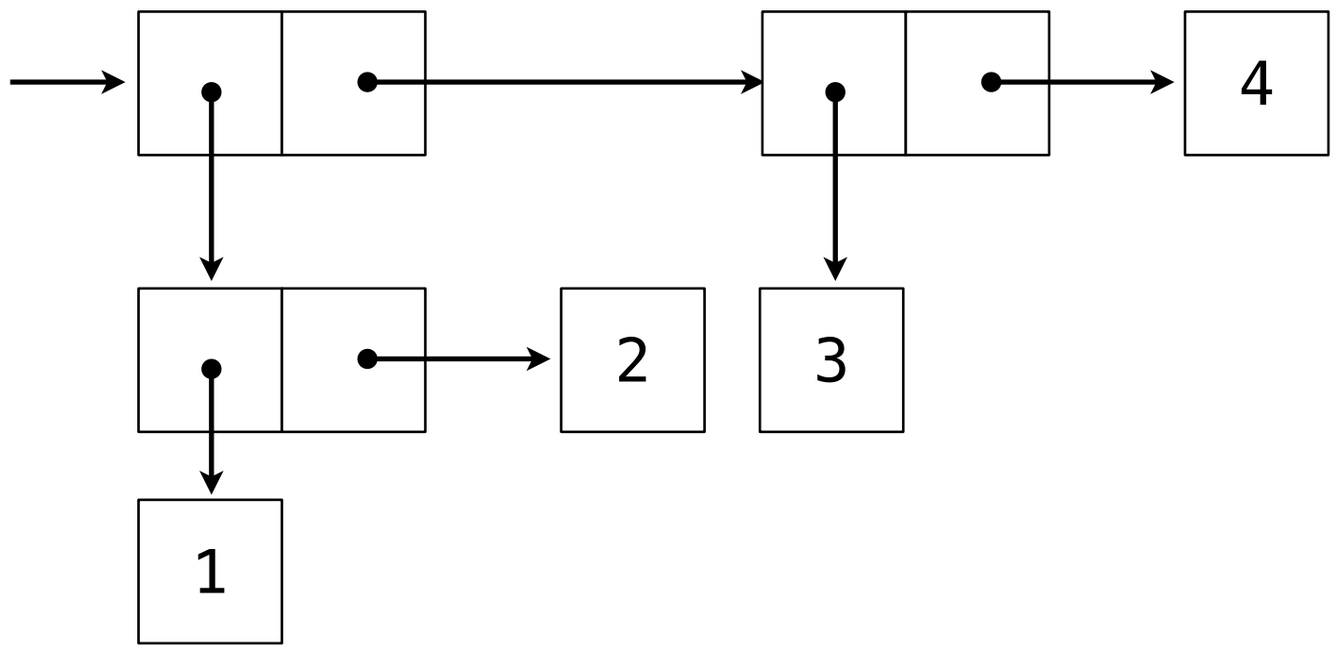
Box-and-Pointer Notation for Nested Pairs





Alternative Nested Structures

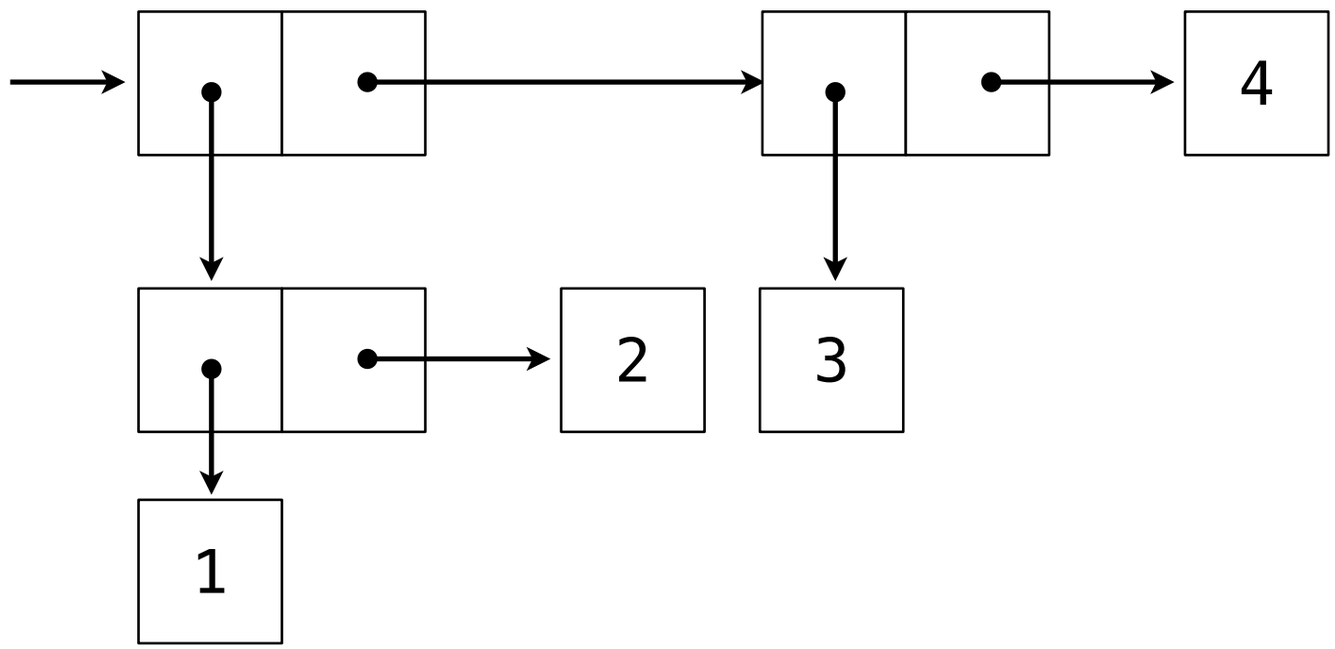
```
((1, 2), (3, 4))
```



Alternative Nested Structures

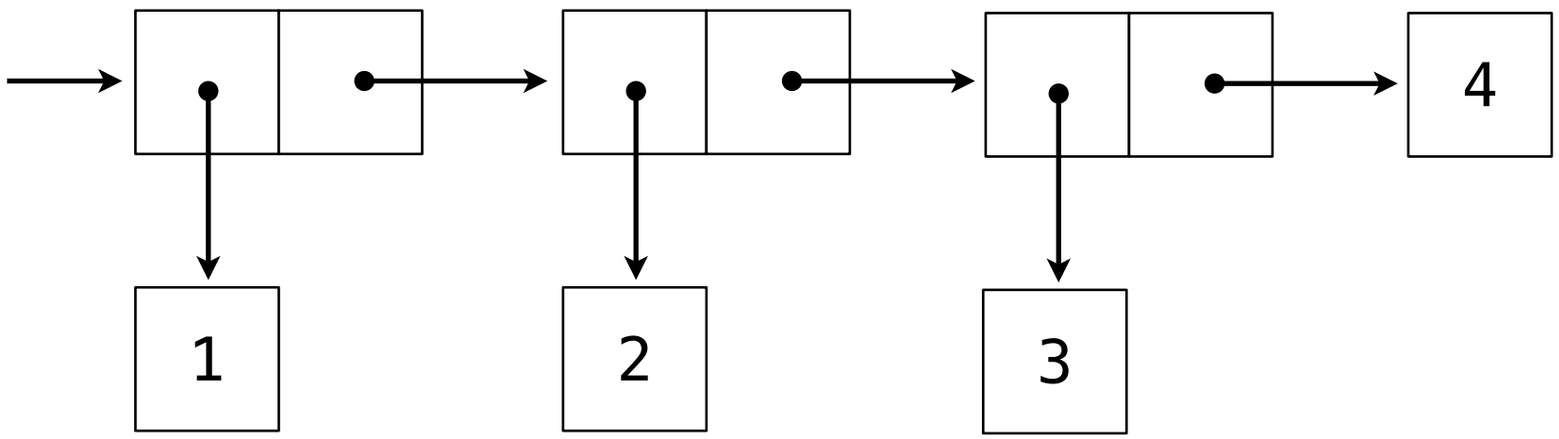
```
(1, (2, (3, 4)))
```

```
((1, 2), (3, 4))
```

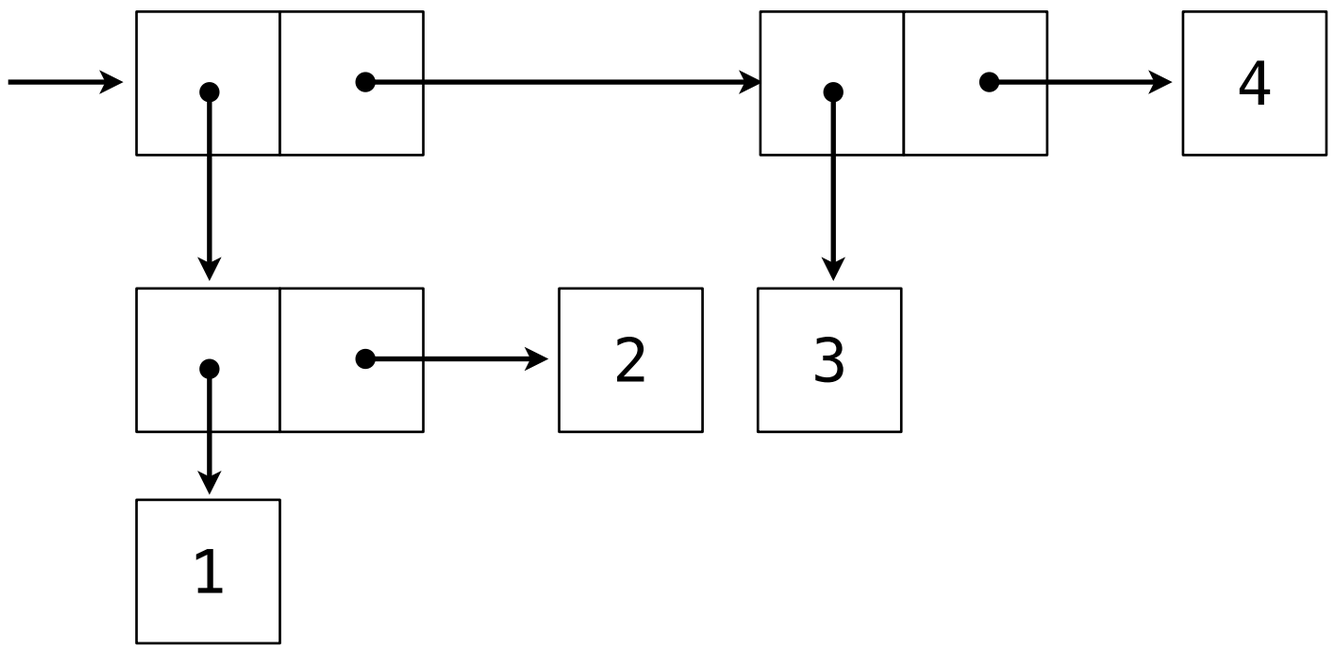


Alternative Nested Structures

```
(1, (2, (3, 4)))
```

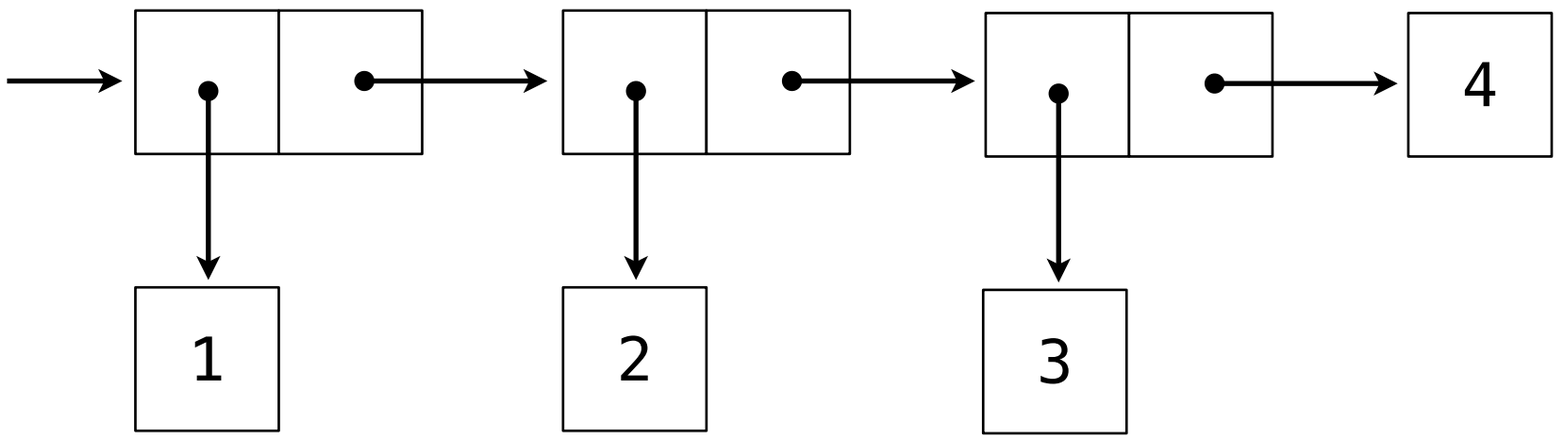



```
((1, 2), (3, 4))
```



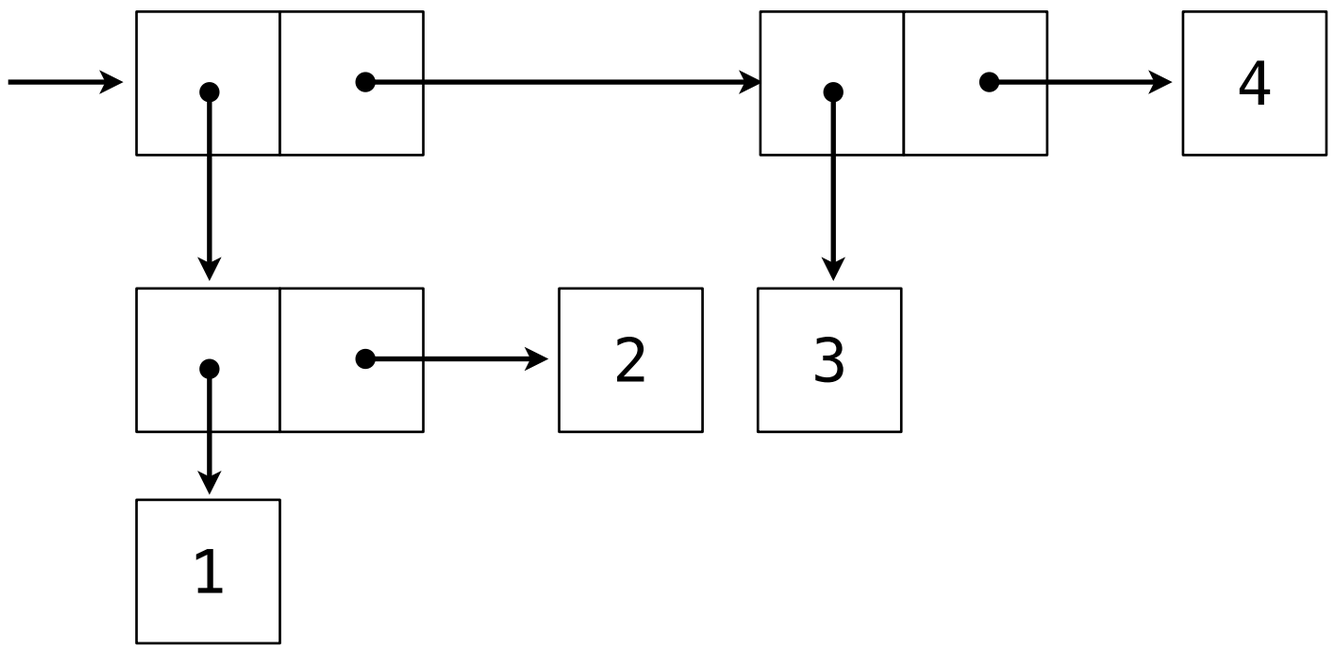
Alternative Nested Structures

```
(1, (2, (3, 4)))
```



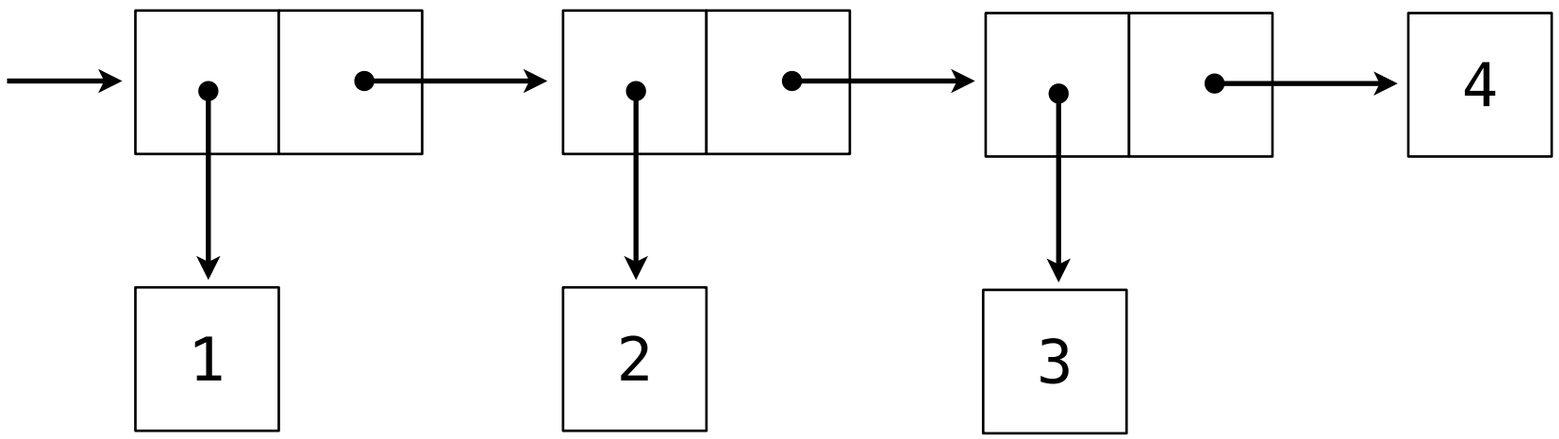

```
(1, 2, 3, 4)
```

```
((1, 2), (3, 4))
```

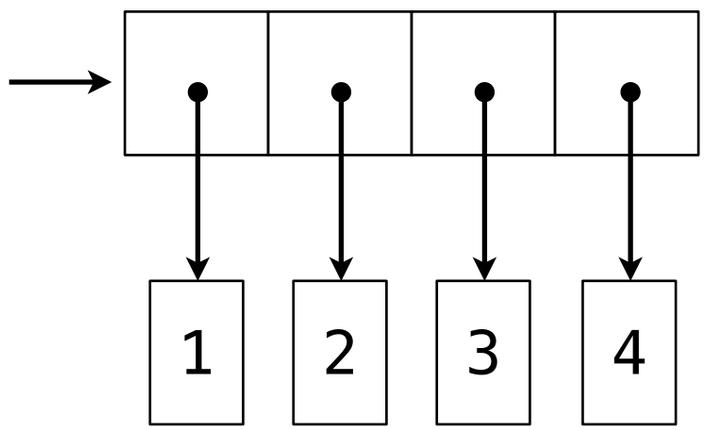


Alternative Nested Structures

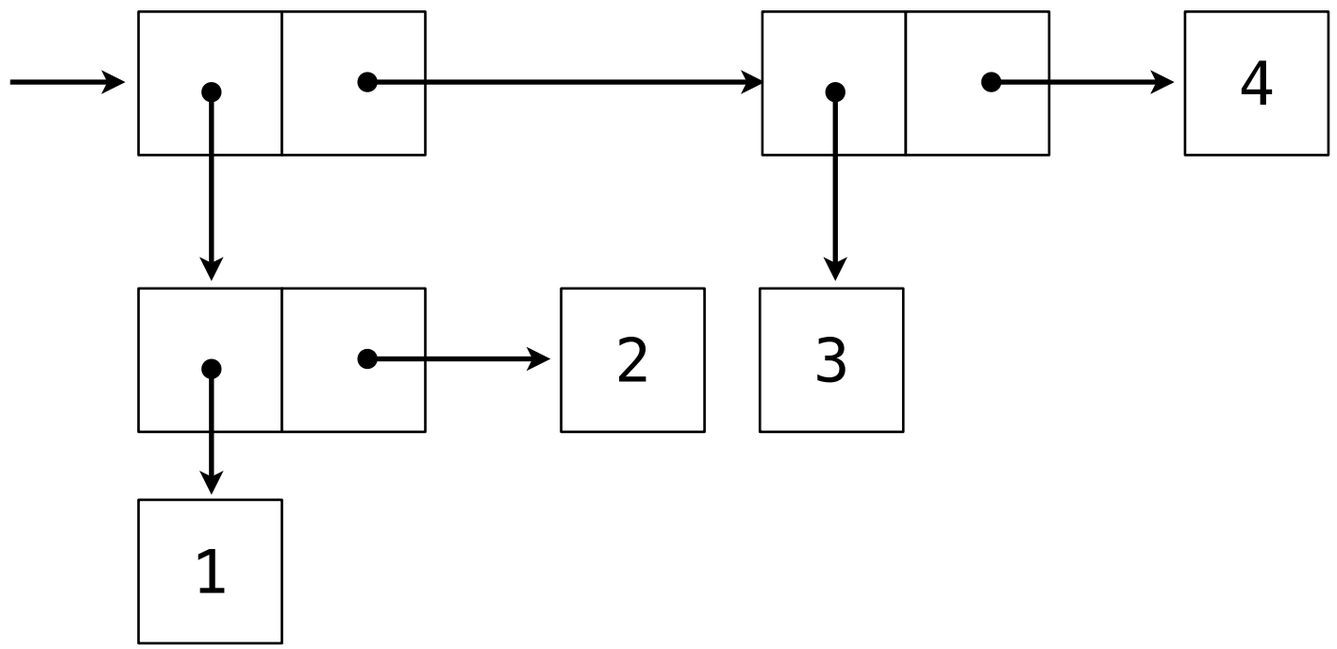
```
(1, (2, (3, 4)))
```




```
(1, 2, 3, 4)
```




```
((1, 2), (3, 4))
```



The Closure Property of Data Types

The Closure Property of Data Types

- A method for combining data values satisfies the *closure property* if:

The Closure Property of Data Types

- A method for combining data values satisfies the *closure property* if:
- The result of combination can itself be combined using the same method.

The Closure Property of Data Types

- A method for combining data values satisfies the *closure property* if:
- The result of combination can itself be combined using the same method.
- Closure is the key to power in any means of combination because it permits us to create hierarchical structures.

The Closure Property of Data Types

- A method for combining data values satisfies the *closure property* if:
- The result of combination can itself be combined using the same method.
- Closure is the key to power in any means of combination because it permits us to create hierarchical structures.
- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on.

The Closure Property of Data Types

- A method for combining data values satisfies the *closure property* if:
- The result of combination can itself be combined using the same method.
- Closure is the key to power in any means of combination because it permits us to create hierarchical structures.
- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on.

Tuples can contain tuples as elements

Recursive Lists

Recursive Lists

Constructor:

```
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
```

Recursive Lists

Constructor:

```
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
```

Selectors:

```
def first(s):
    """Return the first element of a recursive list s."""

def rest(s):
    """Return the rest of the elements of a recursive list s."""
```

Recursive Lists

Constructor:

```
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
```

Selectors:

```
def first(s):
    """Return the first element of a recursive list s."""

def rest(s):
    """Return the rest of the elements of a recursive list s."""
```

Behavior condition(s):

Recursive Lists

Constructor:

```
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
```

Selectors:

```
def first(s):
    """Return the first element of a recursive list s."""

def rest(s):
    """Return the rest of the elements of a recursive list s."""
```

Behavior condition(s):

If a recursive list `s` was constructed from first element `f` and recursive list `r`, then

Recursive Lists

Constructor:

```
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
```

Selectors:

```
def first(s):
    """Return the first element of a recursive list s."""

def rest(s):
    """Return the rest of the elements of a recursive list s."""
```

Behavior condition(s):

If a recursive list `s` was constructed from first element `f` and recursive list `r`, then

- `first(s)` returns `f`, and

Recursive Lists

Constructor:

```
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
```

Selectors:

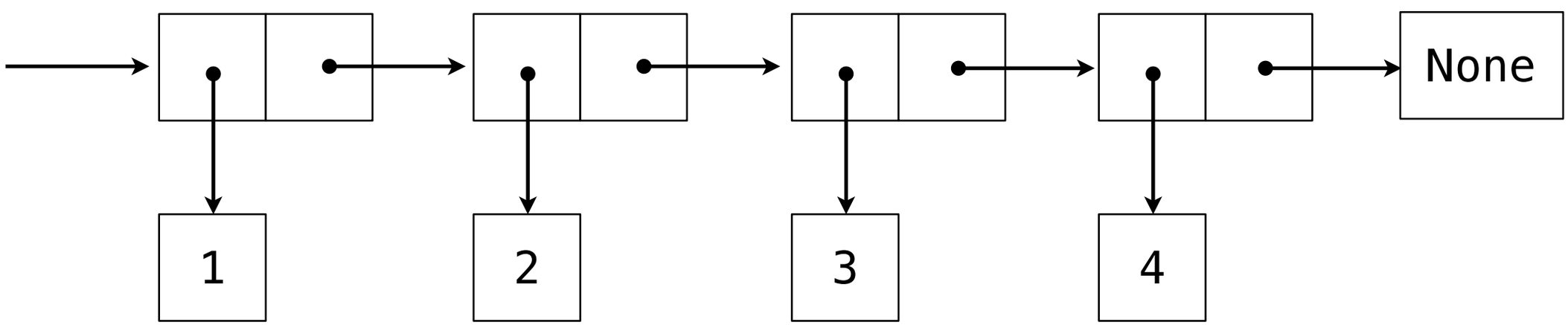
```
def first(s):
    """Return the first element of a recursive list s."""

def rest(s):
    """Return the rest of the elements of a recursive list s."""
```

Behavior condition(s):

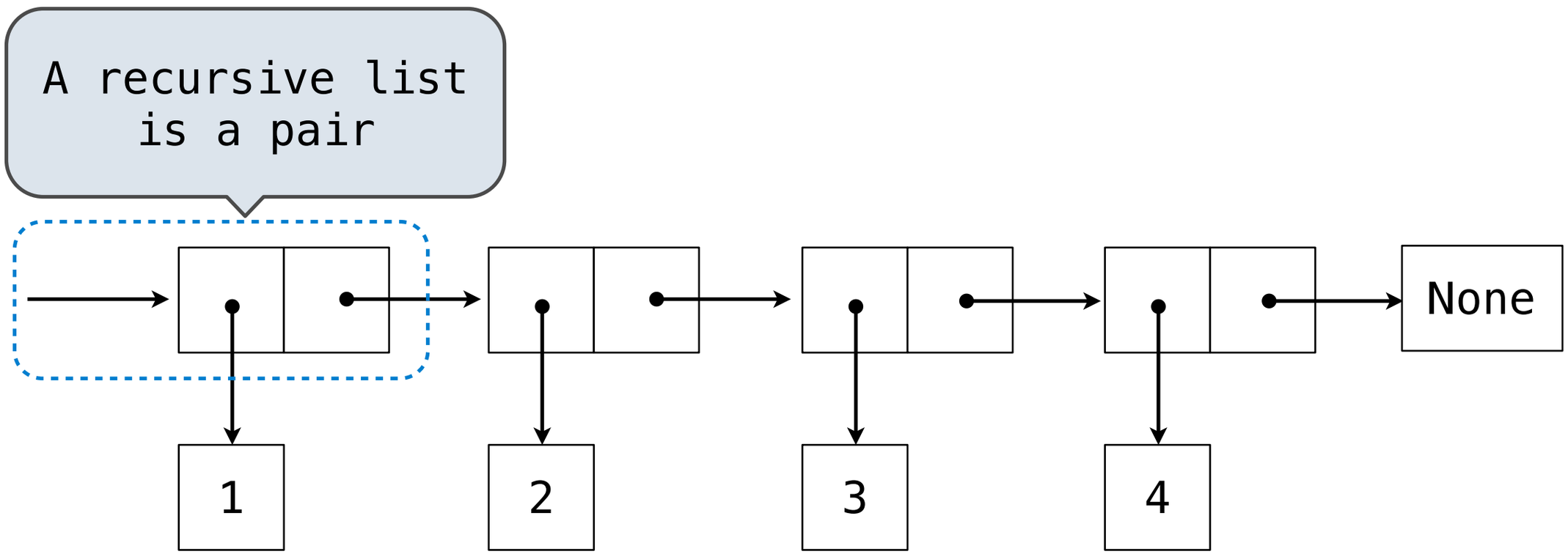
If a recursive list `s` was constructed from first element `f` and recursive list `r`, then

- `first(s)` returns `f`, and
- `rest(s)` returns `r`, which is a recursive list.

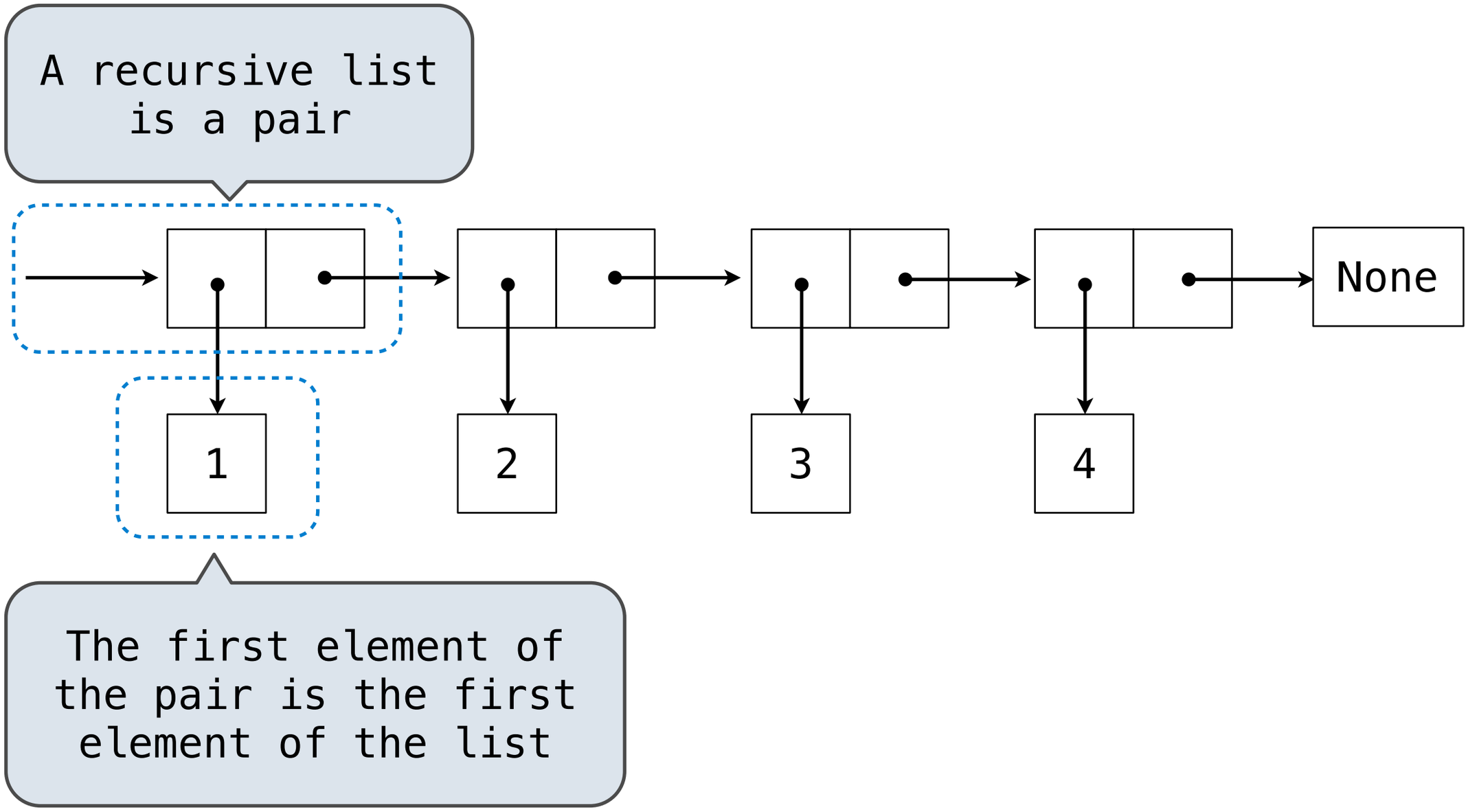
Implementing Recursive Lists with Pairs



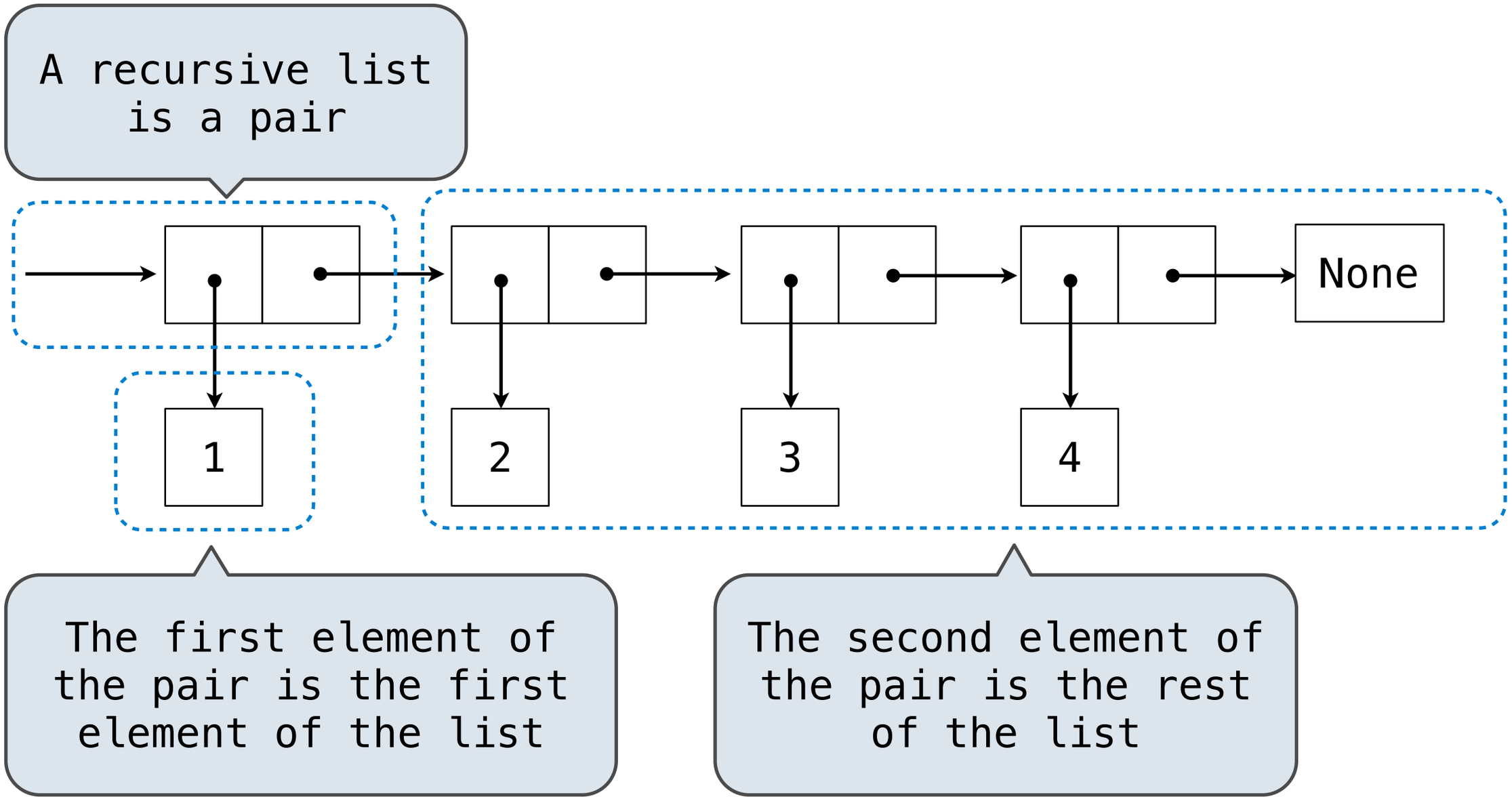
Implementing Recursive Lists with Pairs



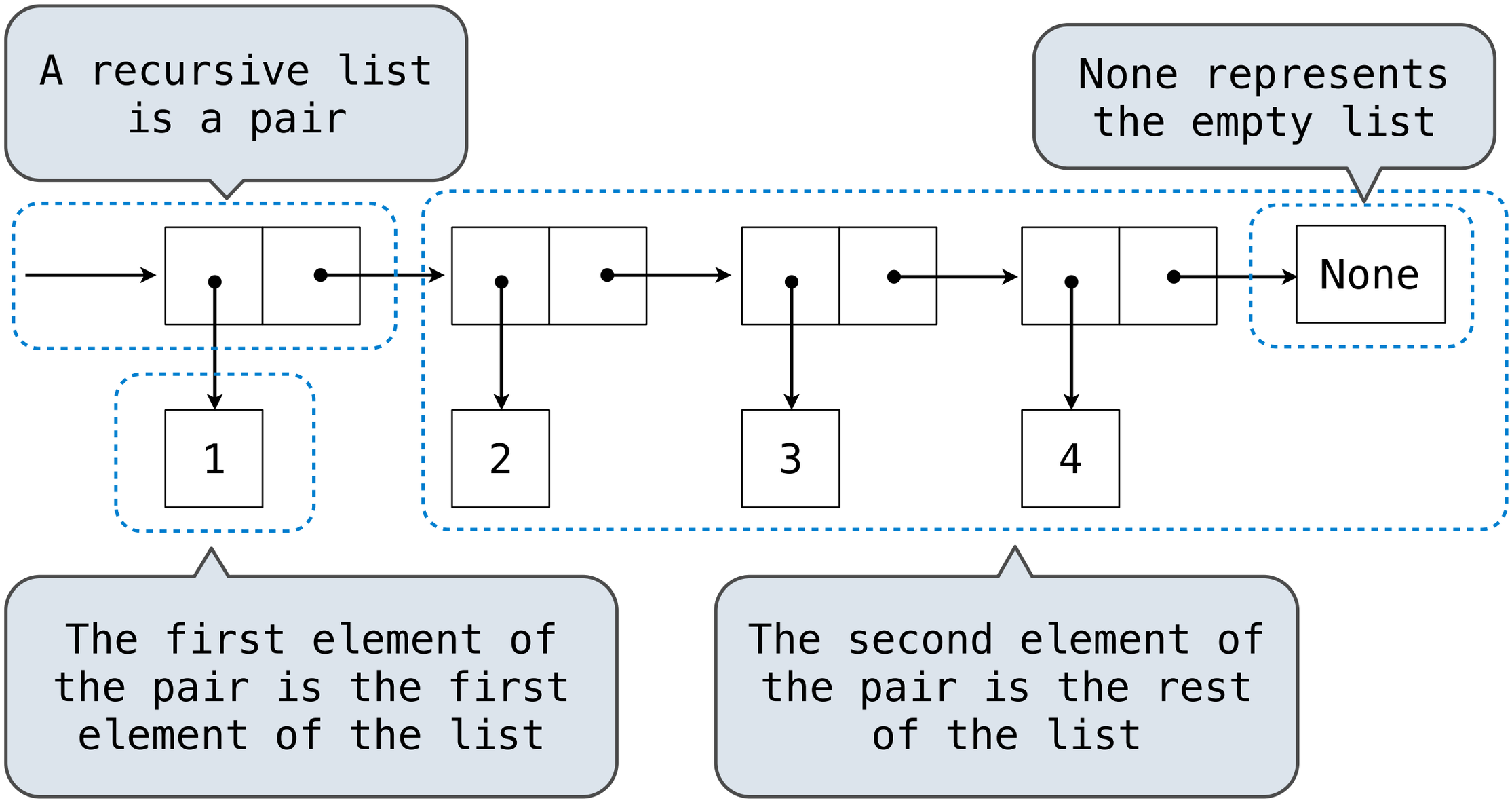
Implementing Recursive Lists with Pairs



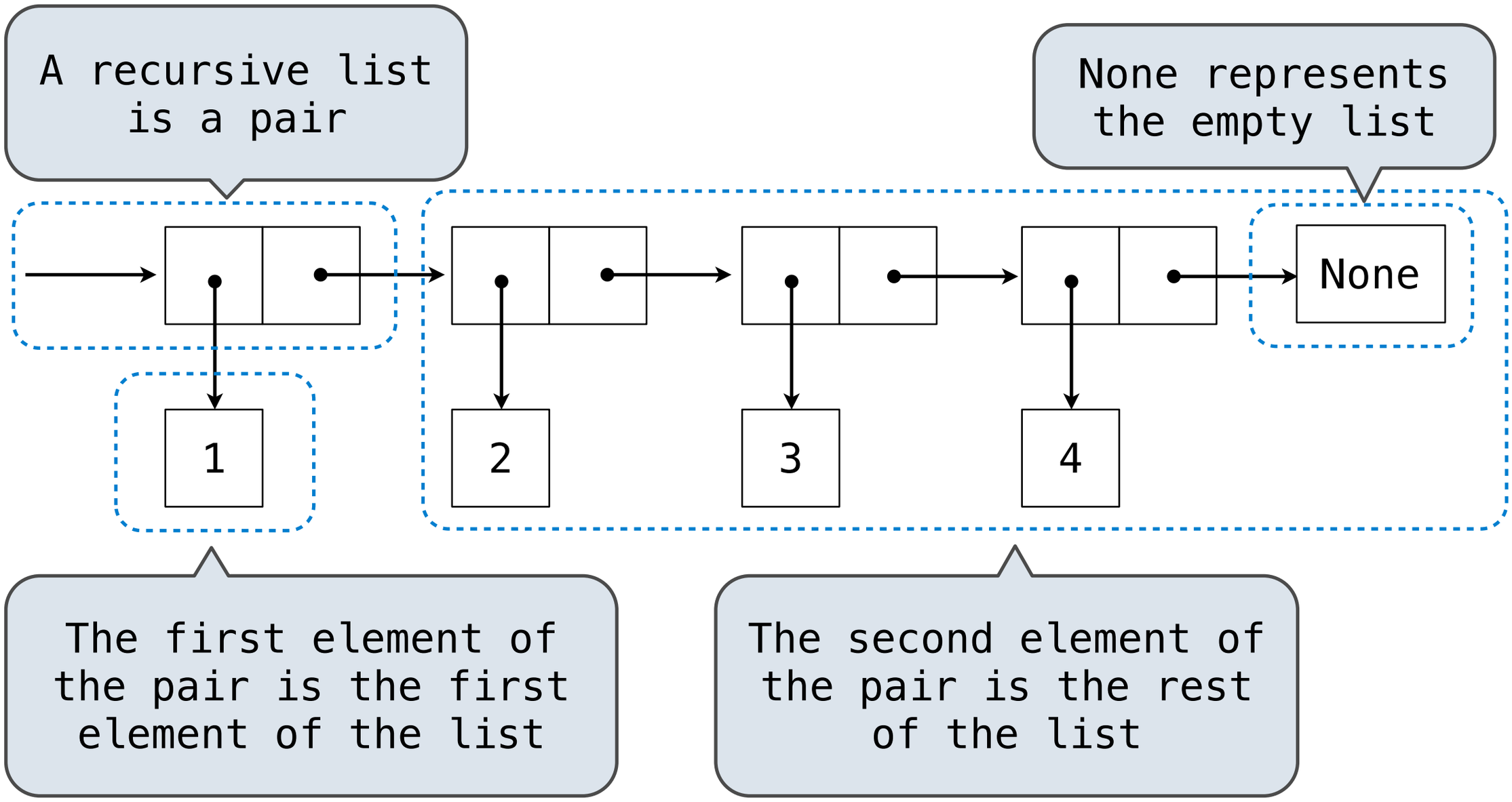
Implementing Recursive Lists with Pairs



Implementing Recursive Lists with Pairs



Implementing Recursive Lists with Pairs



(Demo)

Implementing the Sequence Abstraction

Implementing the Sequence Abstraction

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Implementing the Sequence Abstraction

```
def len_rlist(s):
    """Return the length of recursive list s."""
    length = 0
    while s != empty_rlist:
        s, length = rest(s), length + 1
    return length
```

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Implementing the Sequence Abstraction

```
def len_rlist(s):
    """Return the length of recursive list s."""
    length = 0
    while s != empty_rlist:
        s, length = rest(s), length + 1
    return length
```

```
def getitem_rlist(s, i):
    """Return the element at index i of recursive list s."""
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Implementing the Sequence Abstraction

```
def len_rlist(s):
    """Return the length of recursive list s."""
    length = 0
    while s != empty_rlist:
        s, length = rest(s), length + 1
    return length
```

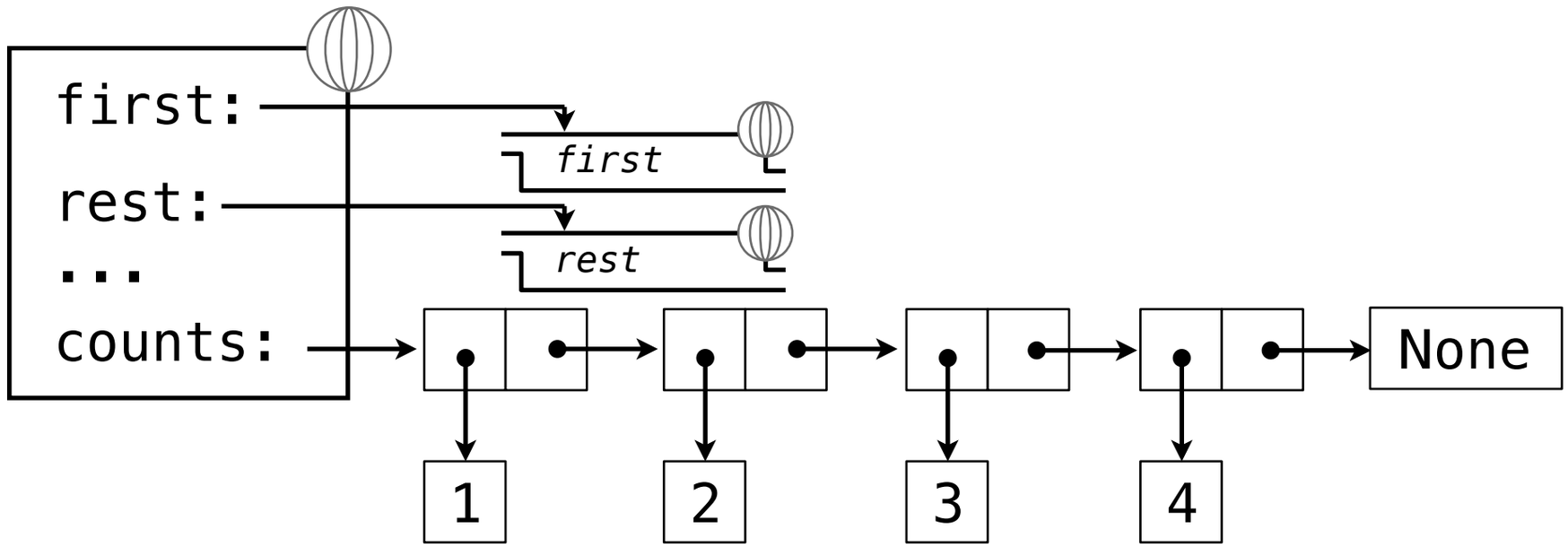
(Demo)

```
def getitem_rlist(s, i):
    """Return the element at index i of recursive list s."""
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

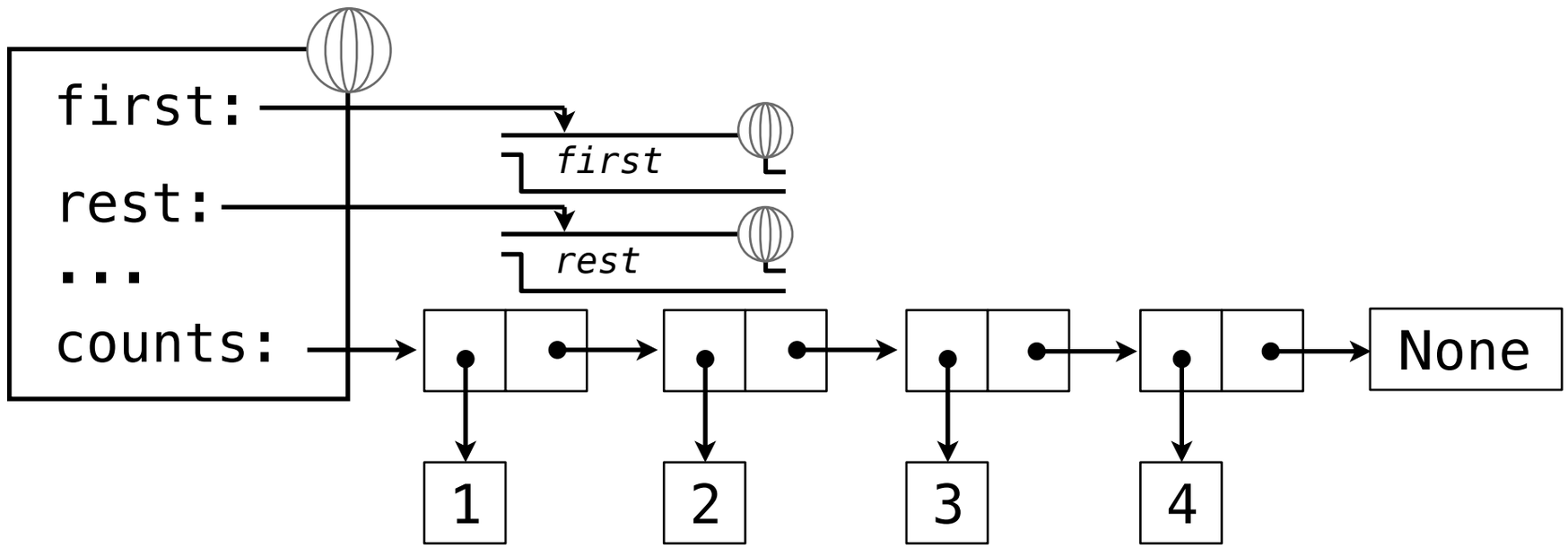
Length. A sequence has a finite length.

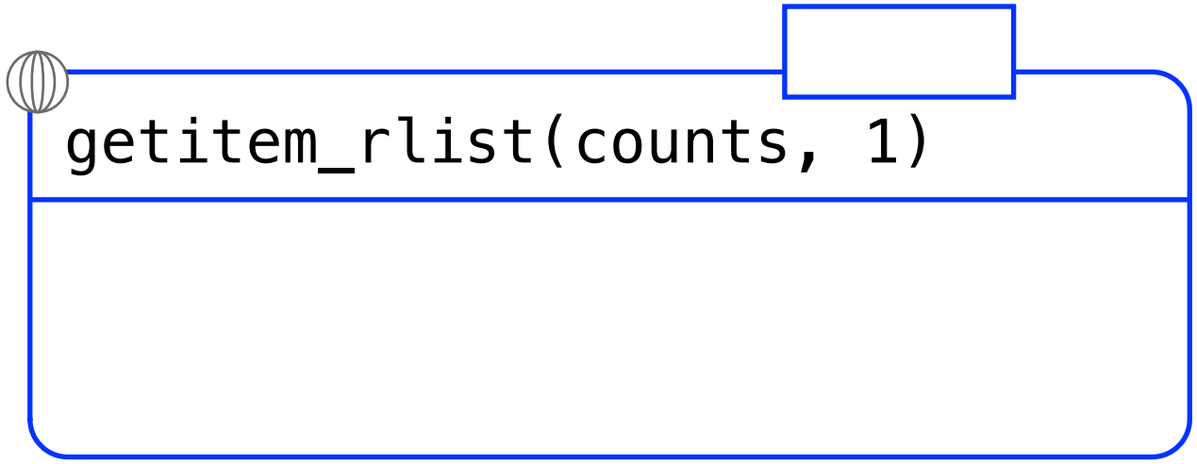
Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Environment Diagram for getitem_rlist

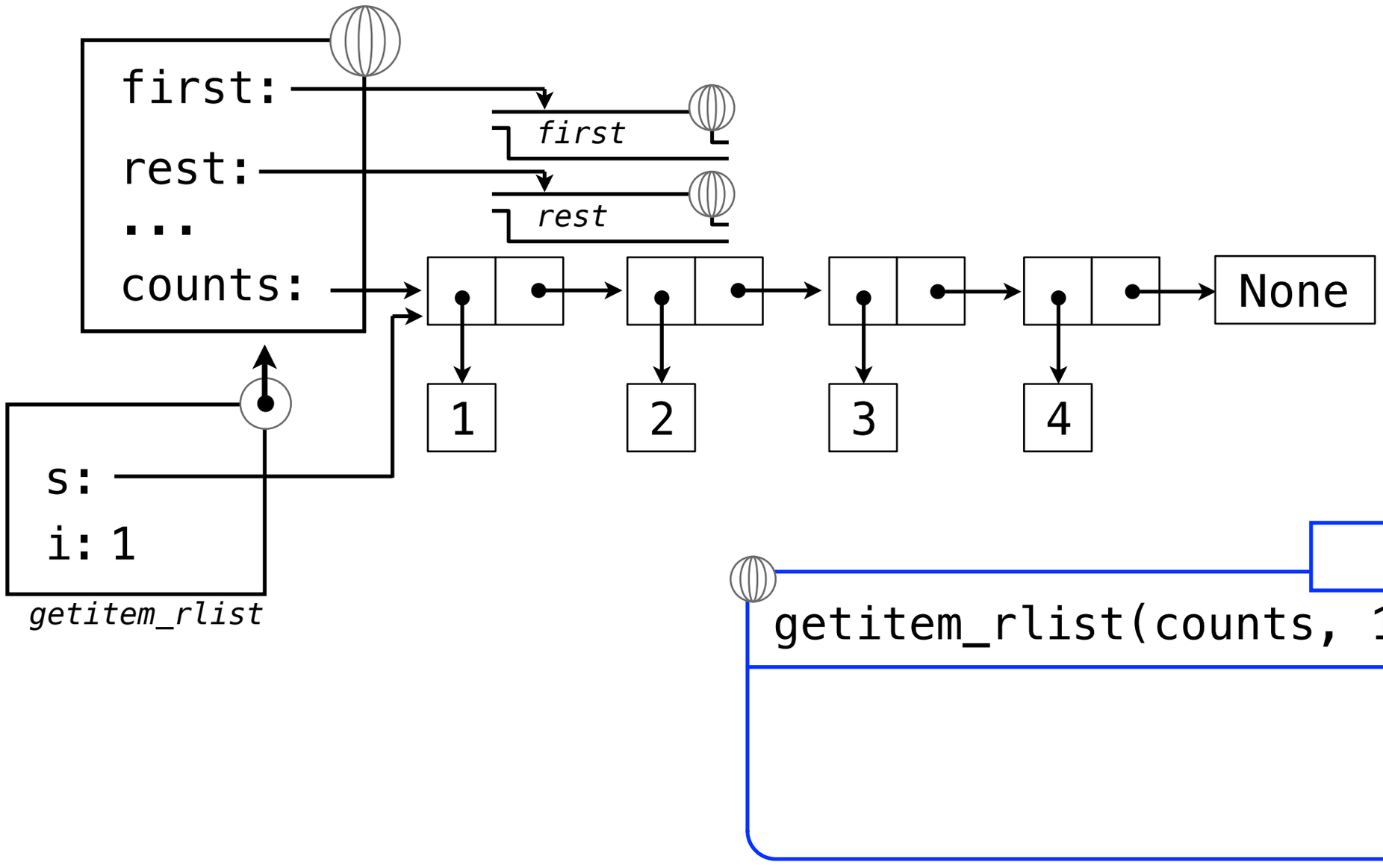


Environment Diagram for getitem_rlist

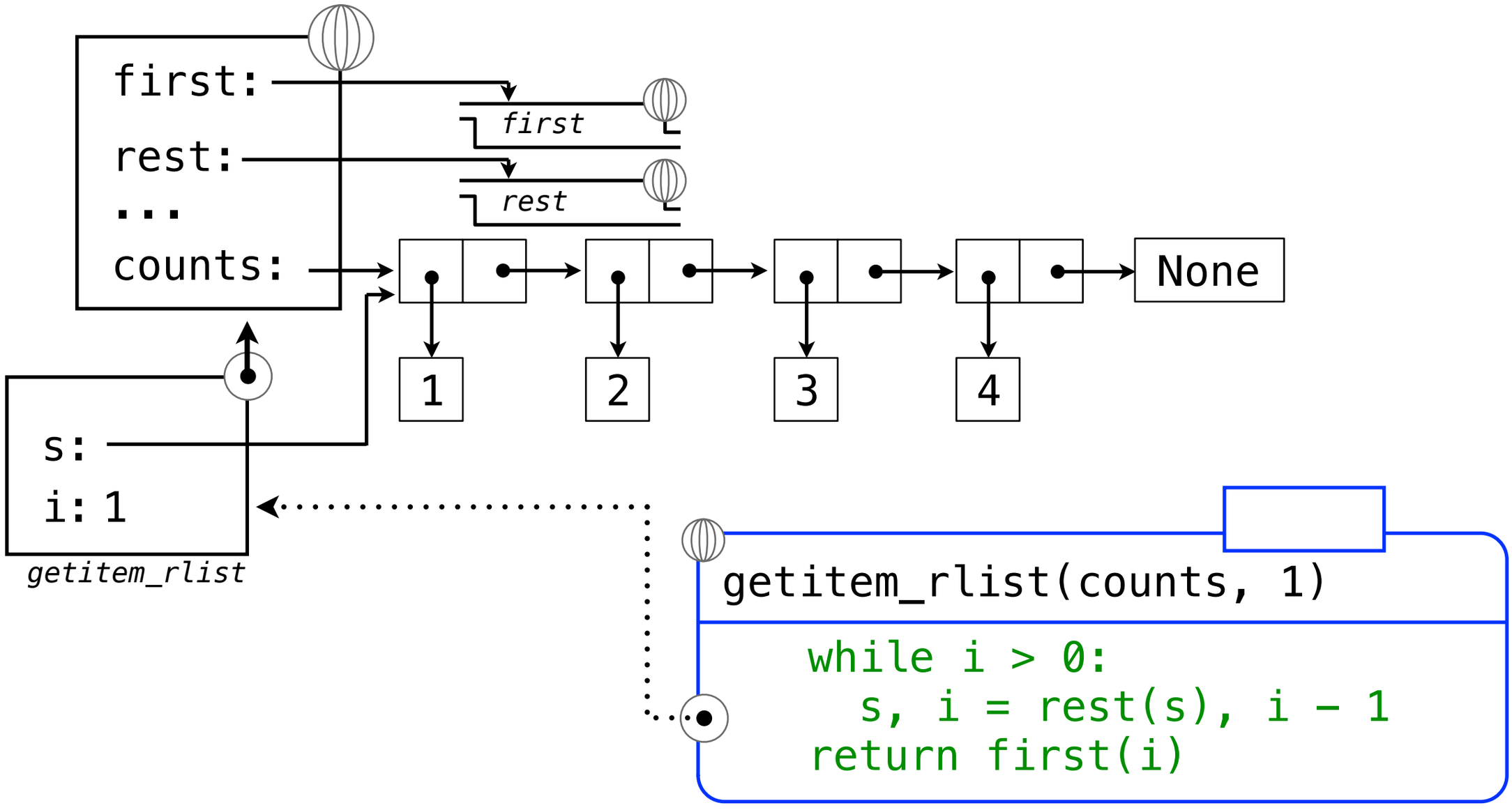




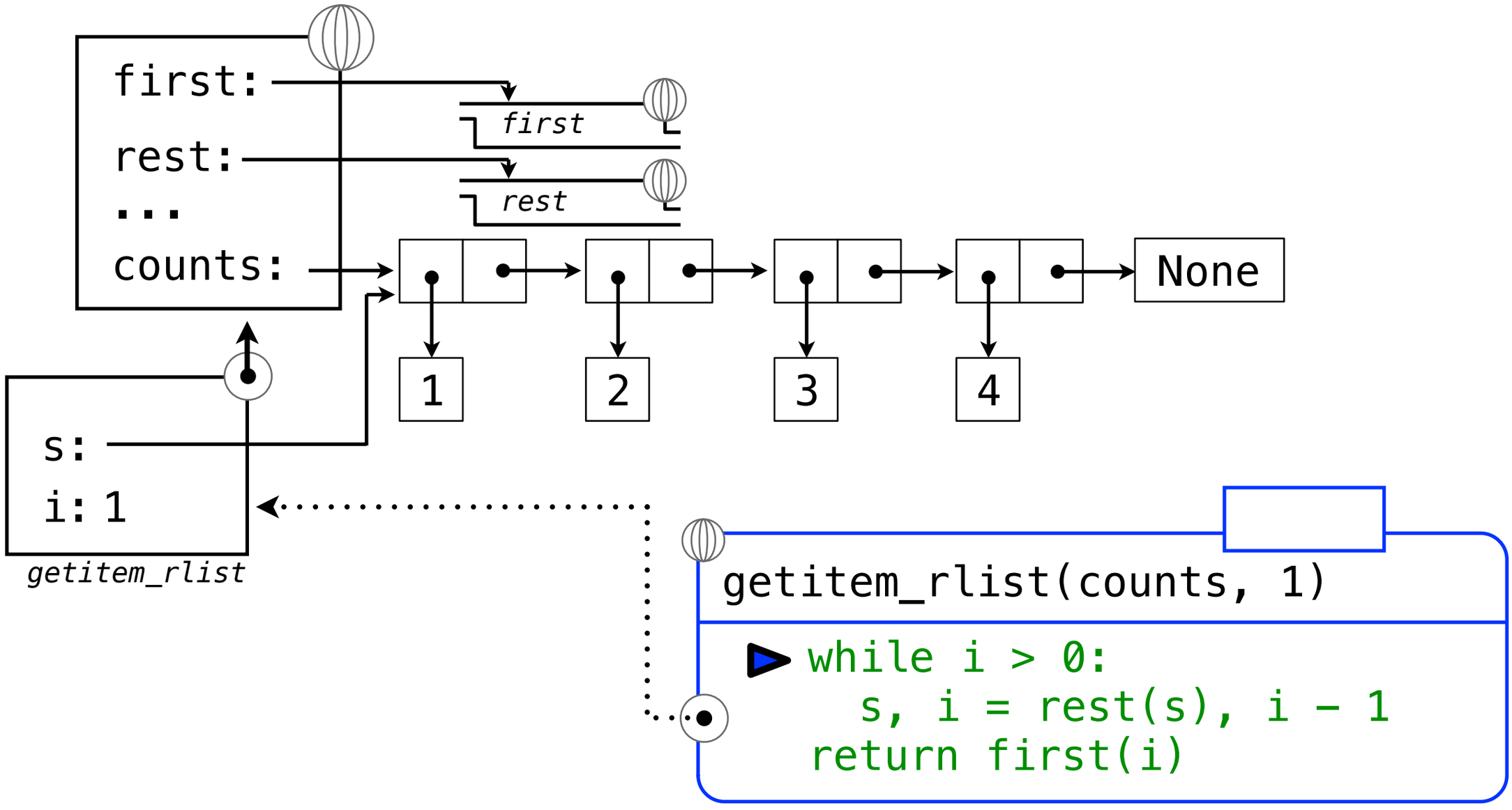
Environment Diagram for getitem_rlist



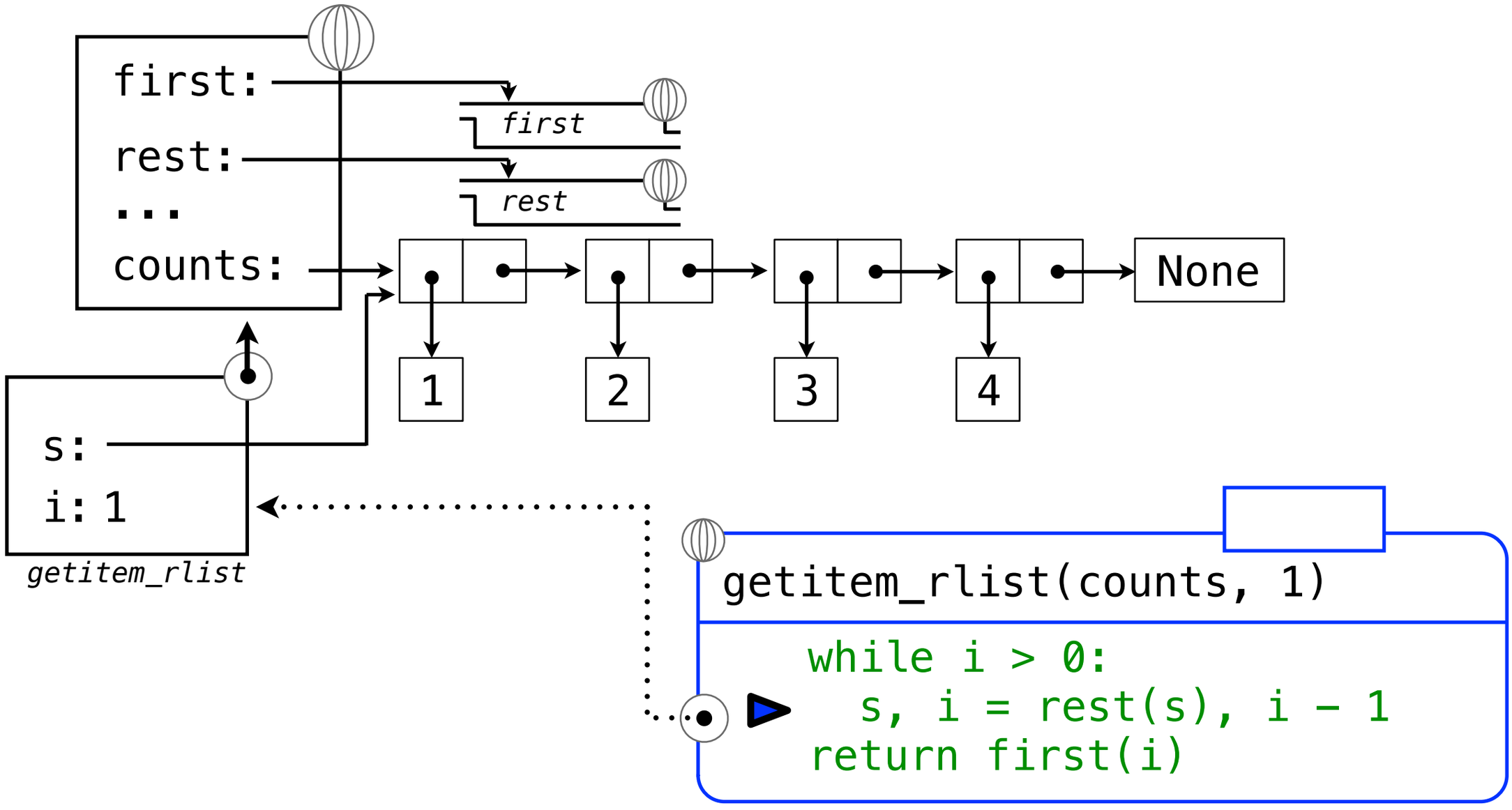
Environment Diagram for getitem_rlist



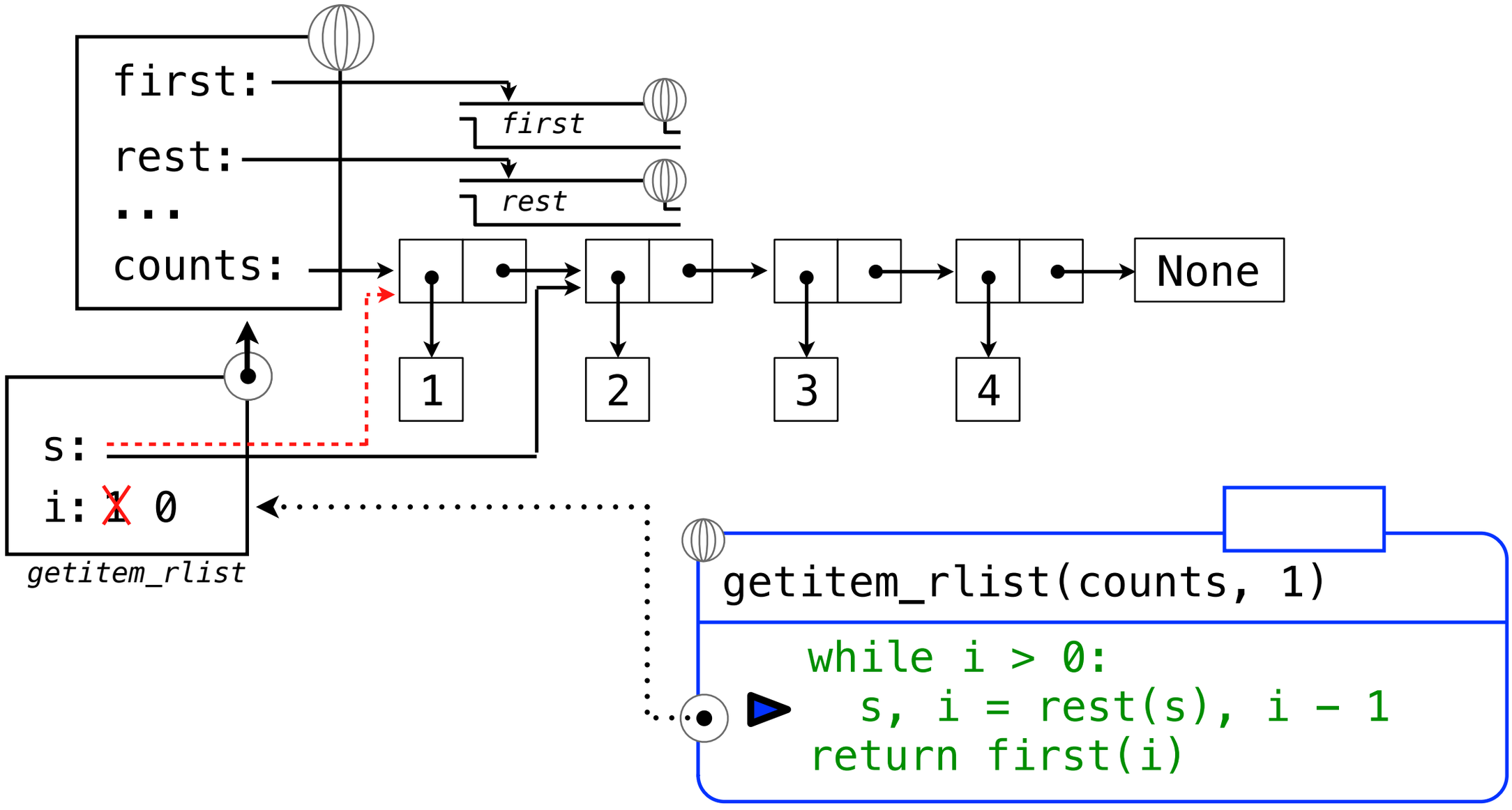
Environment Diagram for getitem_rlist



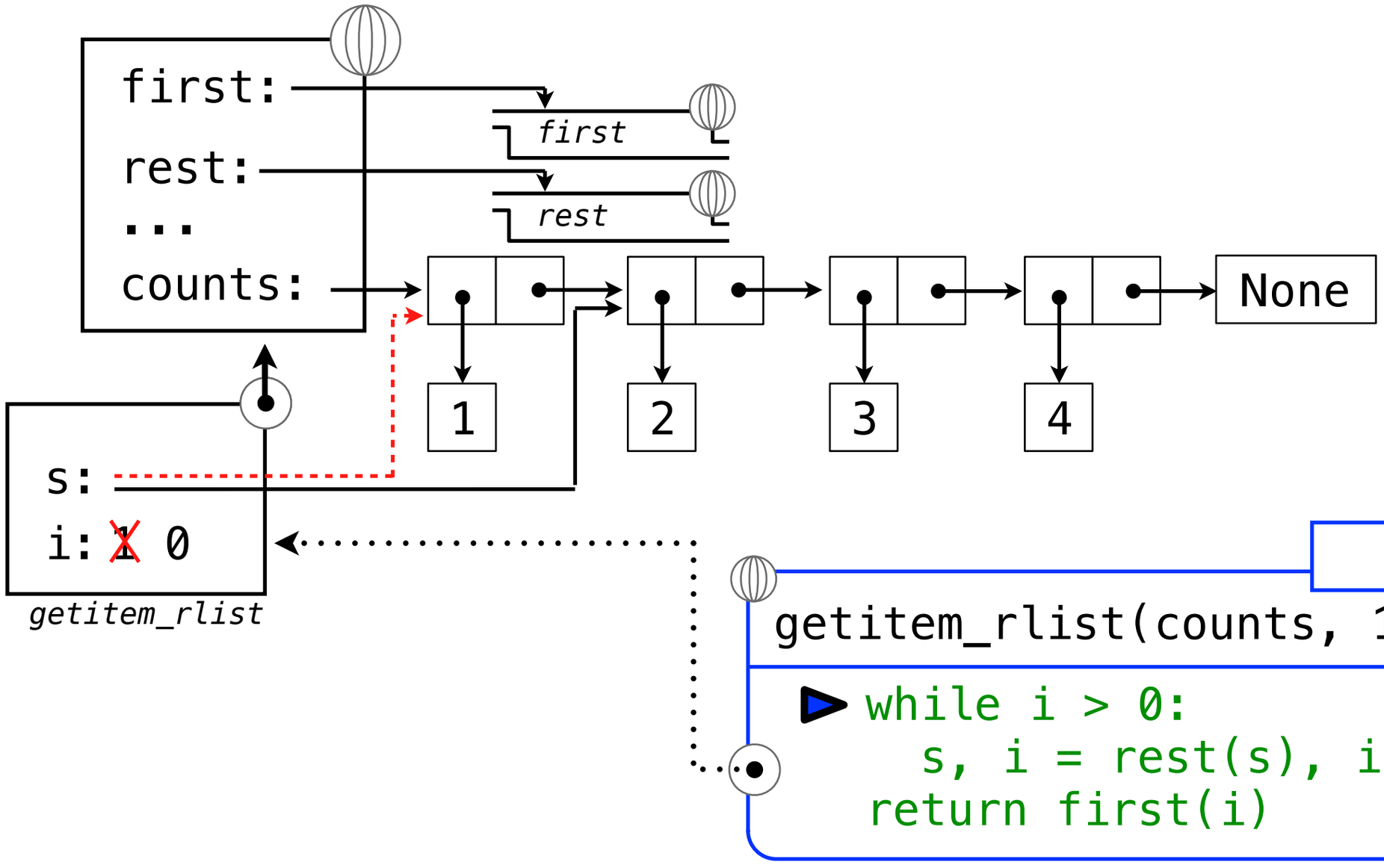
Environment Diagram for getitem_rlist



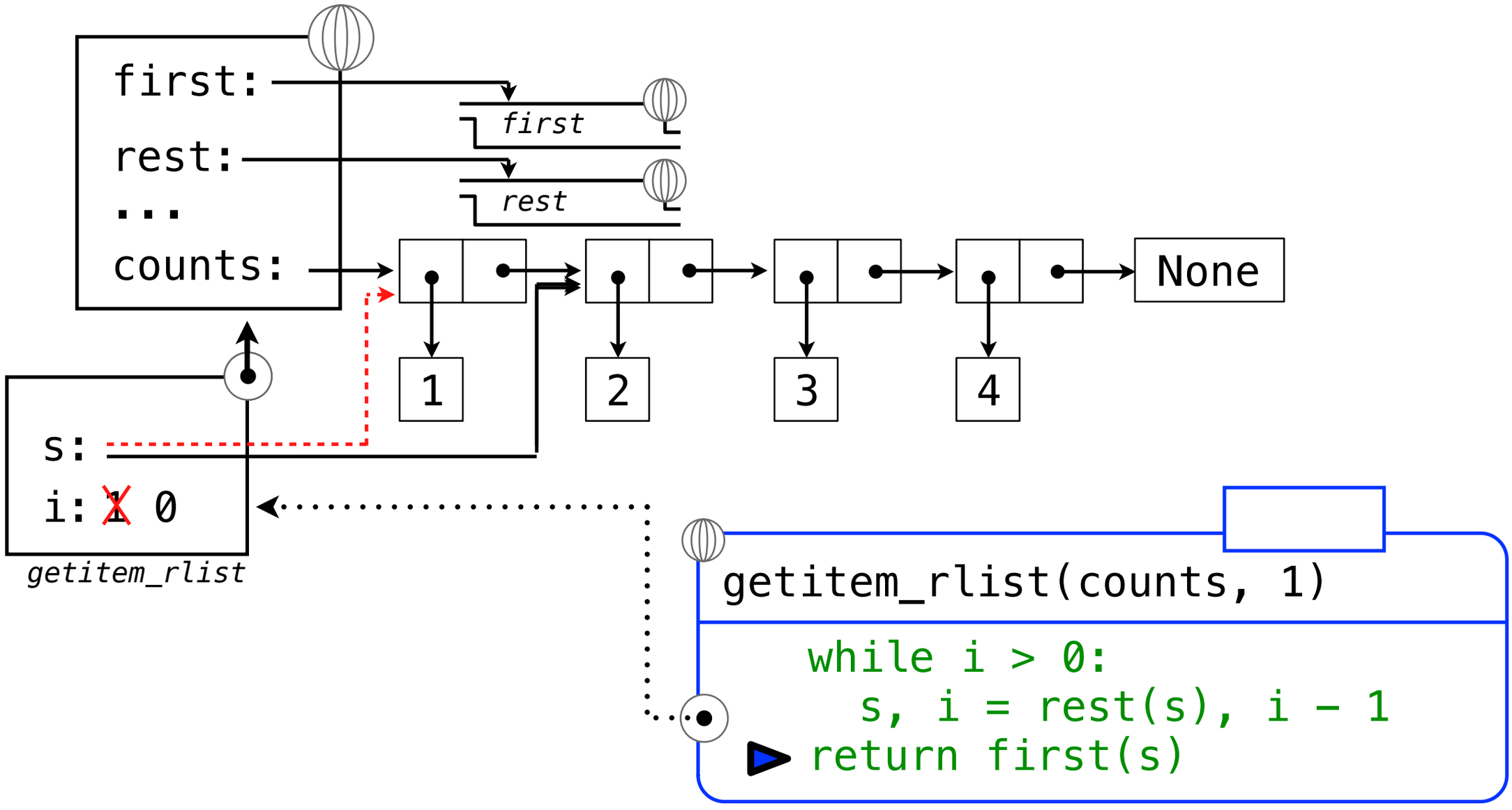
Environment Diagram for getitem_rlist



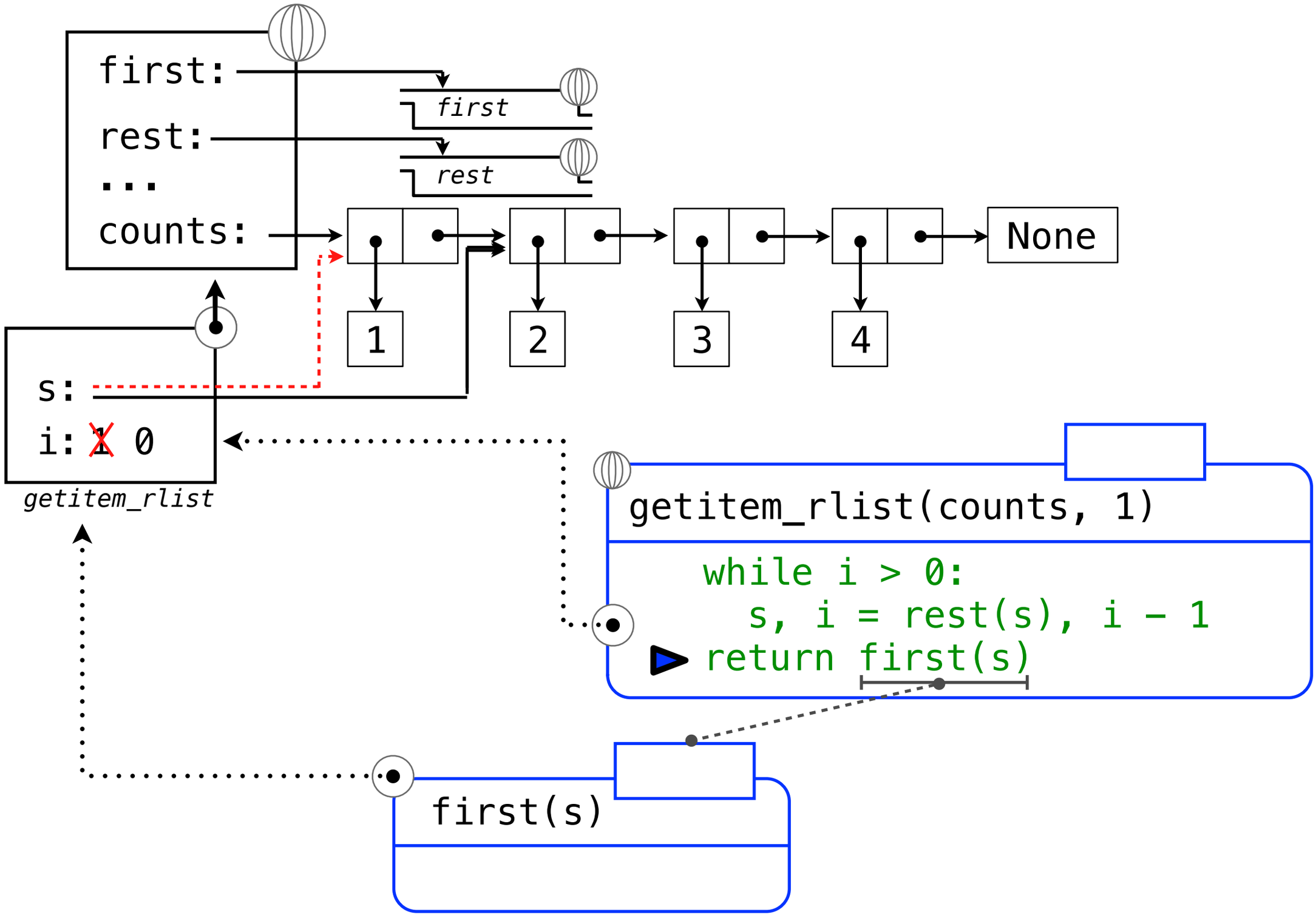
Environment Diagram for getitem_rlist



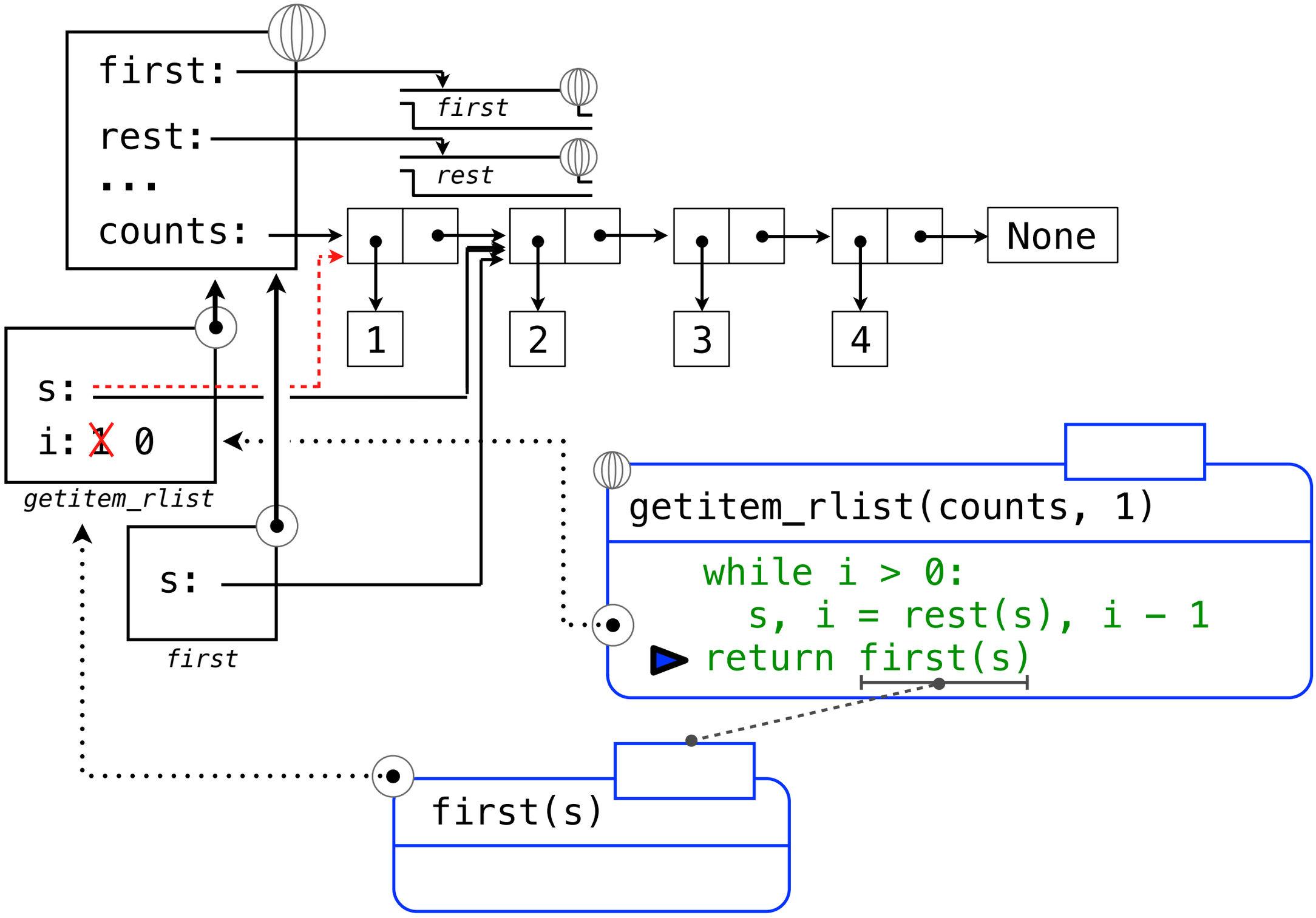
Environment Diagram for getitem_rlist



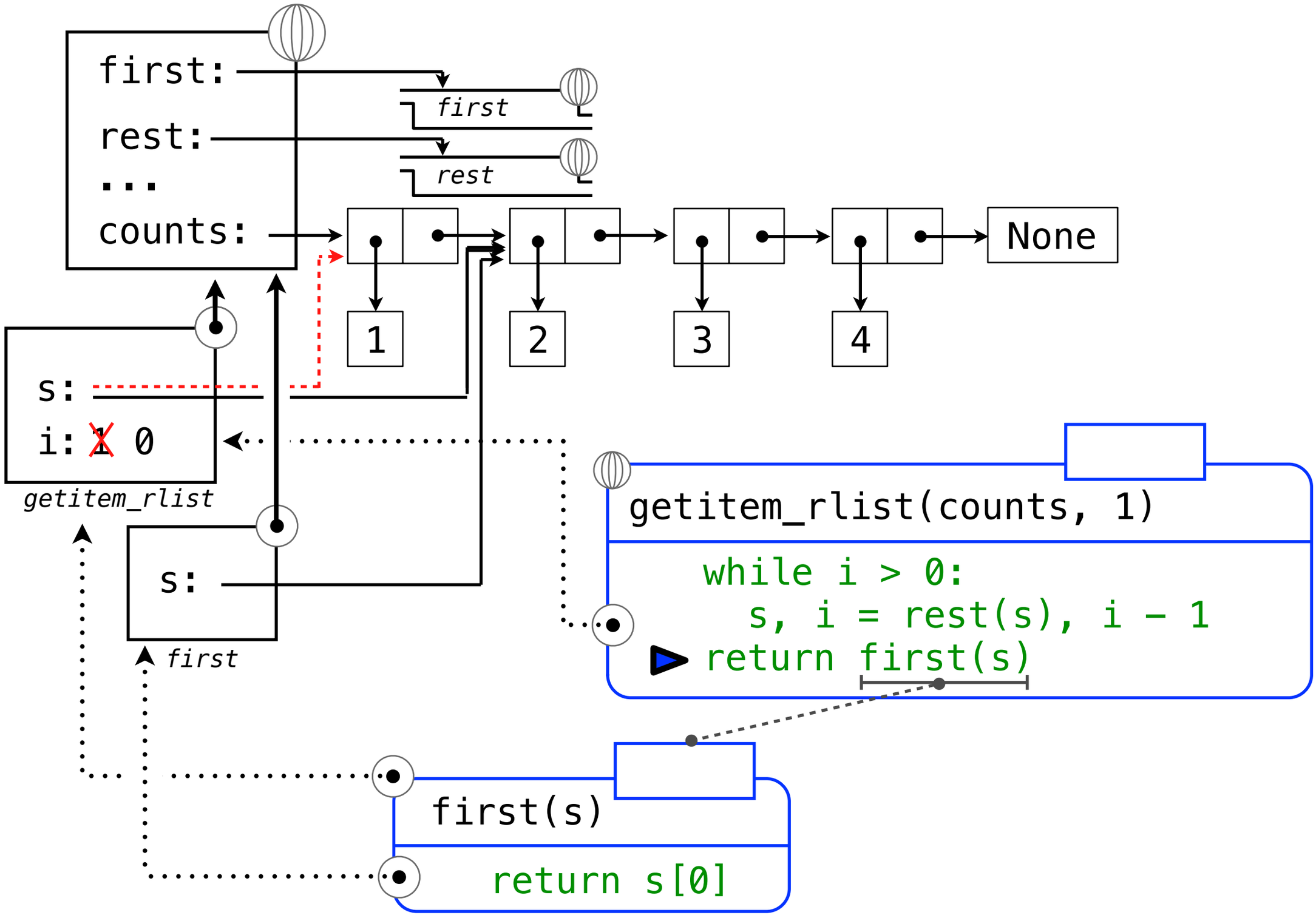
Environment Diagram for getitem_rlist



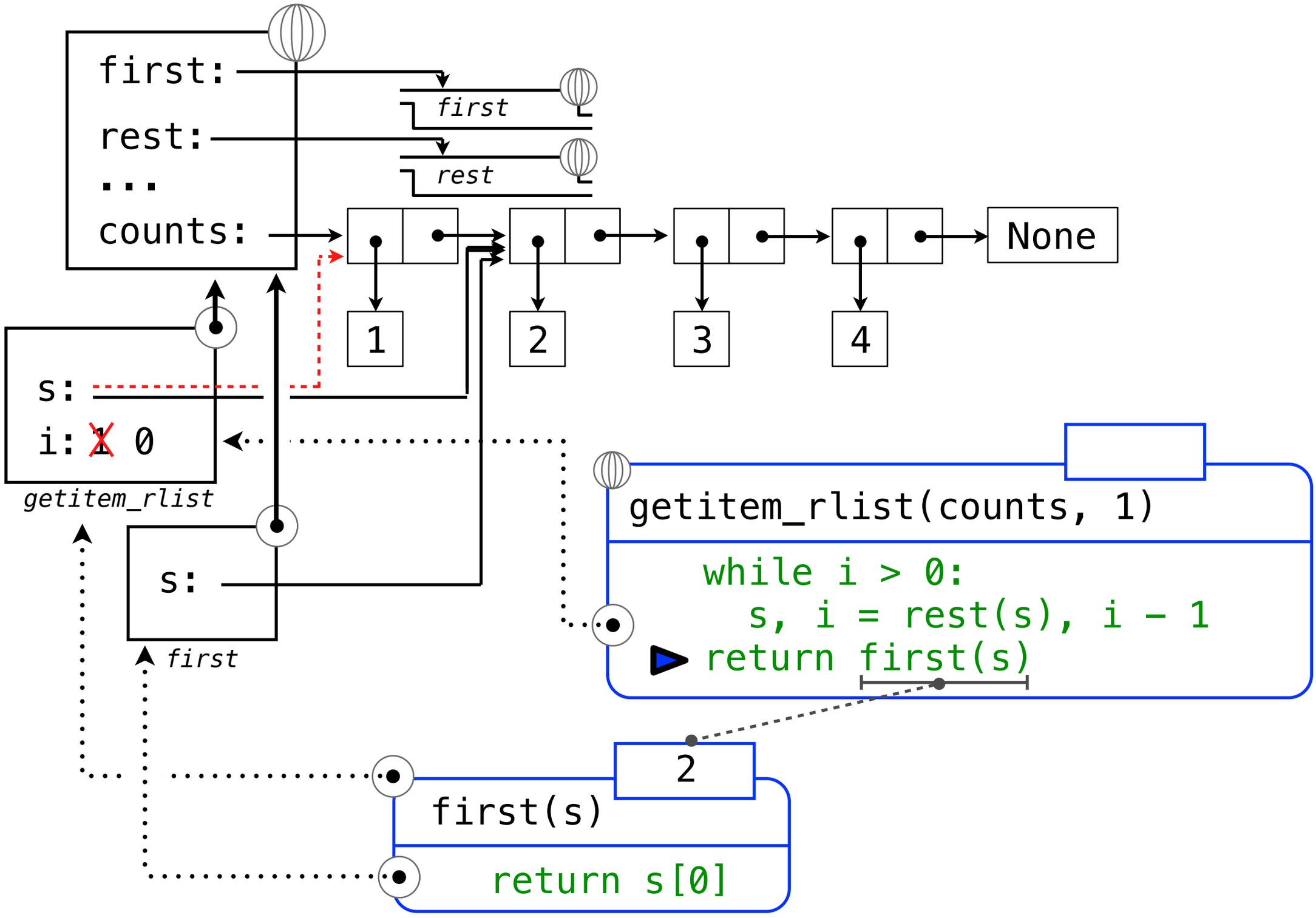
Environment Diagram for getitem_rlist



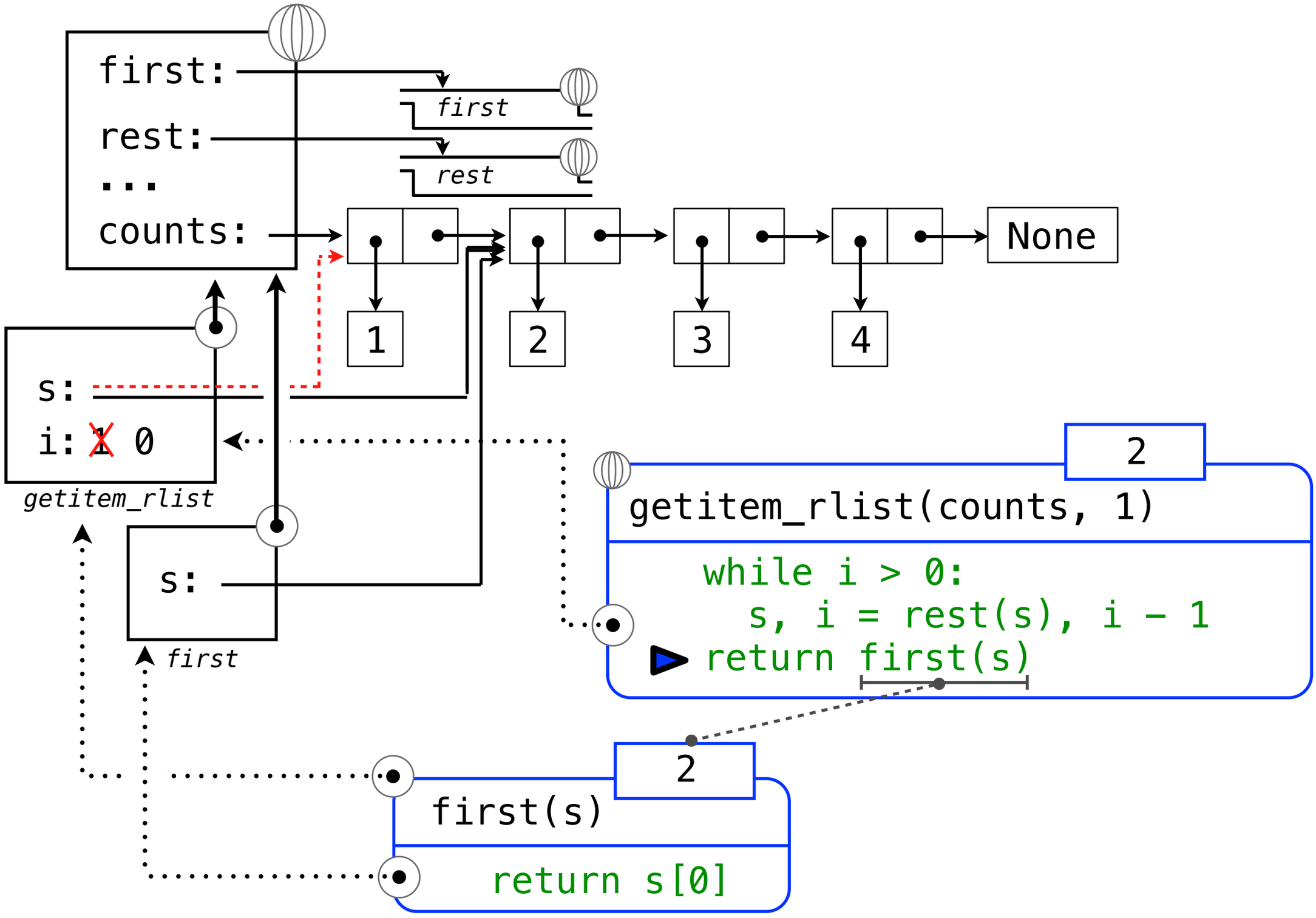
Environment Diagram for getitem_rlist



Environment Diagram for getitem_rlist



Environment Diagram for getitem_rlist



Sequence Iteration

(Demo)

Sequence Iteration

(Demo)

```
def count(s, value):
    total = 0
    for elem in s:
        if elem == value:
            total = total + 1
    return total
```

Name bound in the first frame of the current environment

For Statement Execution Procedure

For Statement Execution Procedure

```
for <name> in <expression>:
    <suite>
```

For Statement Execution Procedure

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header <expression>, which must yield an iterable value.

For Statement Execution Procedure

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header <expression>, which must yield an iterable value.

2. For each element value in that sequence, in order:

For Statement Execution Procedure

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header <expression>, which must yield an iterable value.

2. For each element value in that sequence, in order:

   A. Bind <name> to that value in the local environment.

For Statement Execution Procedure

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header <expression>, which must yield an iterable value.

2. For each element value in that sequence, in order:

   A. Bind <name> to that value in the local environment.

   B. Execute the <suite>.

Sequence Unpacking in For Statements

Sequence Unpacking in For Statements

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (4, 4))
>>> same_count = 0
```

Sequence Unpacking in For Statements

A sequence of
fixed-length sequences

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (4, 4))
>>> same_count = 0
```

Sequence Unpacking in For Statements

A sequence of fixed-length sequences

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (4, 4))
>>> same_count = 0
```

```
>>> for x, y in pairs:
        if x == y:
            same_count = same_count + 1

>>> same_count
2
```

Sequence Unpacking in For Statements

A sequence of fixed-length sequences

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (4, 4))
>>> same_count = 0
```

A name for each element in a fixed-length sequence

```
>>> for x, y in pairs:
        if x == y:
            same_count = same_count + 1

>>> same_count
2
```

Sequence Unpacking in For Statements

A sequence of fixed-length sequences

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (4, 4))
>>> same_count = 0
```

A name for each element in a fixed-length sequence

Each name is bound to a value, as in multiple assignment

```
>>> for x, y in pairs:
        if x == y:
            same_count = same_count + 1

>>> same_count
2
```

The Range Type

A sequence of consecutive integers is a range.*

The Range Type

A sequence of consecutive integers is a range.*

* Ranges can actually represent more general sequences, too.

The Range Type

A sequence of consecutive integers is a range.*

```
..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...
```

* Ranges can actually represent more general sequences, too.

The Range Type

A sequence of consecutive integers is a range.*

..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

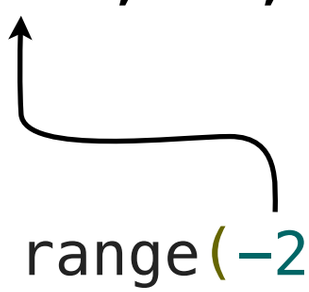
```
range(-2, 2)
```

* Ranges can actually represent more general sequences, too.

The Range Type

A sequence of consecutive integers is a range.*

```
..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...
```

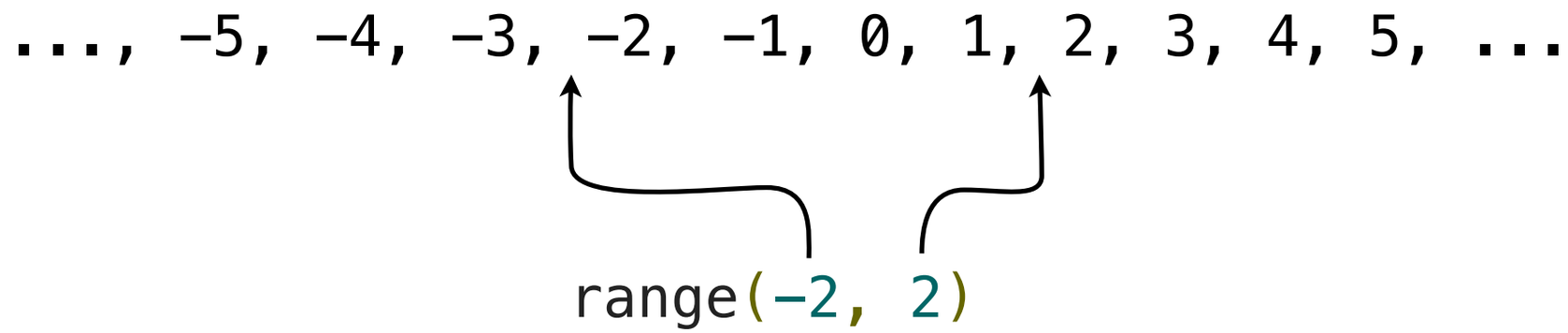


```
range(-2, 2)
```

* Ranges can actually represent more general sequences, too.

The Range Type

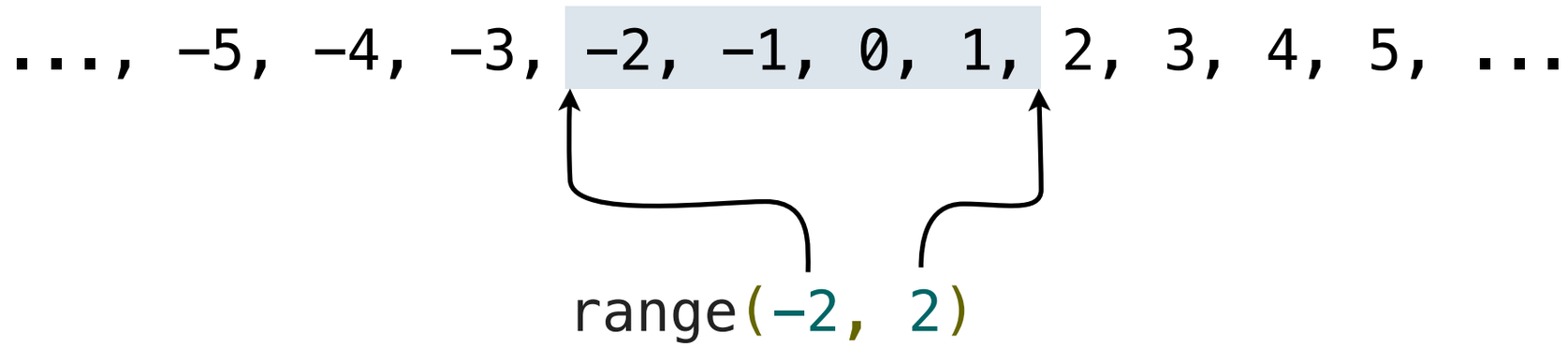
A sequence of consecutive integers is a range.*



* Ranges can actually represent more general sequences, too.

The Range Type

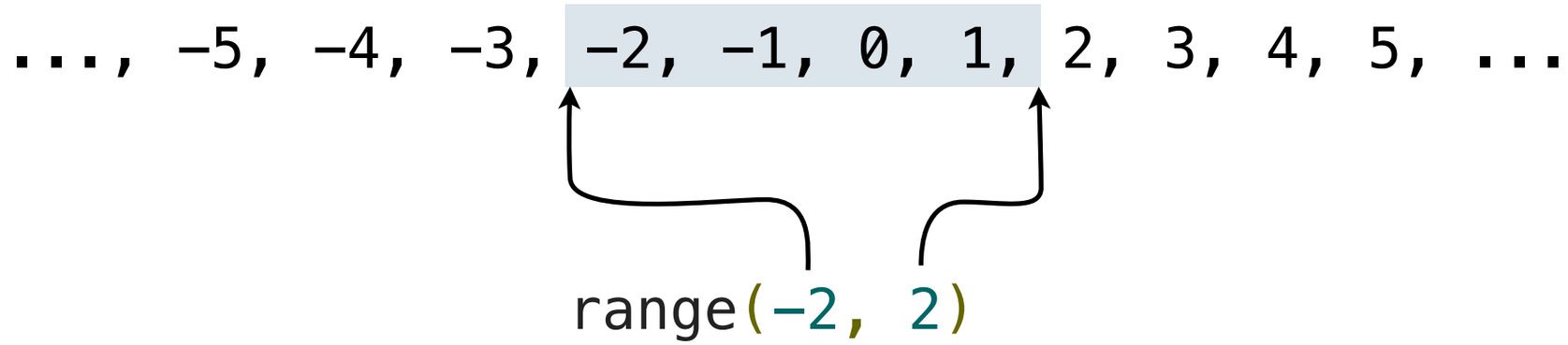
A sequence of consecutive integers is a range.*



* Ranges can actually represent more general sequences, too.

The Range Type

A sequence of consecutive integers is a range.*

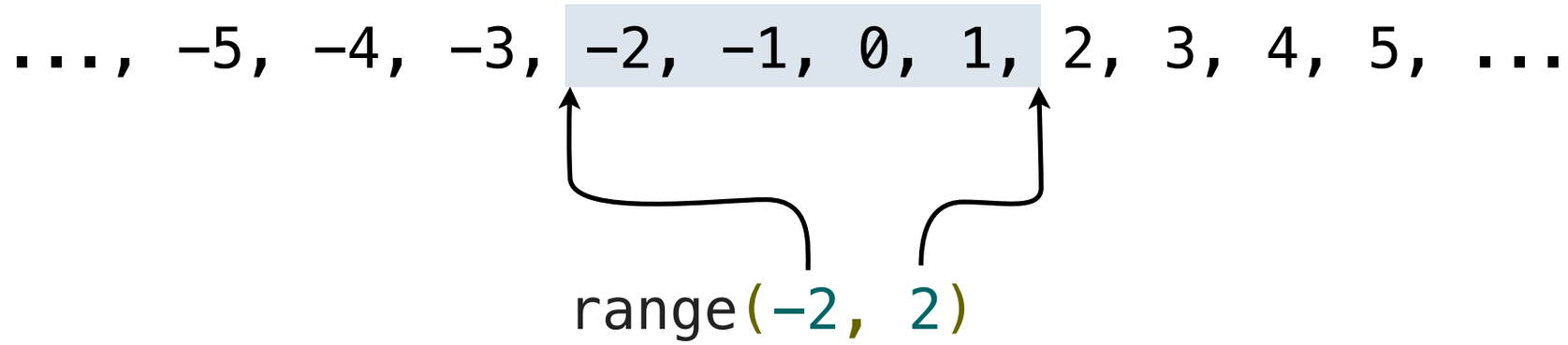


Length: ending value − starting value

* Ranges can actually represent more general sequences, too.

The Range Type

A sequence of consecutive integers is a range.*



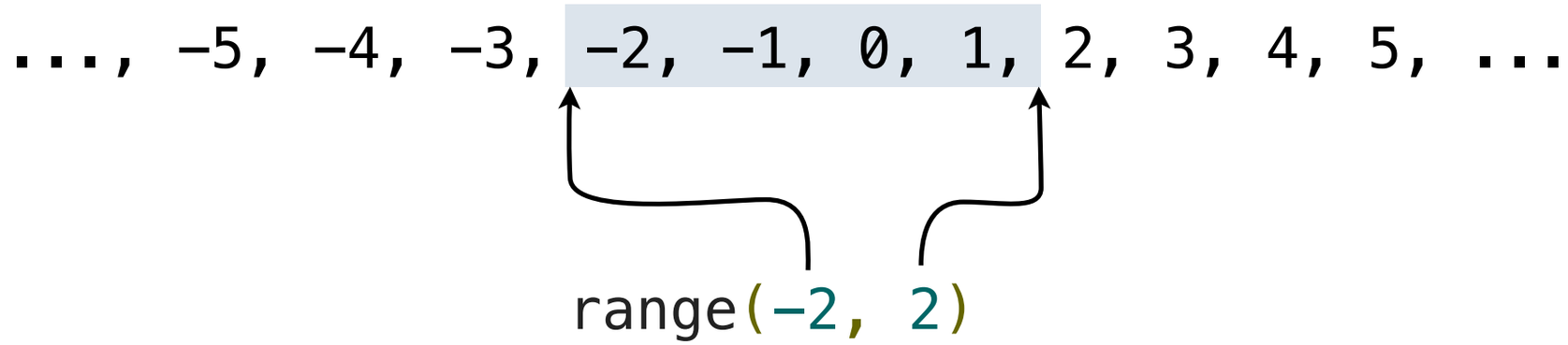
Length: ending value − starting value

Element selection: starting value + index

* Ranges can actually represent more general sequences, too.

The Range Type

A sequence of consecutive integers is a range.*



Length: ending value − starting value

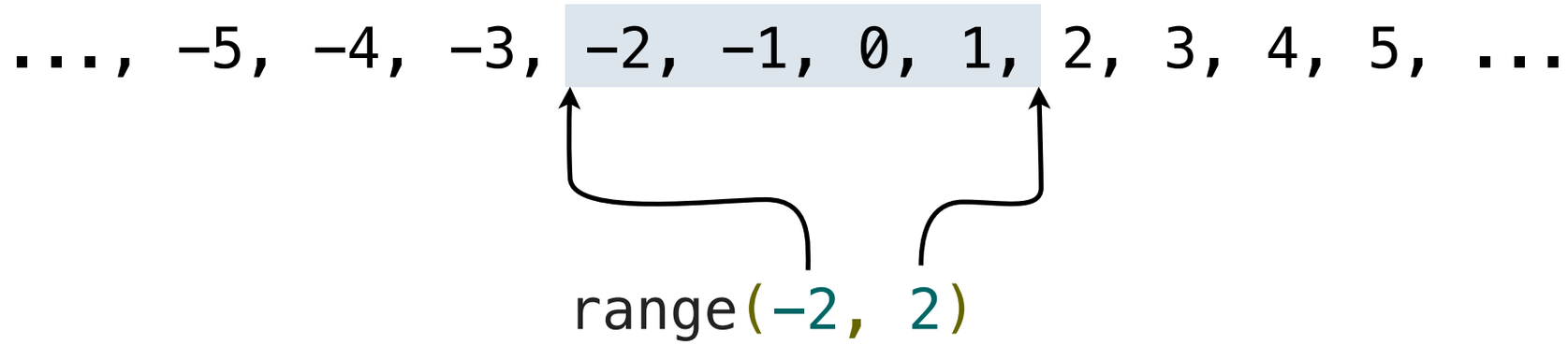
Element selection: starting value + index

```
>>> tuple(range(-2, 2))
(-2, -1, 0, 1)

>>> tuple(range(4))
(0, 1, 2, 3)
```

* Ranges can actually represent more general sequences, too.

The Range Type

A sequence of consecutive integers is a range.*



Length: ending value − starting value

Element selection: starting value + index

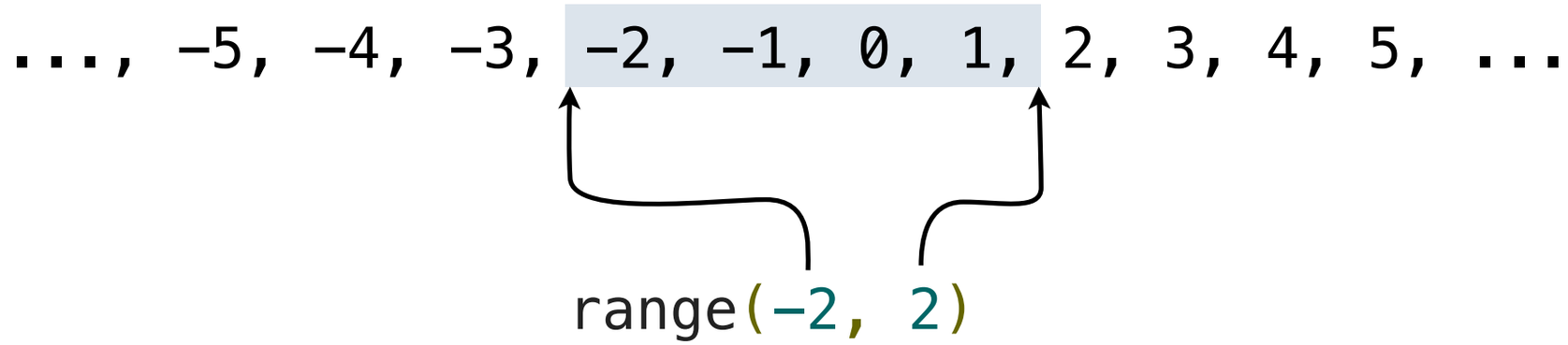
```
>>> tuple(range(-2, 2))
(-2, -1, 0, 1)

>>> tuple(range(4))
(0, 1, 2, 3)
```

Tuple construction

* Ranges can actually represent more general sequences, too.

The Range Type

A sequence of consecutive integers is a range.*



Length: ending value − starting value

Element selection: starting value + index

```
>>> tuple(range(-2, 2))
(-2, -1, 0, 1)

>>> tuple(range(4))
(0, 1, 2, 3)
```

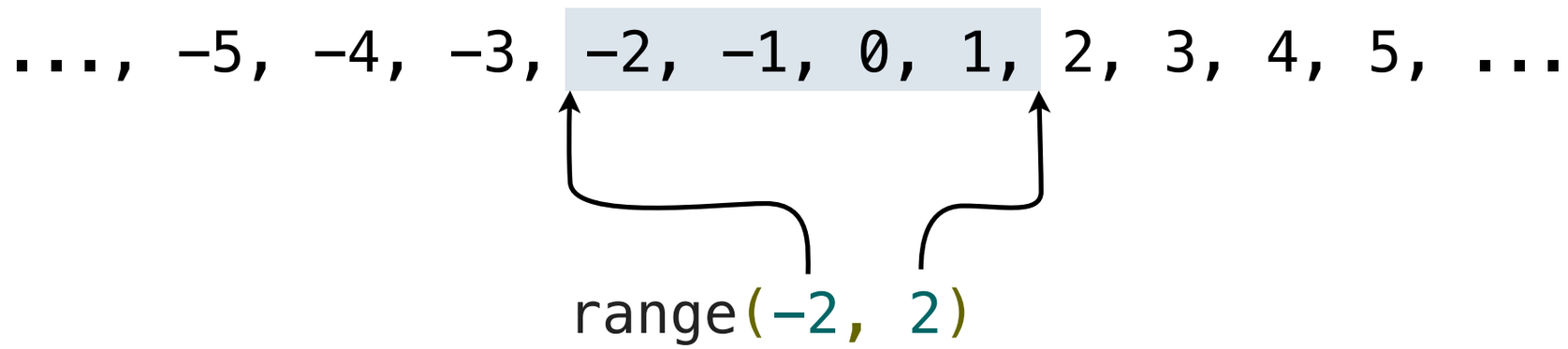
Tuple construction

Default starting value

* Ranges can actually represent more general sequences, too.

The Range Type

A sequence of consecutive integers is a range.*



Length: ending value − starting value

(Demo)

Element selection: starting value + index

```
>>> tuple(range(-2, 2))
(-2, -1, 0, 1)

>>> tuple(range(4))
(0, 1, 2, 3)
```

Tuple construction

Default starting value

* Ranges can actually represent more general sequences, too.

Membership & Slicing

The Python sequence abstraction has two more behaviors!

Membership & Slicing

The Python sequence abstraction has two more behaviors!

Membership.

Membership & Slicing

The Python sequence abstraction has two more behaviors!

Membership.

```
>>> digits = (1, 8, 2, 8)
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Membership & Slicing

The Python sequence abstraction has two more behaviors!

Membership.

```
>>> digits = (1, 8, 2, 8)
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.

Membership & Slicing

The Python sequence abstraction has two more behaviors!

Membership.

```
>>> digits = (1, 8, 2, 8)
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.

```
>>> digits[0:2]
(1, 8)
>>> digits[1:]
(8, 2, 8)
```