Announcements
Announcements

- Homework 6 is due today at 11:59pm
Announcements

- Homework 6 is due today at 11:59pm
- Project 3 is due 7/26 at 11:59pm
Announcements

- Homework 6 is due today at 11:59pm
- Project 3 is due 7/26 at 11:59pm
  - Earn 1 EC point for completing it by 7/25
Announcements

• Homework 6 is due today at 11:59pm
• Project 3 is due 7/26 at 11:59pm
  • Earn 1 EC point for completing it by 7/25
• Quiz 5 tomorrow at the beginning of lecture
Announcements

• Homework 6 is due today at 11:59pm
• Project 3 is due 7/26 at 11:59pm
  • Earn 1 EC point for completing it by 7/25
• Quiz 5 tomorrow at the beginning of lecture
  • May cover mutability, OOP I (Monday)
Announcements

• Homework 6 is due today at 11:59pm
• Project 3 is due 7/26 at 11:59pm
  • Earn 1 EC point for completing it by 7/25
• Quiz 5 tomorrow at the beginning of lecture
  • May cover mutability, OOP I (Monday)
• Project 1 revisions due 7/27 at 11:59pm
Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications
This week (Objects), the goals are:
This week (Objects), the goals are:

- To learn the paradigm of object-oriented programming
This week (Objects), the goals are:

- To learn the paradigm of object-oriented programming
- To study applications of, and problems that be solved using, OOP
Practical OOP
Checking Types (and Accounts)
• We often check the type of an object to determine what operations it permits
Checking Types (and Accounts)

- We often check the type of an object to determine what operations it permits.

```python
>>> a = Account('Brian')
>>> ch = CheckingAccount('Brian')
```
Checking Types (and Accounts) (demo)

- We often check the type of an object to determine what operations it permits.

```python
>>> a = Account('Brian')
>>> ch = CheckingAccount('Brian')
```
• We often check the type of an object to determine what operations it permits

• The `type` built-in function returns the class that its argument is an instance of

```python
>>> a = Account('Brian')
>>> ch = CheckingAccount('Brian')
```
Checking Types (and Accounts)  

- We often check the type of an object to determine what operations it permits.
- The `type` built-in function returns the class that its argument is an instance of.

```python
>>> a = Account('Brian')
>>> ch = CheckingAccount('Brian')

>>> type(a) == Account
True

>>> type(ch) == Account
False

>>> type(ch) == CheckingAccount
True
```
• We often check the type of an object to determine what operations it permits

• The `type` built-in function returns the class that its argument is an instance of

• The `isinstance` built-in function returns whether its first argument (object) is an instance of the second argument (class) or a subclass

```python
>>> a = Account('Brian')
>>> ch = CheckingAccount('Brian')

>>> type(a) == Account
True

>>> type(ch) == Account
False

>>> type(ch) == CheckingAccount
True
```
Checking Types (and Accounts)

- We often check the type of an object to determine what operations it permits.
- The `type` built-in function returns the class that its argument is an instance of.
- The `isinstance` built-in function returns whether its first argument (object) is an instance of the second argument (class) or a subclass.

```python
>>> a = Account('Brian')
>>> ch = CheckingAccount('Brian')

>>> type(a) == Account
True
>>> type(ch) == Account
False
>>> type(ch) == CheckingAccount
True

>>> isinstance(a, Account)
True
>>> isinstance(ch, Account)
True
>>> isinstance(a, CheckingAccount)
False
>>> isinstance(ch, CheckingAccount)
True
```
Checking Types (and Accounts)  (demo)

• We often check the type of an object to determine what operations it permits.

• The **type** built-in function returns the class that its argument is an instance of.

• The **isinstance** built-in function returns whether its first argument (object) is an instance of the second argument (class) or a subclass.

• `isinstance(obj, cls)` is usually preferred over `type(obj) == cls`.

```python
>>> a = Account('Brian')
>>> ch = CheckingAccount('Brian')

>>> type(a) == Account
True

>>> type(ch) == Account
False

>>> type(ch) == CheckingAccount
True

>>> isinstance(a, Account)
True

>>> isinstance(ch, Account)
True

>>> isinstance(a, CheckingAccount)
False

>>> isinstance(ch, CheckingAccount)
True
```
Python's Magic Methods
Python's Magic Methods

(demo)
Python's Magic Methods

given:

```python
>>> x = Rational(3, 5)
>>> y = Rational(1, 3)
>>> y
Rational(1, 3)
```
Python's Magic Methods

• How does the Python interpreter display values?

```python
given
>>> x = Rational(3, 5)
>>> y = Rational(1, 3)
>>> y
Rational(1, 3)
```
Python's Magic Methods

• How does the Python interpreter display values?
  • First, it evaluates the expression to some value

```python
>>> x = Rational(3, 5)
>>> y = Rational(1, 3)
>>> y
Rational(1, 3)
```
How does the Python interpreter display values?

- First, it evaluates the expression to some value
- Then, it calls `repr` on that value and prints that string

```python
>>> x = Rational(3, 5)
>>> y = Rational(1, 3)
>>> y
Rational(1, 3)
```
• How does the Python interpreter display values?
  • First, it evaluates the expression to some value
  • Then, it calls repr on that value and prints that string

```python
>>> x = Rational(3, 5)
>>> y = Rational(1, 3)
>>> y
Rational(1, 3)
>>> repr(y)
'Rational(1, 3)'
>>> print(repr(y))
Rational(1, 3)
```
• How does the Python interpreter display values?
  • First, it evaluates the expression to some value
  • Then, it calls `repr` on that value and prints that string

• How do magic methods work?

```python
>>> x = Rational(3, 5)
>>> y = Rational(1, 3)
>>> y
Rational(1, 3)
>>> repr(y)
'Rational(1, 3)'
>>> print(repr(y))
Rational(1, 3)
```
Python's Magic Methods

• How does the Python interpreter display values?
  • First, it evaluates the expression to some value
  • Then, it calls repr on that value and prints that string

• How do magic methods work?

    >>> x = Rational(3, 5)
    >>> y = Rational(1, 3)
    >>> y
    Rational(1, 3)
    >>> repr(y)
    'Rational(1, 3)'
    >>> print(repr(y))
    Rational(1, 3)
    >>> x * y
    Rational(1, 5)
    >>> x.__mul__(y)
    Rational(1, 5)
How does the Python interpreter display values?

- First, it evaluates the expression to some value.
- Then, it calls `repr` on that value and prints that string.

How do magic methods work?

Are integers objects too? (Yep!)

```python
>>> x = Rational(3, 5)
>>> y = Rational(1, 3)
>>> y
Rational(1, 3)
>>> repr(y)
'Rational(1, 3)'
>>> print(repr(y))
Rational(1, 3)
>>> x * y
Rational(1, 5)
>>> x.__mul__(y)
Rational(1, 5)
```
Python's Magic Methods

• How does the Python interpreter display values?
  • First, it evaluates the expression to some value
  • Then, it calls repr on that value and prints that string

• How do magic methods work?
• Are integers objects too? (Yep!)
• Are ___ objects too? (Yep!)

```
>>> x = Rational(3, 5)
>>> y = Rational(1, 3)
>>> y
Rational(1, 3)
>>> repr(y)
'Rational(1, 3)'
>>> print(repr(y))
Rational(1, 3)
>>> x * y
Rational(1, 5)
>>> x.__mul__(y)
Rational(1, 5)
```
Linked Lists
The Link Class
The Link Class

empty = 'X'

def link(first, rest=empty):
    return [first, rest]

def first(lnk):
    return lnk[0]

def rest(lnk):
    return lnk[1]
The Link Class

empty = 'X'

def link(first, rest=empty):
    return [first, rest]

def first(lnk):
    return lnk[0]

def rest(lnk):
    return lnk[1]

>>> link_adt = link(1,
    link(2,
    link(3)))

>>> first(rest(link_adt))
2
The Link Class

```python
empty = 'X'

def link(first, rest=empty):
    return [first, rest]

def first(lnk):
    return lnk[0]

def rest(lnk):
    return lnk[1]

>>> link_adt = link(1,
    link(2,
    link(3)))

>>> first(rest(link_adt))
2
```
The Link Class

```python
empty = 'X'

class Link:
    def link(first, rest=empty):
        return [first, rest]

    def first(lnk):
        return lnk[0]

    def rest(lnk):
        return lnk[1]

>>> link_adt = link(1,
                  link(2,
                       link(3)))

>>> first(rest(link_adt))
2
```
The Link Class

empty = 'X'

def link(first, rest=empty):
    return [first, rest]

def first(lnk):
    return lnk[0]

def rest(lnk):
    return lnk[1]

>>> link_adt = link(1,
    link(2,
    link(3)))

>>> first(rest(link_adt))
2
The Link Class

empty = 'X'

def link(first, rest=empty):
    return [first, rest]

def first(lnk):
    return lnk[0]

def rest(lnk):
    return lnk[1]

>>> link_adt = link(1,
                   link(2,
                        link(3)))
>>> first(rest(link_adt))
2
The Link Class

```python
empty = 'X'

def link(first, rest=empty):
    return [first, rest]

def first(lnk):
    return lnk[0]

def rest(lnk):
    return lnk[1]

>>> link_adt = link(1,
                   link(2,
                        link(3)))
>>> first(rest(link_adt))
2

class Link:
    empty = ()
    
    def __init__(self, first, rest=empty):
        self.first = first
```
The Link Class

empty = 'X'

def link(first, rest=empty):
    return [first, rest]

def first(lnk):
    return lnk[0]

def rest(lnk):
    return lnk[1]

>>> link_adt = link(1,
    link(2,
    link(3)))

>>> first(rest(link_adt))
2

class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
The Link Class

```python
class Link:
    empty = ()

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

def link(first, rest=empty):
    return [first, rest]

def first(lnk):
    return lnk[0]

def rest(lnk):
    return lnk[1]

>>> link_adt = link(1,
    link(2,
    link(3)))
>>> first(rest(link_adt))
2
```

```python
empty = 'X'

def link(first, rest=empty):
    return [first, rest]

def first(lnk):
    return lnk[0]

def rest(lnk):
    return lnk[1]

>>> link_adt = link(1,
    link(2,
    link(3)))
>>> first(rest(link_adt))
2
```
The Link Class

empty = 'X'

def link(first, rest=empty):
    return [first, rest]

def first(lnk):
    return lnk[0]

def rest(lnk):
    return lnk[1]

>>> link_adt = link(1,
    link(2,
    link(3)))
>>> first(rest(link_adt))
2

class Link:
    empty = ()

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

>>> link_cls = Link(1,
    Link(2,
    Link(3)))
>>> link_cls.rest.first
The Link Class

empty = 'X'

def link(first, rest=empty):
    return [first, rest]

def first(lnk):
    return lnk[0]

def rest(lnk):
    return lnk[1]

>>> link_adt = link(1,
    link(2,
    link(3)))

>>> first(rest(link_adt))
2

class Link:
    empty = ()

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

>>> link_cls = Link(1,
    Link(2,
    Link(3)))

>>> link_cls.rest.first
2
Mutable Linked Lists
Mutable Linked Lists

- Instances of user-defined classes are mutable by default
Mutable Linked Lists

- Instances of user-defined classes are mutable by default
Instances of user-defined classes are mutable by default

```python
class Link:
    empty = ()

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

    def __repr__(self):
        if self.rest is Link.empty:
            return 'Link({0})'.format(self.first)
        else:
            return 'Link({0}, {1})'.format(self.first, repr(self.rest))
```
Linked Lists are Sequences
Linked Lists are Sequences

class Link:
Linked Lists are Sequences

class Link:
    empty = ()
Linked Lists are Sequences

class Link:
    empty = ()

...

def __getitem__(self, i):
Linked Lists are Sequences

```python
class Link:
    empty = ()

...

def __getitem__(self, i):
    if i == 0:
```
Linked Lists are Sequences

class Link:
    empty = ()

...

def __getitem__(self, i):
    if i == 0:
        return self.first
Linked Lists are Sequences

class Link:
    empty = ()

...

def __getitem__(self, i):
    if i == 0:
        return self.first
    elif self.rest is Link.empty:
Linked Lists are Sequences

```python
class Link:
    empty = ()

...

def __getitem__(self, i):
    if i == 0:
        return self.first
    elif self.rest is Link.empty:
        raise IndexError('...')
```
Linked Lists are Sequences

class Link:
    empty = ()

...

def __getitem__(self, i):
    if i == 0:
        return self.first
    elif self.rest is Link.empty:
        raise IndexError('...')
    else:
Linked Lists are Sequences

```python
class Link:
    empty = ()

...

def __getitem__(self, i):
    if i == 0:
        return self.first
    elif self.rest is Link.empty:
        raise IndexError('...')
    else:
        return self.rest[i - 1]
```
Linked Lists are Sequences

class Link:
    empty = ()
    ...

def __getitem__(self, i):
    if i == 0:
        return self.first
    elif self.rest is Link.empty:
        raise IndexError('...')
    else:
        return self.rest[i - 1]

Sneaky recursive call: equivalent to self.rest.__getitem__(i-1)
class Link:
    empty = ()
    ...

def __getitem__(self, i):
    if i == 0:
        return self.first
    elif self.rest is Link.empty:
        raise IndexError('...')
    else:
        return self.rest[i - 1]

def __len__(self):

Linked Lists are Sequences
Linked Lists are Sequences

```python
class Link:
    empty = ()

    def __getitem__(self, i):
        if i == 0:
            return self.first
        elif self.rest is Link.empty:
            raise IndexError('...')
        else:
            return self.rest[i - 1]

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

Sneaky recursive call: equivalent to `self.rest.__getitem__(i-1)`
Linked Lists are Sequences

```python
class Link:
    empty = ()

    ...  # Omitted for brevity

    def __getitem__(self, i):
        if i == 0:
            return self.first
        elif self.rest is Link.empty:
            raise IndexError('...')
        else:
            return self.rest[i - 1]

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

Sneaky recursive call: equivalent to `self.rest.__getitem__(i-1)`

Another sneaky recursive call: equivalent to `self.rest.__len__()`
Linked Lists are Sequences

```python
class Link:
    empty = ()

    def __getitem__(self, i):
        if i == 0:
            return self.first
        elif self.rest is Link.empty:
            raise IndexError('...')
        else:
            return self.rest[i - 1]

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

Sneaky recursive call: equivalent to `self.rest.__getitem__(i-1)`

Another sneaky recursive call: equivalent to `self.rest.__len__()`

Where's the base case??
Linked Lists are Sequences

```python
class Link:
    empty = ()

...  # code not shown

def __getitem__(self, i):
    if i == 0:
        return self.first
    elif self.rest is Link.empty:
        raise IndexError('...')
    else:
        return self.rest[i - 1]

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

Sneaky recursive call: equivalent to `self.rest.__getitem__(i-1)`

Another sneaky recursive call: equivalent to `self.rest.__len__()`

Where's the base case??
Another sneaky recursive call: equivalent to `self.rest.__len__()`

Where's the base case??

---

Here's the implementation of the `Link` class:

```python
class Link:
    empty = ()

    def __getitem__(self, i):
        if i == 0:
            return self.first
        elif self.rest is Link.empty:
            raise IndexError('...')
        else:
            return self.rest[i - 1]

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

Linked Lists are Sequences (demo)
The `__setitem__` Magic Method
The `__setitem__` Magic Method

```python
>>> s = Link(1, Link(2, Link(3)))
```
The __setitem__ Magic Method

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
```
The __setitem__ Magic Method

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
>>> s
```
The `__setitem__` Magic Method

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
>>> s
Link(1, Link(3, Link(3)))
```
The __setitem__ Magic Method

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
>>> s
Link(1, Link(3, Link(3)))
```

class Link:
The __setitem__ Magic Method

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
>>> s
Link(1, Link(3, Link(3)))

class Link:
    ...
The `__setitem__` Magic Method

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
>>> s
Link(1, Link(3, Link(3)))
```

class Link:
    ...
    def __setitem__(self, i, val):
```
The `__setitem__` Magic Method

```python
class Link:
    ...
    def __setitem__(self, i, val):
        if i == 0:
            s = Link(1, Link(2, Link(3)))
            s[1] = 3
            s
            Link(1, Link(3, Link(3)))
```
The \_\_setitem\_\_ Magic Method

```python
class Link:
...

def \_\_setitem\_\_(self, i, val):
    if i == 0:
        self.first = val

>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
>>> s
Link(1, Link(3, Link(3)))
```
The __setitem__ Magic Method

```python
class Link:
    ...

    def __setitem__(self, i, val):
        if i == 0:
            self.first = val
        elif self.rest is Link.empty:
```

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
>>> s
Link(1, Link(3, Link(3)))
```
The `__setitem__` Magic Method

```python
class Link:
    ...

def __setitem__(self, i, val):
    if i == 0:
        self.first = val
    elif self.rest is Link.empty:
        raise IndexError('...')
```

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
>>> s
Link(1, Link(3, Link(3)))
```
The `__setitem__` Magic Method

```python
class Link:
    ...
    def __setitem__(self, i, val):
        if i == 0:
            self.first = val
        elif self.rest is Link.empty:
            raise IndexError('...')
        else:
            ...
```

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
>>> s
Link(1, Link(3, Link(3)))
```
The __setitem__ Magic Method

```python
class Link:
...

def __setitem__(self, i, val):
    if i == 0:
        self.first = val
    elif self.rest is Link.empty:
        raise IndexError('...')
    else:
        self.rest[i - 1] = val
```

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
>>> s
Link(1, Link(3, Link(3)))
```
The **__setitem__** Magic Method  

```python
class Link:
  ...
  def __setitem__(self, i, val):
    if i == 0:
      self.first = val
    elif self.rest is Link.empty:
      raise IndexError('...')
    else:
      self.rest[i - 1] = val
```

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
>>> s
Link(1, Link(3, Link(3)))
```
Mutating Map
Mutating Map

```python
>>> s = Link(1, Link(2, Link(3)))
```
Mutating Map

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
```
Mutating Map

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
```
Mutating Map

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
Link(1, Link(4, Link(9)))
```
Mutating Map

```python
class Link:
    ...

>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))
```
Mutating Map

```python
class Link:
    ...
    def map(self, f):
        >>>> s = Link(1, Link(2, Link(3)))
        >>>> s.map(lambda x: x * x)
        >>>> s
        Link(1, Link(4, Link(9)))
```
Mutating Map

```python
class Link:
    ...
    def map(self, f):
        for i in range(len(self)):
Mutating Map

```python
class Link:
    ...

def map(self, f):
    for i in range(len(self)):
        self[i] = f(self[i])

>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))
```
class Link:
    
    def map(self, f):
        for i in range(len(self)):
            self[i] = f(self[i])

Mutating Map (demo)

>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))
Mutating Map (demo)

```python
class Link:
...

def map(self, f):
    for i in range(len(self)):
        self[i] = f(self[i])
```

Runtime?
Mutating Map

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

    def map(self, f):
        for i in range(len(self)):
            self[i] = f(self[i])
```

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))
```
Mutating Map

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

    def map(self, f):
        for i in range(len(self)):
            self[i] = f(self[i])

>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))
```
Mutating Map

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

    def map(self, f):
        for i in range(len(self)):
            self[i] = f(self[i])

>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))

self[0] = f(self[0])
self[1] = f(self[1])
```
Mutating Map

```python
class Link:
    ...

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

def map(self, f):
    for i in range(len(self)):
        self[i] = f(self[i])

>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))
```

```python
self[0] = f(self[0])
s[1] = f(s[1])
s[2] = f(s[2])
```
Mutating Map

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

    def map(self, f):
        for i in range(len(self)):
            self[i] = f(self[i])
```

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))
```

```python
>>> self[0] = f(self[0])
>>> self[1] = f(self[1])
>>> self[2] = f(self[2])
```
Mutating Map

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

def map(self, f):
    for i in range(len(self)):
        self[i] = f(self[i])

>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))
```
Mutating Map

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

    def map(self, f):
        for i in range(len(self)):
            self[i] = f(self[i])

>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))

```

\[ \Theta(n^2) \]
Mutating Map (Improved)

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))
```
Mutating Map (Improved)

>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))

class Link:
Mutating Map (Improved)

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))

class Link:
    ...
```
Mutating Map (Improved)

>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))

class Link:
    ...
    def map(self, f):

Mutating Map (Improved)

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))

class Link:
    ...
    def map(self, f):
        self.first = f(self.first)
```
Mutating Map (Improved)

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))

class Link:
    ...
    def map(self, f):
        self.first = f(self.first)
        if self.rest is not Link.empty:
```
Mutating Map (Improved)

```python
class Link:
    ...
    def map(self, f):
        self.first = f(self.first)
        if self.rest is not Link.empty:
            self.rest.map(f)
```

```python
global s
s = Link(1, Link(2, Link(3)))
s.map(lambda x: x * x)
s
Link(1, Link(4, Link(9)))
```
Mutating Map (Improved)

```python
class Link:
    ...

def map(self, f):
    self.first = f(self.first)
    if self.rest is not Link.empty:
        self.rest.map(f)

>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))
```

Runtime?
Mutating Map (Improved)

```python
class Link:
    ...
    def map(self, f):
        self.first = f(self.first)
        if self.rest is not Link.empty:
            self.rest.map(f)
```

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))
```

Runtime? \( \Theta(n) \)
Mutating Map (Improved)      (demo)

>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))

```
class Link:
    ...
    def map(self, f):
        self.first = f(self.first)
        if self.rest is not Link.empty:
            self.rest.map(f)
```
contains and in
contains and in

```python
>>> s = Link(1, Link(2, Link(3)))
>>> 2 in s
True
>>> 4 in s
False
```
contains and in  (demo)

```python
>>> s = Link(1, Link(2, Link(3)))
>>> 2 in s
True
>>> 4 in s
False
```
contains and in (demo)

```python
class Link:

>>> s = Link(1, Link(2, Link(3)))
>>> 2 in s
True
>>> 4 in s
False
```
contains and in (demo)

class Link:
    ...

>>> s = Link(1, Link(2, Link(3)))
>>> 2 in s
    True
>>> 4 in s
    False
```python
class Link:
    ...
    def __contains__(self, e):

>>> s = Link(1, Link(2, Link(3)))
>>> 2 in s
True
>>> 4 in s
False
```
contains and in (demo)

class Link:
  ...
  def __contains__(self, e):
    return self.first == e or e in self.rest

>>> s = Link(1, Link(2, Link(3)))
>>> 2 in s
True
>>> 4 in s
False
Break!
Environments
Environment Frames
Environment Frames

- An environment is a sequence of frames
Environment Frames

- An environment is a sequence of frames
  - Each frame has some data (bindings) and a parent, which points to another frame
Environment Frames

• An environment is a sequence of frames
  • Each frame has some data (bindings) and a parent, which points to another frame
• A linked list is a sequence of values
Environment Frames

- An environment is a sequence of frames
  - Each frame has some data (bindings) and a parent, which points to another frame
- A linked list is a sequence of values
  - Each link has some data (first) and a rest, which points to another link
Environment Frames

- An environment is a sequence of frames
  - Each frame has some data (bindings) and a parent, which points to another frame
- A linked list is a sequence of values
  - Each link has some data (first) and a rest, which points to another link
- An environment is just a special case of a linked list!
Environment Frames

• An environment is a sequence of frames
  • Each frame has some data (bindings) and a parent, which points to another frame

• A linked list is a sequence of values
  • Each link has some data (first) and a rest, which points to another link

• An environment is just a special case of a linked list!
Environment Frames

- An environment is a sequence of frames
  - Each frame has some data (bindings) and a parent, which points to another frame
- A linked list is a sequence of values
  - Each link has some data (first) and a rest, which points to another link
- An environment is just a special case of a linked list!
The Call Stack
The Call Stack

- A stack is a data structure that permits two operations
The Call Stack

- A stack is a data structure that permits two operations
The Call Stack

- A stack is a data structure that permits two operations
  - Add to the top of a stack ("push")
The Call Stack

• A stack is a data structure that permits two operations
  • Add to the top of a stack ("push")
  • Remove from the top of a stack ("pop")
The Call Stack

• A stack is a data structure that permits two operations
  • Add to the top of a stack ("push")
  • Remove from the top of a stack ("pop")

• Two new Link operations required: insert_front and remove_front
The Call Stack

- A *stack* is a data structure that permits two operations
  - Add to the top of a stack ("push")
  - Remove from the top of a stack ("pop")

- Two new Link operations required: insert_front and remove_front
The Call Stack

• A stack is a data structure that permits two operations
  • Add to the top of a stack ("push")
  • Remove from the top of a stack ("pop")

• Two new Link operations required: insert_front and remove_front
• A call stack keeps track of frames that are currently open
The Call Stack
(demo)

- A stack is a data structure that permits two operations
  - Add to the top of a stack ("push")
  - Remove from the top of a stack ("pop")

- Two new Link operations required: insert_front and remove_front
- A call stack keeps track of frames that are currently open
  - Calling a function adds a new frame to the stack
The Call Stack

- A stack is a data structure that permits two operations
  - Add to the top of a stack ("push")
  - Remove from the top of a stack ("pop")

- Two new Link operations required: insert_front and remove_front
- A call stack keeps track of frames that are currently open
  - Calling a function adds a new frame to the stack
  - Returning from a function removes that frame from the stack
The Call Stack

- A stack is a data structure that permits two operations
  - Add to the top of a stack ("push")
  - Remove from the top of a stack ("pop")

- Two new Link operations required: insert_front and remove_front

- A call stack keeps track of frames that are currently open
  - Calling a function adds a new frame to the stack
  - Returning from a function removes that frame from the stack
  - The current frame is always on the top of the stack
Python
Python
• What if we could have Python functions use the environment frames and the call stack that we just defined?
What if we could have Python functions use the environment frames and the call stack that we just defined?
Brython

• What if we could have Python functions use the environment frames and the call stack that we just defined?
Brython

• What if we could have Python functions use the environment frames and the call stack that we just defined?
Brython

• What if we could have Python functions use the environment frames and the call stack that we just defined?

• Two important parts:
Brython

• What if we could have Python functions use the environment frames and the call stack that we just defined?

• Two important parts:
  • What should happen when defining a Brython function?
Brython

• What if we could have Python functions use the environment frames and the call stack that we just defined?

• Two important parts:
  • What should happen when defining a Brython function?
  • What should happen when calling a Brython function?
Brython

• What if we could have Python functions use the environment frames and the call stack that we just defined?

• Two important parts:
  • What should happen when **defining** a Brython function?
  • What should happen when **calling** a Brython function?

A different Brython: [http://brython.info/](http://brython.info/)
Function Definitions
Function Definitions

• What happens in a function definition?
Function Definitions

• What happens in a function definition?
  • Determine the current frame of execution: this is the function's parent frame
Function Definitions

• What happens in a function definition?
  • Determine the current frame of execution: this is the function's parent frame
  • Bind the function name to the function value
Function Calls
Function Calls

- What happens in a function call?
Function Calls

• What happens in a function call?
  • Create a brand new call frame (using the function parent as the parent of that frame) and insert it into the stack
Function Calls

• What happens in a function call?
  • Create a brand new call frame (using the function parent as the parent of that frame) and insert it into the stack
  • Bind function's parameters to arguments
Function Calls

- What happens in a function call?
  - Create a brand new call frame (using the function parent as the parent of that frame) and insert it into the stack
  - Bind function's parameters to arguments
  - Execute the function in the environment of the call frame
Function Calls

• What happens in a function call?
  • Create a brand new call frame (using the function parent as the parent of that frame) and insert it into the stack
  • Bind function's parameters to arguments
  • Execute the function in the environment of the call frame
    • Remember: the current frame is at the top of the stack
Function Calls

- What happens in a function call?
  - Create a brand new call frame (using the function parent as the parent of that frame) and insert it into the stack
  - Bind function's parameters to arguments
  - Execute the function in the environment of the call frame
    - Remember: the current frame is at the top of the stack
  - After executing the function, remove the frame from the stack
Function Calls

• What happens in a function call?
  • Create a brand new call frame (using the function parent as the parent of that frame) and insert it into the stack
  • Bind function's parameters to arguments
  • Execute the function in the environment of the call frame
    • Remember: the current frame is at the top of the stack
  • After executing the function, remove the frame from the stack
Summary

- Linked lists are one way to store sequential data
Summary

• Linked lists are one way to store sequential data

• An object-based implementation of the linked list abstraction allows for easy mutability
Summary

• Linked lists are one way to store sequential data

• An object-based implementation of the linked list abstraction allows for easy mutability
  • No more crazy nonlocal stuff!
Summary

• Linked lists are one way to store sequential data

• An object-based implementation of the linked list abstraction allows for easy mutability
  
  • No more crazy nonlocal stuff!

• Implementing magic methods lets us hook into convenient Python syntax and built-in functions
Summary

• Linked lists are one way to store sequential data

• An object-based implementation of the linked list abstraction allows for easy mutability
  
  • No more crazy nonlocal stuff!

• Implementing magic methods lets us hook into convenient Python syntax and built-in functions

• Linked lists can be used to implement some of the core ideas of this course!