61A Lecture 18

Friday, March 6

Announcements

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm (get started now!)

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
    - Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
  - Project party on Tuesday 3/10 5pm-6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
  - Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11
- Homework 6 due Monday 3/16 @ 11:59pm (not yet released)

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
  - Project party on Tuesday 3/10 5pm-6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11
- Homework 6 due Monday 3/16 @ 11:59pm (not yet released)
- Midterm 2 is on Thursday 3/19 7pm-9pm

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
  - Project party on Tuesday 3/10 5pm-6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11
- Homework 6 due Monday 3/16 @ 11:59pm (not yet released)
- Midterm 2 is on Thursday 3/19 7pm-9pm
  - Emphasis: mutable data, object-oriented programming, recursion, and recursive data

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
  - Project party on Tuesday 3/10 5pm-6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11
- Homework 6 due Monday 3/16 @ 11:59pm (not yet released)
- Midterm 2 is on Thursday 3/19 7pm-9pm
  - Emphasis: mutable data, object-oriented programming, recursion, and recursive data
  - Fill out conflict form if you cannot attend due to a course conflict

Hog Contest Results

Hog Contest Results

Excellent participation!

51 qualified submissions

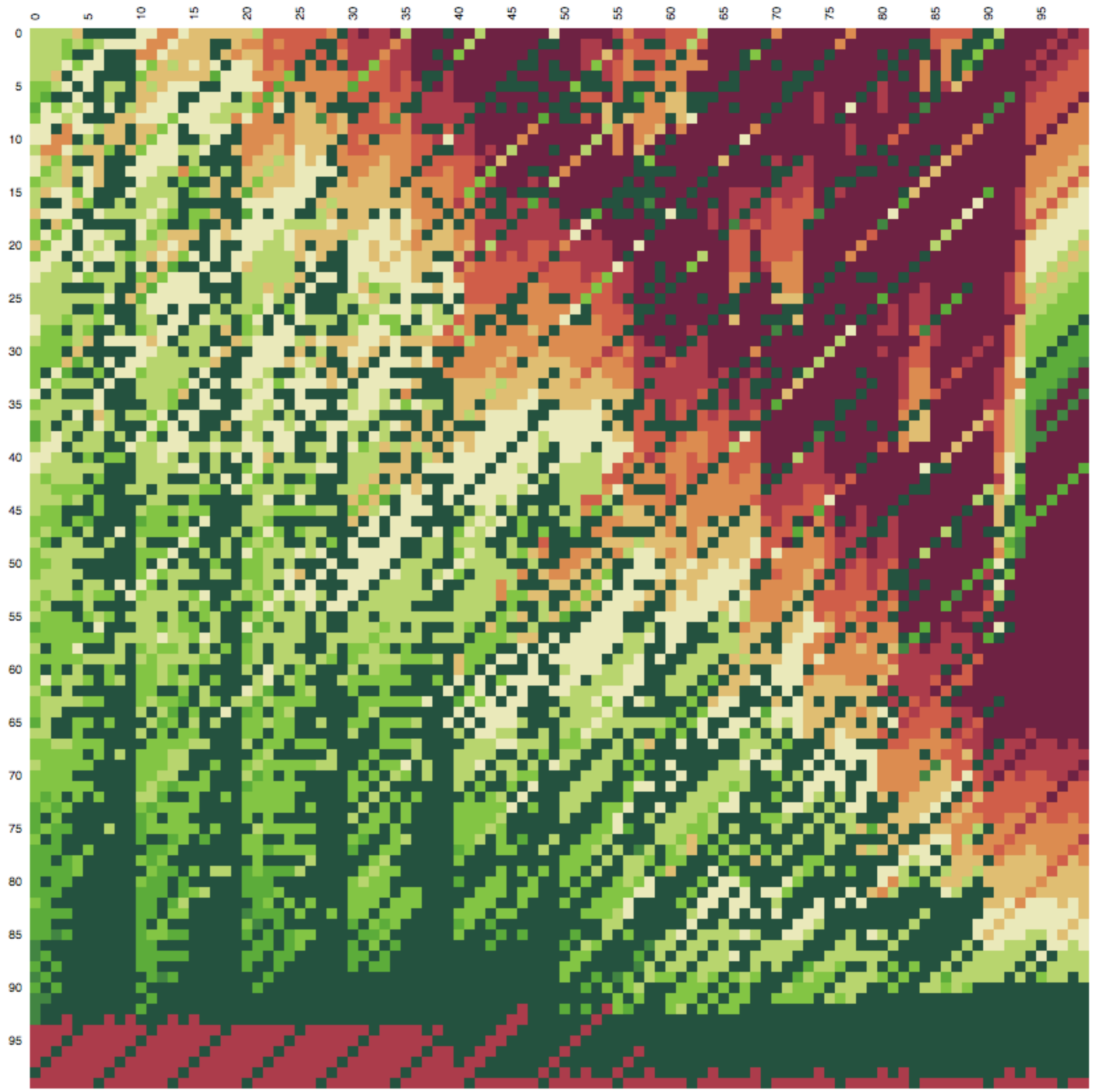
Lots of excellent ideas

Hog Contest Results

```
Excellent participation!

51 qualified submissions

Lots of excellent ideas
```



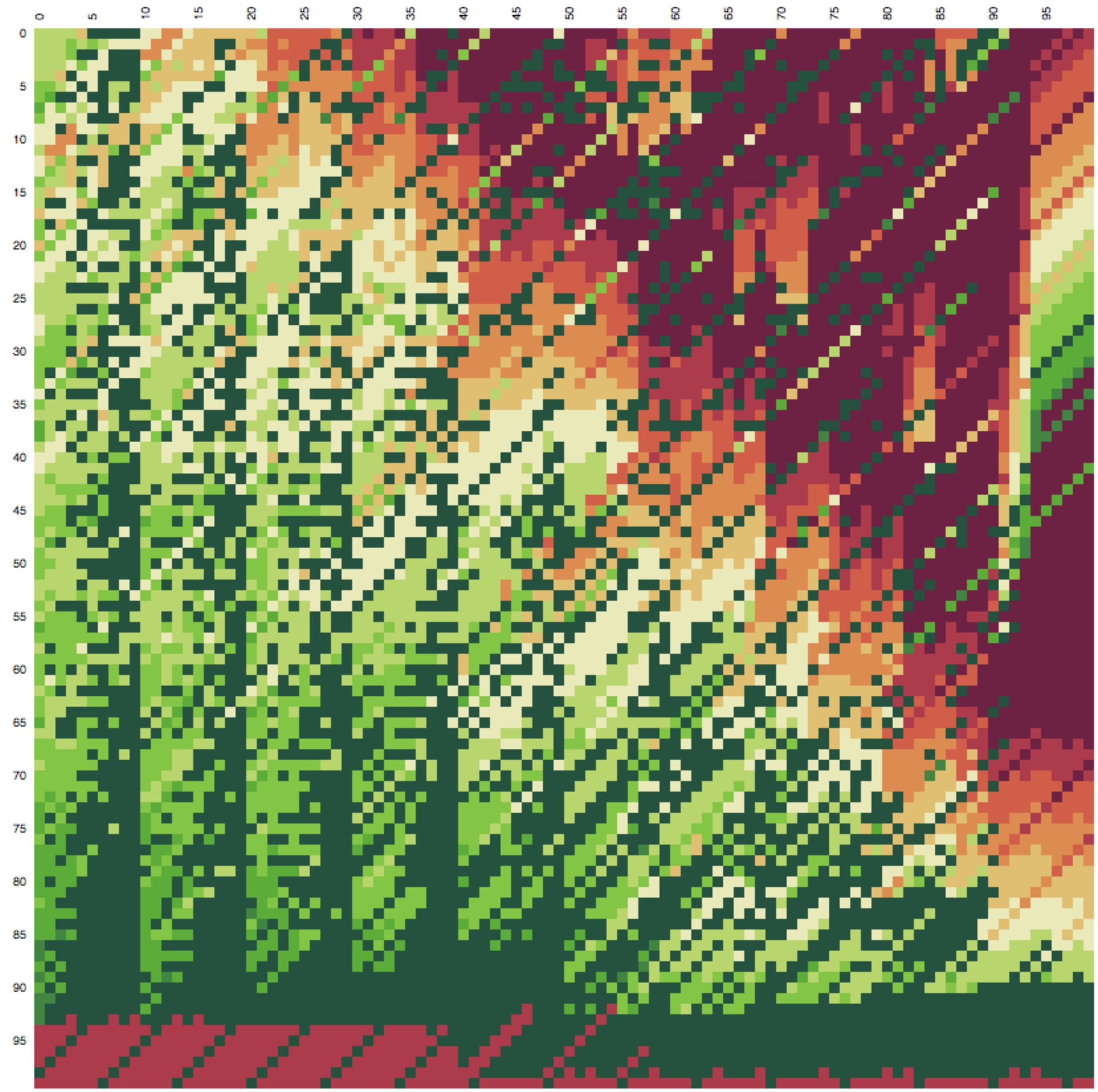
Hog Contest Results

Excellent participation!

51 qualified submissions

Lots of excellent ideas

(Results)



Type Coercion

Review: Type Dispatching Analysis

Minimal violation of abstraction barriers: we define cross-type functions as necessary.

Extensible: Any new numeric type can "install" itself into the existing system by adding new entries to the cross-type function dictionaries

Review: Type Dispatching Analysis

```
Minimal violation of abstraction barriers: we define cross-type functions as necessary.

Extensible: Any new numeric type can "install" itself into the existing system by adding
new entries to the cross-type function dictionaries
```

Arg 1	**Arg 2**	**Add**	**Multiply**
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

Coercion

Coercion

`Idea: Some types can be converted into other types`

Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

```
def rational_to_complex(r):
    """Return complex equal to rational."""
    return ComplexRI(r.numer/r.denom, 0)
```

Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

```
def rational_to_complex(r):
    """Return complex equal to rational."""
    return ComplexRI(r.numer/r.denom, 0)
```

Question: Can any numeric type be coerced into any other?

Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

```
def rational_to_complex(r):
    """Return complex equal to rational."""
    return ComplexRI(r.numer/r.denom, 0)
```

Question: Can any numeric type be coerced into any other?

Question: Can any two numeric types be coerced into a common type?

Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

```
def rational_to_complex(r):
    """Return complex equal to rational."""
    return ComplexRI(r.numer/r.denom, 0)
```

Question: Can any numeric type be coerced into any other?

Question: Can any two numeric types be coerced into a common type?

Question: Is coercion exact?

Applying Operators with Coercion

Applying Operators with Coercion

```
class Number:
```

Applying Operators with Coercion

```
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)
```

Applying Operators with Coercion

```
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)
```

Always defer to add method

Applying Operators with Coercion

```
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
```

Always defer to add method

Applying Operators with Coercion

```
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
```

Always defer to add method

Applying Operators with Coercion

```
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
```

Always defer to add method

Same interface: no change required

Applying Operators with Coercion

```
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
```

Always defer to add method

Same interface: no change required

Applying Operators with Coercion

```
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
```

Always defer to add method

Same interface: no change required

```
coercions = {('rat', 'com'): rational_to_complex}
```

Applying Operators with Coercion

```
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
            return (self.coerce_to(other.type_tag), other)
```

Always defer to add method

Same interface: no change required

```
coercions = {('rat', 'com'): rational_to_complex}
```

Applying Operators with Coercion

```
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
            return (self.coerce_to(other.type_tag), other)


    def coerce_to(self, other_tag):
        coercion_fn = self.coercions[(self.type_tag, other_tag)]
        return coercion_fn(self)

   coercions = {('rat', 'com'): rational_to_complex}
```

Always defer to add method

Same interface: no change required

Applying Operators with Coercion

```
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
            return (self.coerce_to(other.type_tag), other)
        elif (other.type_tag, self.type_tag) in self.coercions:
            return (self, other.coerce_to(self.type_tag))

    def coerce_to(self, other_tag):
        coercion_fn = self.coercions[(self.type_tag, other_tag)]
        return coercion_fn(self)

   coercions = {('rat', 'com'): rational_to_complex}
```

Always defer to add method

Same interface: no change required

Applying Operators with Coercion

```
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
            return (self.coerce_to(other.type_tag), other)
        elif (other.type_tag, self.type_tag) in self.coercions:
            return (self, other.coerce_to(self.type_tag))

    def coerce_to(self, other_tag):
        coercion_fn = self.coercions[(self.type_tag, other_tag)]
        return coercion_fn(self)

coercions = {('rat', 'com'): rational_to_complex}
```

Always defer to add method

Same interface: no change required

(Demo)

Coercion Analysis

Coercion Analysis

```
Minimal violation of abstraction barriers: we define cross-type coercion as necessary
```

Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary

Requires that all types can be coerced into a common type

Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

Arg 1	Arg 2	Add	Multiply
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

Arg 1	Arg 2	Add	Multiply
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

From	To	Coerce
Complex	Rational	
Rational	Complex	

Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

Arg 1	Arg 2	Add	Multiply
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

From	To	Coerce
Complex	Rational	
Rational	Complex	

Type	Add	Multiply
Complex		
Rational		

Linked Lists

Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list

Linked List Structure

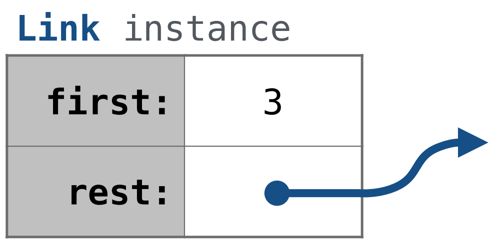
A linked list is either empty **or** a first value and the rest of the linked list

3 , 4 , 5

Linked List Structure

`A linked list is either empty **or** a first value and the rest of the linked list`

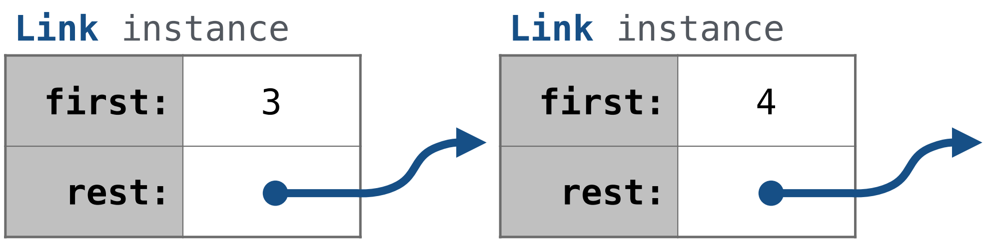
3 , 4 , 5



Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list

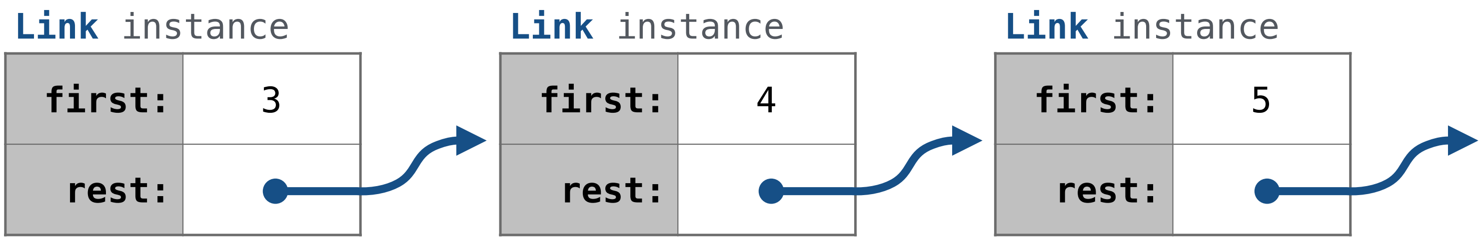
3 , 4 , 5



Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list

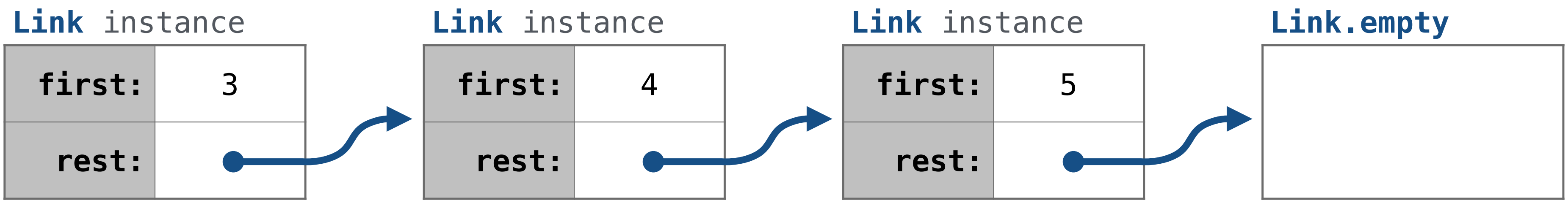
3 , 4 , 5



Linked List Structure

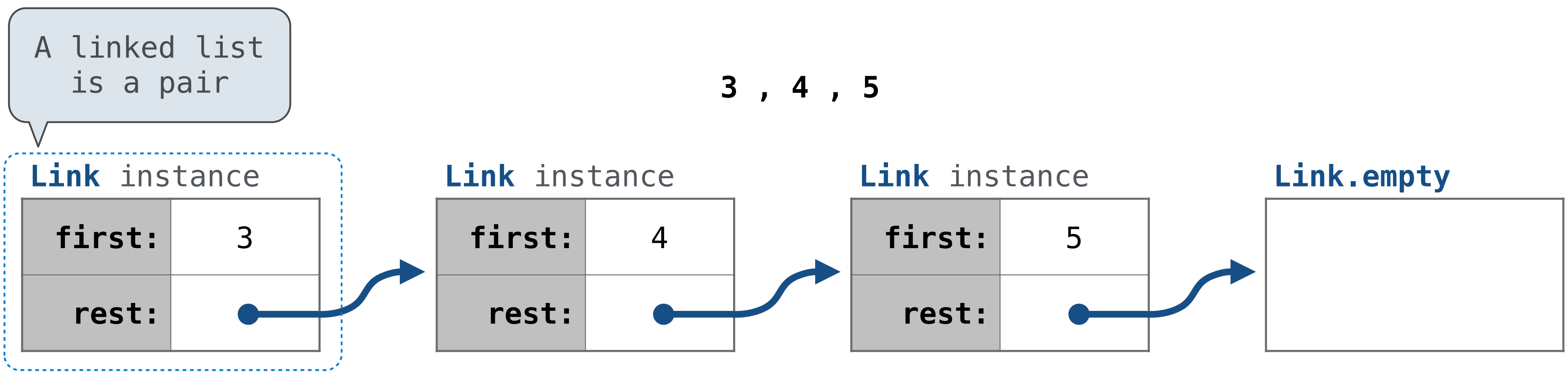
A linked list is either empty **or** a first value and the rest of the linked list

3 , 4 , 5



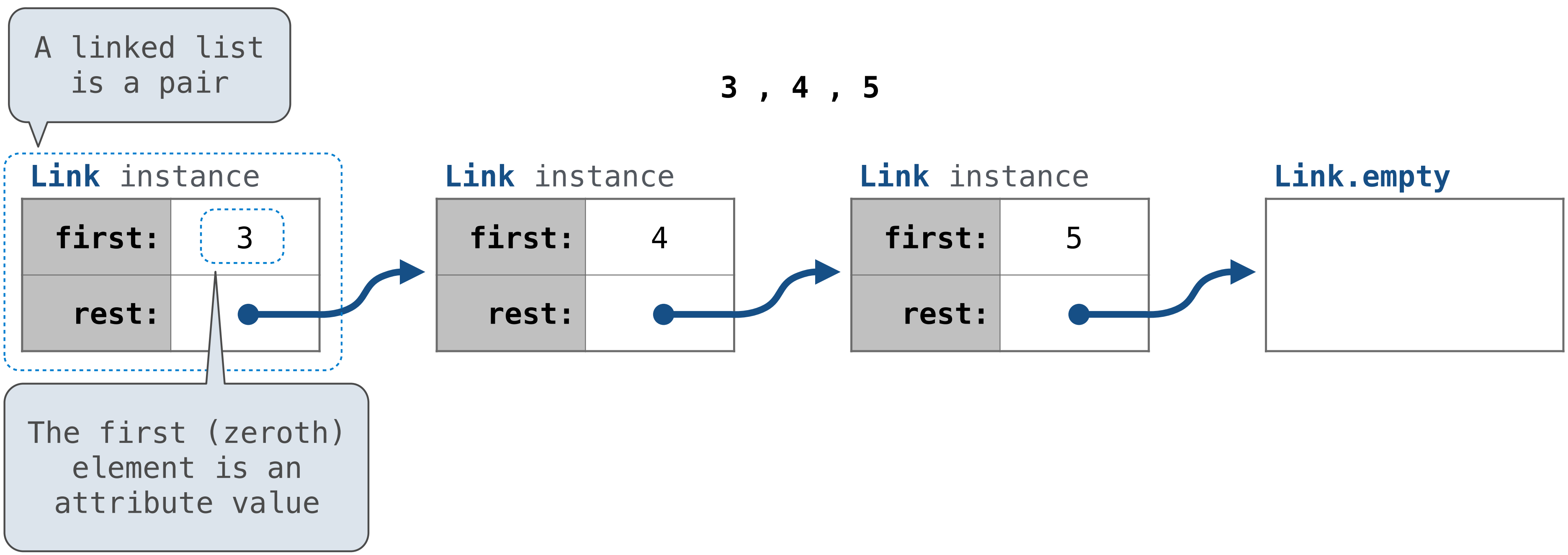
Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



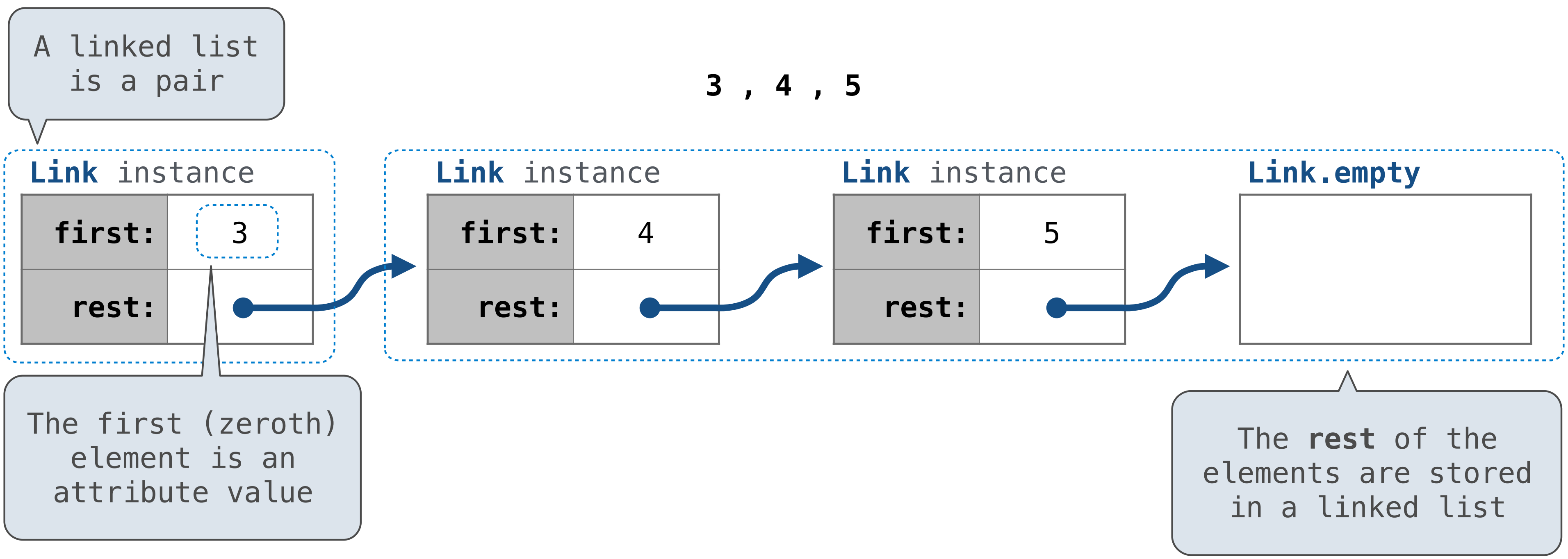
Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



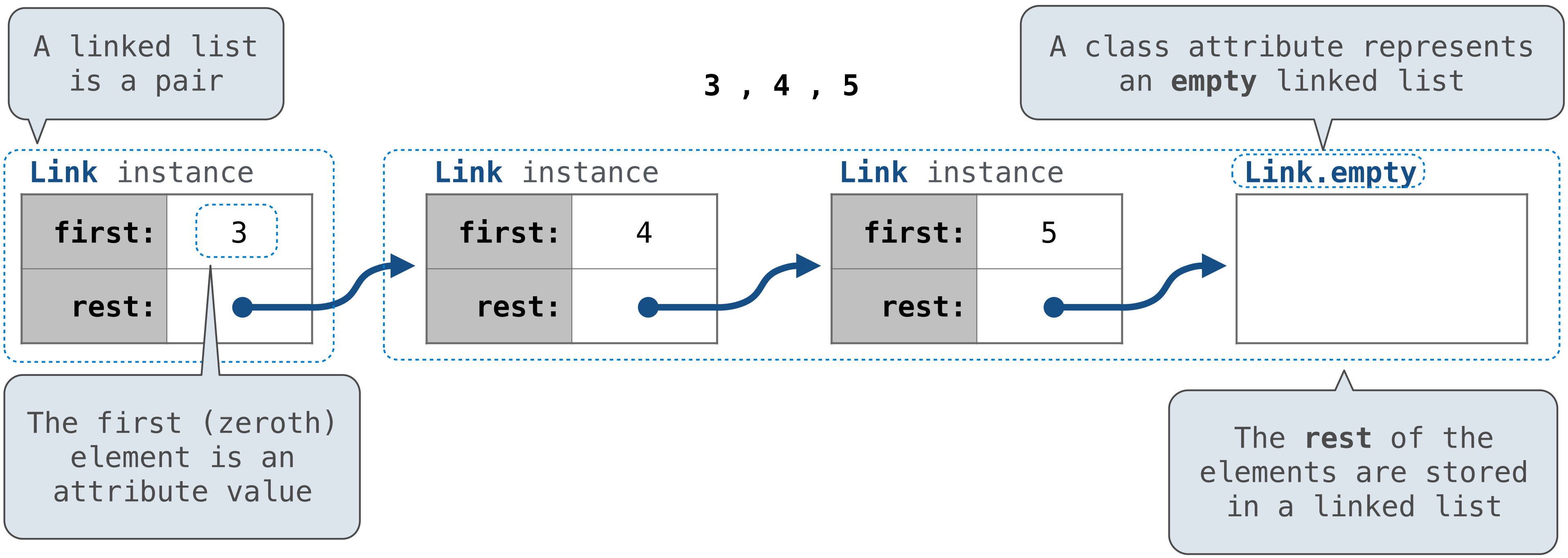
Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



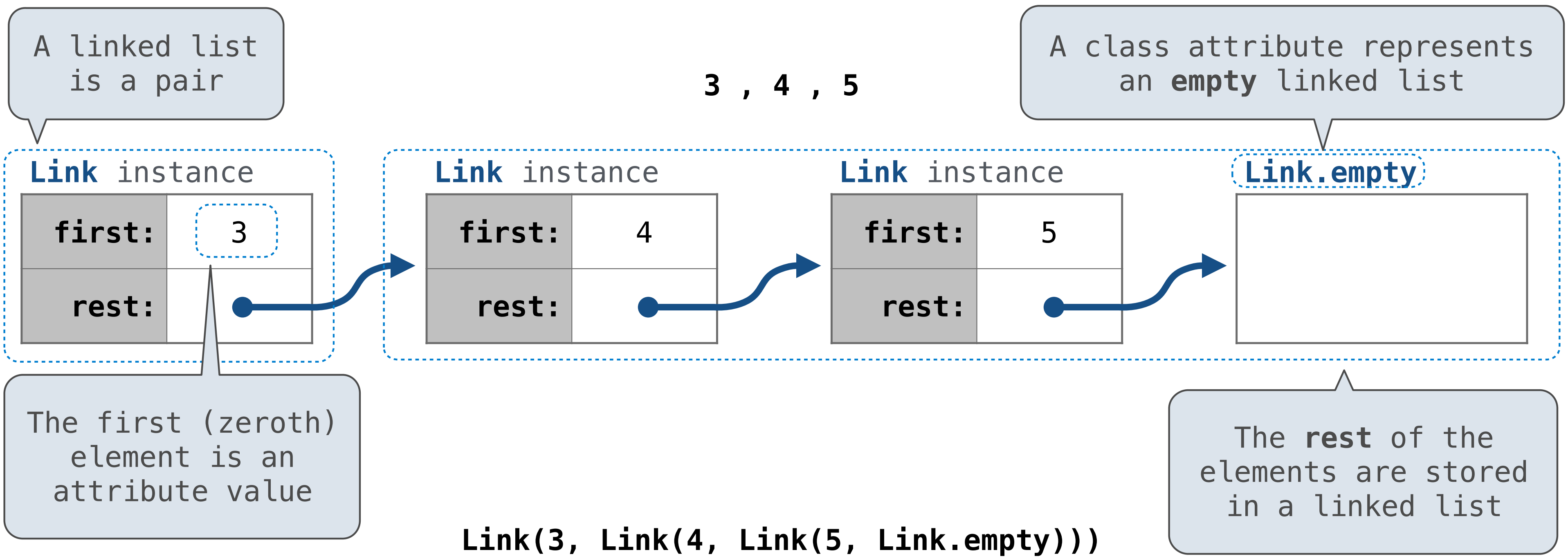
Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



Linked List Structure

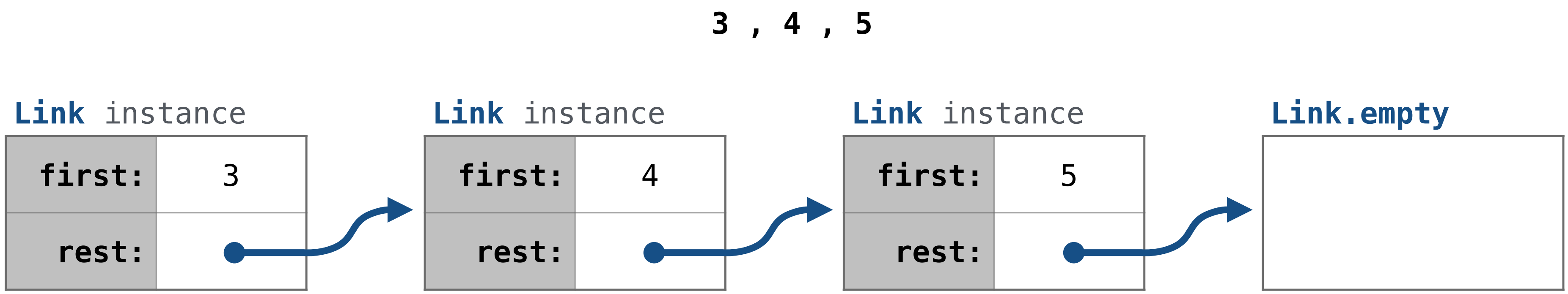
A linked list is either empty **or** a first value and the rest of the linked list



Link(3, Link(4, Link(5, Link.empty)))

Linked List Structure

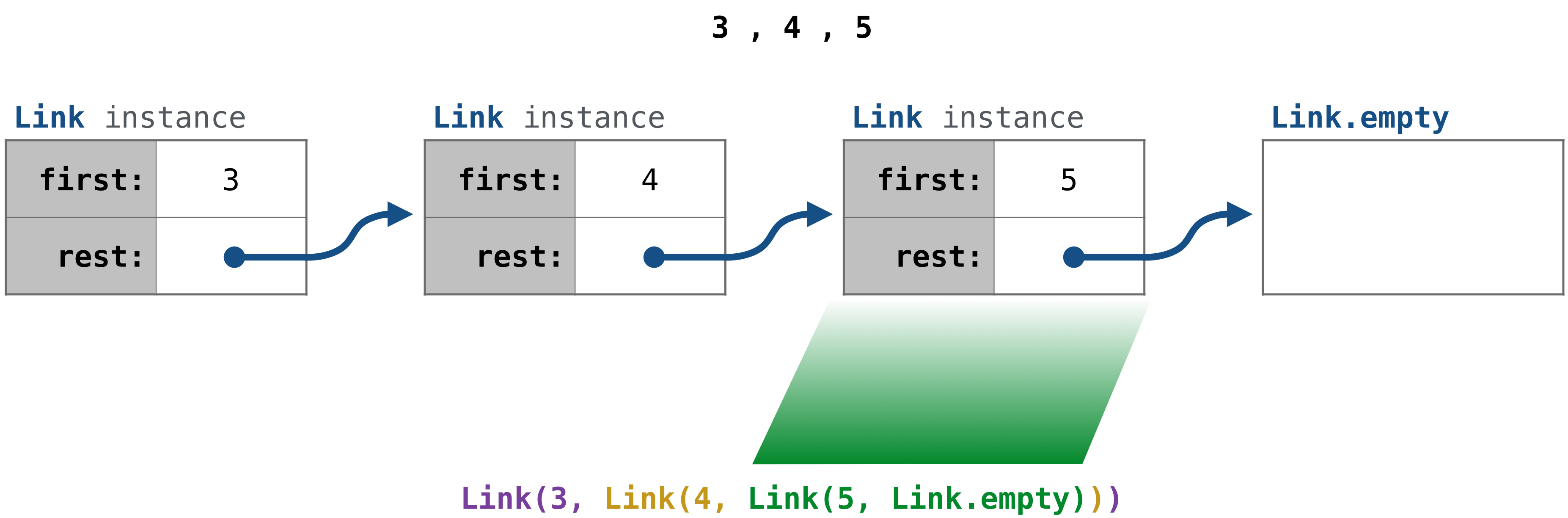
A linked list is either empty **or** a first value and the rest of the linked list




```
Link(3, Link(4, Link(5, Link.empty)))
```

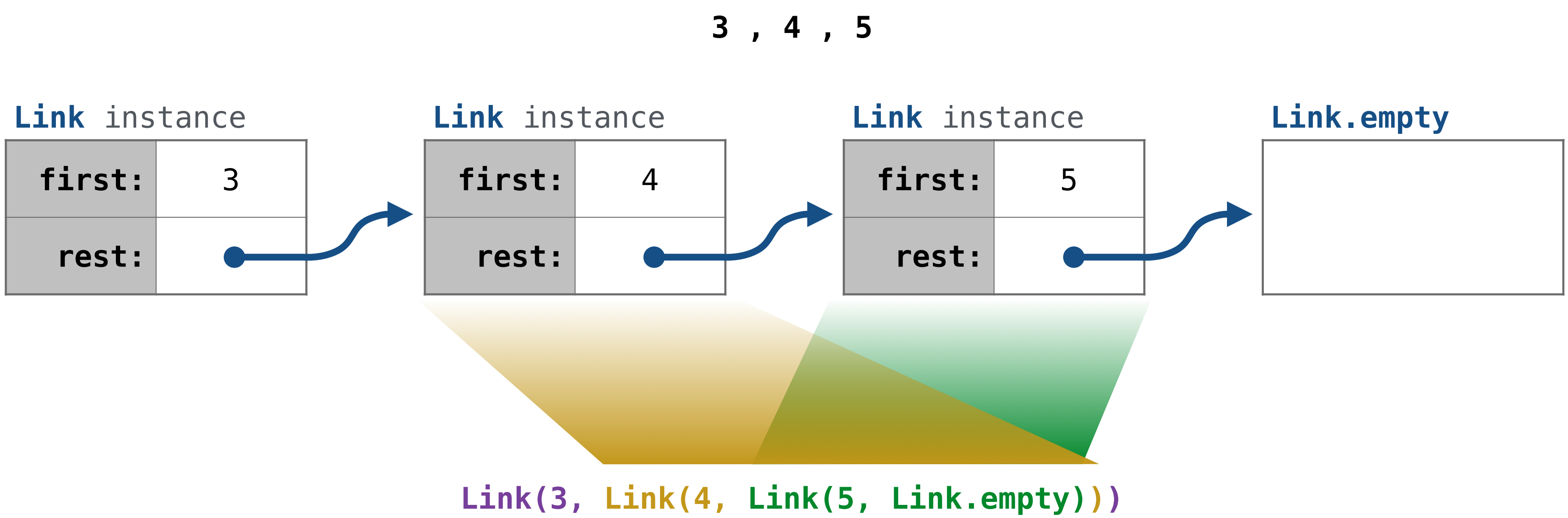
Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



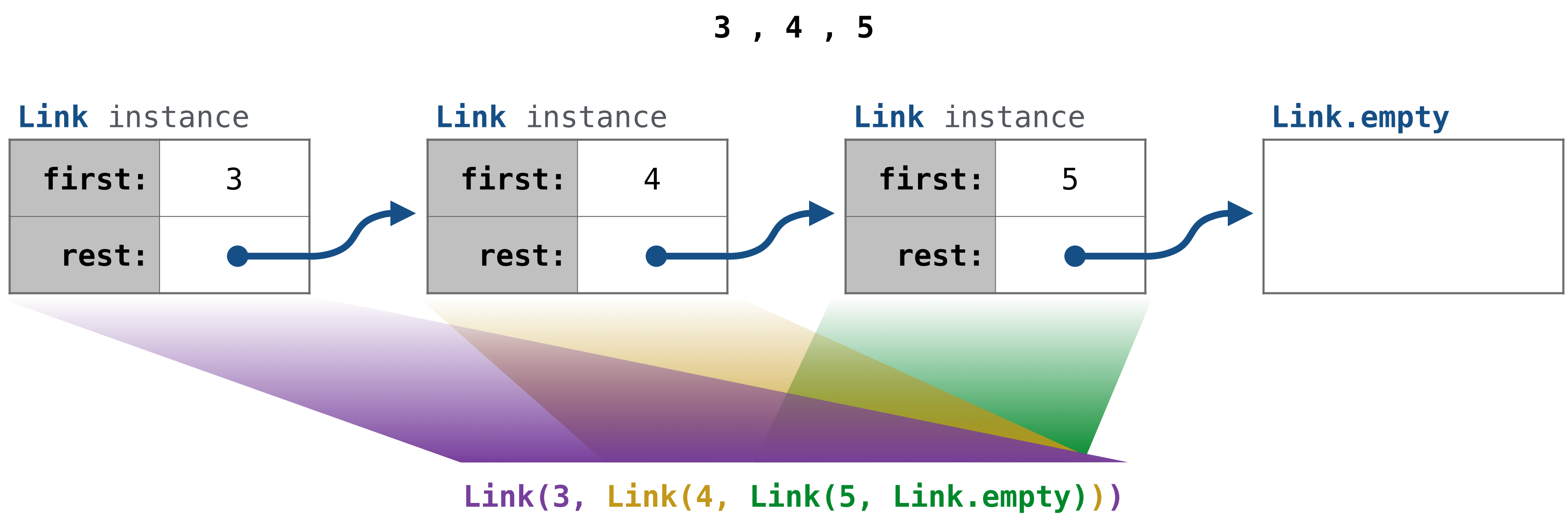
Linked List Structure

`A linked list is either empty` **`or`** `a first value and the rest of the linked list`



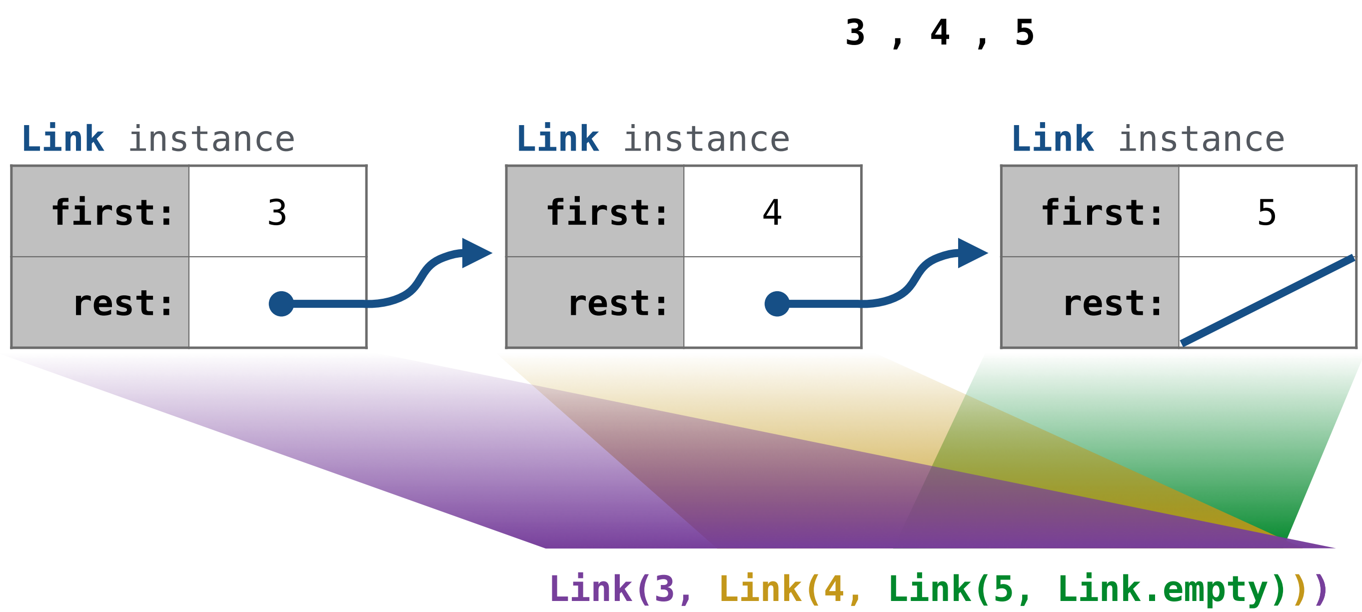
Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



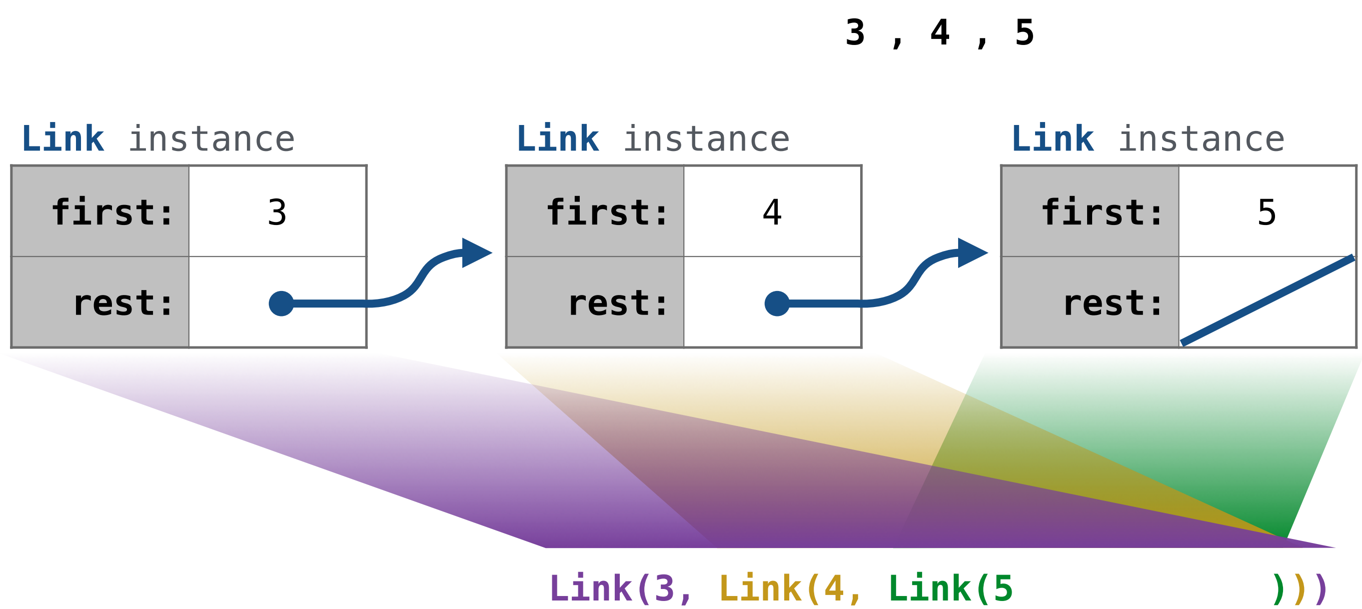
Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



Linked List Class

```
Link(3, Link(4, Link(5          )))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

```
Link(3, Link(4, Link(5          )))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:
```

```
Link(3, Link(4, Link(5            )))
```

Linked List Class

Linked list class: attributes are passed to __init__

```
class Link:

    def __init__(self, first, rest=empty):
```

```
Link(3, Link(4, Link(5          )))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
```

```
Link(3, Link(4, Link(5          )))
```

Linked List Class

Linked list class: attributes are passed to __init__

```
class Link:

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

```
Link(3, Link(4, Link(5            )))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

Returns whether rest is a Link

```
Link(3, Link(4, Link(5          )))
```

Linked List Class

Linked list class: attributes are passed to __init__

```
class Link:

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

Returns whether rest is a Link

help(isinstance): Return whether an object is an instance of a class or of a subclass thereof.

Link(3, Link(4, Link(5)))

Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

Returns whether rest is a Link

help(isinstance): Return whether an object is an instance of a class or of a subclass thereof.

Link(3, Link(4, Link(5)))

Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

Some zero-length sequence

Returns whether rest is a Link

help(isinstance): Return whether an object is an instance of a class or of a subclass thereof.

`Link(3, Link(4, Link(5          )))`

Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

Some zero-length sequence

Returns whether rest is a Link

help(isinstance): Return whether an object is an instance of a class or of a subclass thereof.

`Link(3, Link(4, Link(5          )))`

(Demo)

Sequence Operations

Linked List Class

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked lists are sequences

More special method names:	
`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked lists are sequences

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert ...
        self.first = first
        self.rest = rest
```

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked lists are sequences

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert ...
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]
```

More special method names:

__getitem__	Element selection []
__len__	Built-in len function

Linked List Class

Linked lists are sequences

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert ...
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]
```

This element selection syntax

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked lists are sequences

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert ...
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]
```

Calls this method

This element selection syntax

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked lists are sequences

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert ...
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]

    def __len__(self):
        return 1 + len(self.rest)
```

Calls this method

This element selection syntax

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked lists are sequences

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert ...
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]

    def __len__(self):
        return 1 + len(self.rest)
```

Calls this method (`__getitem__`)

This element selection syntax (`[i-1]`)

Recursive call to `__len__` (`len(self.rest)`)

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked lists are sequences

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert ...
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]

    def __len__(self):
        return 1 + len(self.rest)
```

Calls this method

This element selection syntax

Recursive call to __len__

More special method names:

__getitem__	Element selection []
__len__	Built-in len function

Methods can be recursive too!

(Demo)

Linked List Processing

(Demo)