Announcements
Linked Lists
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
```
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
```
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

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

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
```
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link.

The rest of a linked list can contain the linked list as a sub-list.

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
guess = Link(1, Link(2, Link(3)))
guess.rest.first = 5
t = guess.rest
rest = t.rest
rest.first = s
s.first
5
guess.rest.rest.rest.rest.rest.first
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
2
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
2
```

Global frame

```
<table>
<thead>
<tr>
<th>First</th>
<th>Rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>First</th>
<th>Rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
```

Note: The actual environment diagram is much more complicated.
Environment Diagrams
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
Go Bears!

```python
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
```

```
Global frame
  oski

 func oski(bear)[parent=G]

 f1: oski [parent=G]

 Return Value

 Return Value

 Return Value

 Return Value

 Return Value
```
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
Go Bears!

```python
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
```

Return Value

Global frame
oski

f1: oski [parent=G]
    bear
    cal
    Return Value

f2: cal
    Return Value

Return Value

func oski(bear)[parent=G]
func abs(...)[parent=G]
func cal(berk)[parent=f1]
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
Go Bears!

```python
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
```

```
<table>
<thead>
<tr>
<th>Global frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>oski</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>f1: oski [parent=G]</th>
</tr>
</thead>
<tbody>
<tr>
<td>bear</td>
</tr>
<tr>
<td>cal</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>f2: cal [parent=f1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>berk</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Return Value</td>
</tr>
</tbody>
</table>
```

```
| return [berk+1, berk-1] |
```

```
| bear = lambda ley: berk-ley |
```

```
| return [berk, cal(berk)] |
```

```
| return cal(2) |
```

```
| oski(abs) |
```
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
Go Bears!

```python
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
```

```
Global frame
oski

f1: oski [parent=G]
    bear
    cal
    Return Value

f2: cal [parent=f1]
    berk 2
    Return Value
```

Return Value

`func oski(bear)[parent=G]`

`func λ(ley) [parent=f2]`

`func cal(berk) [parent=f1]`
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
    oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
    return oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
Go Bears!

```python
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
```

Return Value

```
Global frame
osi
```

```
f1: oski [parent=G]
    bear
    cal
    Return Value
```

```
f2: cal [parent=f1]
    berk 2
    Return Value
```

```
f3: cal [parent=f1]
    berk 2
    Return Value
```

```
f4: λ [parent=f2]
    ley 2
    Return Value
```

func oski(bear) [parent=G]

func λ(ley) [parent=f2]

func cal(berk) [parent=f1]
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
    oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
Go Bears!

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
Objects
Land Owners

Instance attributes are found before class attributes; class attributes are inherited
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
greeting = 'Sir'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting
Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):

Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

>>> Worker().work()

>>> jack

>>> jack.work()

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

>>> jack.work()

>>> john.work()

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> john.elf.work(john)
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker() >>> Worker().work() <<<
john = Bourgeoisie() >>> john.work() <<<
jack.greeting = 'Maam'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.' + self.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()<class Worker>
greeting: 'Sir'

>>> jack<class Bourgeoisie>
greeting: 'Peon'

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)
Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

```python
>>> Worker().work()
<class Worker>
greeting: 'Sir'

>>> jack
<class Bourgeoisie>
greeting: 'Peon'

>>> jack.work()

>>> john.work()

jack <Worker>

ej:

>>> john.elf.work(john)
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work() <class Worker>
  greeting: 'Sir'

>>> jack <class Bourgeoisie>
  greeting: 'Peon'

>>> jack.work()

>>> john.work() <class Bourgeoisie>

>>> john.elf.work(john) <class Bourgeoisie>
  elf: 

jack <Worker>
  elf: 

john <Bourgeoisie>
  elf: 

David Kahn
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
greeting = 'Sir'
def __init__(self):
    self.elf = Worker
def work(self):
    return self.greeting + ', I work'
def __repr__(self):
    return 'Bourgeoisie.greeting'

class Bourgeoisie(Worker):
greeting = 'Peon'
def work(self):
    print(Worker.work(self))
    return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
<class Worker>
greeting: 'Sir'

>>> jack
<class Bourgeoisie>
greeting: 'Peon'

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)

<Worker>
elf: 
greeting: 'Maam'

<Worker>
elf: 
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()  <class Worker>
    greeting: 'Sir'

>>> jack
<class Bourgeoisie>
    greeting: 'Peon'

>>> jack.work()  jack <Worker>

>>> jack.elf.work(john)

>>> john.work()  john <Bourgeoisie>

>>> john.elf.work(john)

jack <Worker>
elf: 
greeting: 'Maam'

john <Bourgeoisie>
elf: 


Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
<class Worker>
greeting: 'Sir'

>>> jack.work()
<class Bourgeoisie>
greeting: 'Peon'

>>> john.work()
'class Worker>

greeting: 'Maam'

>>> john.elf.work(john)

jack <Worker>
elf: 

greeting: 'Maam'

john <Bourgeoisie>
elf: 

Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.' + Worker.__repr__(self)

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

```
>>> Worker().work()
'Sir, I work'

>>> jack
<class Worker>
greeting: 'Sir'

>>> jack.work()
<class Bourgeoisie>
greeting: 'Peon'

>>> john.work()

>>> john.elf.work(john)
```

Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting
class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)

<class Worker>
greeting: 'Sir'

greeting: 'Peon'

<class Bourgeoisie>
greeting: 'Maam'

jack <Worker>  
elf:  
greeting: 'Maam'

john <Bourgeoisie>
elf:  

Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.greeting'

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

```python
>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)
```

<class Worker>

greeting: 'Sir'

<class Bourgeoisie>

greeting: 'Peon'

jack <Worker>

elf: 

greeting: 'Maam'

john <Bourgeoisie>

elf: 
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return self.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

```python
>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()

>>> john.elf.work(john)
```

```
class Worker:
greeting = 'Sir'
def __init__(self):
    self.elf = Worker
def work(self):
    return self.greeting + ', I work'
def __repr__(self):
    return Bourgeoisie.greeting

class Bourgeoisie(Worker):
greeting = 'Peon'
def work(self):
    print(Worker.work(self))
    return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Worker{}
greeting: ' + self.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
njohn = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()

>>> john.elf.work(john)

<class Worker>
greeting: 'Sir'

equal: <class Bourgeoisie>
greeting: 'Peon'
jack <Worker>

equal: <Bourgeoisie>
egreeting: 'Maam'

john <Bourgeoisie>

elf: 

elf: 

Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
greeting = 'Sir'
def __init__(self):
    self.elf = Worker
def work(self):
    return self.greeting + ', I work'
def __repr__(self):
    return Bourgeoisie.greeting

class Bourgeoisie(Worker):
greeting = 'Peon'
def work(self):
    print(Worker.work(self))
    return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
'Peon, I work
'I gather wealth'

>>> john.elf.work(john)

jack <Worker>
greeting: 'Sir'

<class Worker>
greeting: 'Sir'

jack <Worker>
egreeting: 'Peon'

<class Bourgeoisie>
egreeting: 'Peon'

john <Bourgeoisie>
egreeting: 'Maam'

elf: john <Bourgeoisie>
ell: john <Bourgeoisie>
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.' + self.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
'Peon, I work
'I gather wealth'

>>> john.elf.work(john)

<class Worker>
greeting: 'Sir'

<class Bourgeoisie>
greeting: 'Peon'

jack <Worker>
elf: 
greeting: 'Maam'

john <Bourgeoisie>
elf: 

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
'Peon, I work'

>>> john.elf.work(john)
'Peon, I work'
```
Trees
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

A: ● ■
B: ● ● ● ●
C: ● ● ■ ■
D: ● ● ●
E: ●
...

Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

*Problem:* Implement `morse` so that `decode` works correctly

A: 🔄 🔄
B: 🔄 🔄 🔄 🔄
C: 🔄 🔄 🔄 🔄
D: 🔄 🔄 🔄
E: 🔄
...


Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

Problem: Implement morse so that decode works correctly

abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

Problem: Implement `morse` so that `decode` works correctly.

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.}

def decode(signals, tree):
    '''Decode signals into a letter.

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '-..', '...']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    '''

    for signal in signals:
        tree = [b for b in tree.branches if b.entry == signal][0]
    leaves = [b for b in tree.branches if not b.branches]
    assert len(leaves) == 1
    return leaves[0].entry
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

Problem: Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter."

    t = morse(abcde)
    [decode(s, t) for s in ['-..', '.', '-.-.', '.', '-..', '.']
    ['d', 'e', 'c', 'a', 'd', 'e']
    """

    for signal in signals:
        tree = [b for b in tree.branches if b.entry == signal][0]
    leaves = [b for b in tree.branches if not b.branches]
    assert len(leaves) == 1
    return leaves[0].entry
```

Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '---', 'c': '---', 'd': '--', 'e': '.
```

def decode(signals, tree):
    """Decode signals into a letter.""
    for signal in signals:
        tree = [b for b in tree.branches if b.entry == signal][0]
    leaves = [b for b in tree.branches if not b.branches]
    assert len(leaves) == 1
    return leaves[0].entry
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter.\n    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['--', '.', '-.-', '.', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']\n    ""
    for signal in signals:
        tree = [b for b in tree.branches if b.entry == signal][0]
    leaves = [b for b in tree.branches if not b.branches]
    assert len(leaves) == 1
    return leaves[0].entry
```

```python
def morse(code):
    ....
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

Problem: Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '---', 'c': '---', 'd': '---', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter."
    for signal in signals:
        tree = [b for b in tree.branches if b.entry == signal][0]
        leaves = [b for b in tree.branches if not b.branches]
        assert len(leaves) == 1
        return leaves[0].entry
```

```python
>>> t = morse(abcde)
>>> [decode(s, t) for s in ['---', '---', '---', '---', '.']]
['d', 'e', 'c', 'a', 'd', 'e']
```

```python
def morse(code):
    ....
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

**Problem:** Implement `morse` so that `decode` works correctly.

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter."

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '-.', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']

    for signal in signals:
        tree = [b for b in tree.branches if b.entry == signal][0]
    leaves = [b for b in tree.branches if not b.branches]
    assert len(leaves) == 1
    return leaves[0].entry
```

An empty list is a false value.
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

**Problem:** Implement `morse` so that `decode` works correctly.

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter."
    for signal in signals:
        tree = [b for b in tree.branches if b.entry == signal][0]
    leaves = [b for b in tree.branches if not b.branches]
    assert len(leaves) == 1
    return leaves[0].entry
```

```python
def morse(code):
    ....
```

Not used

An empty list is a false value
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

Problem: Implement `morse` so that `decode` works correctly.

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': ' .'}

def decode(signals, tree):
    # Decode signals into a letter.
    t = morse(abcde)

    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '-..', '-..', '.']]  
    ['d', 'e', 'c', 'a', 'd', 'e']

    def morse(code):
        ....

    for signal in signals:
        tree = [b for b in tree.branches if b.entry == signal][0]

    leaves = [b for b in tree.branches if not b.branches]

    assert len(leaves) == 1
    return leaves[0].entry
```

An empty list is a false value.
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.}

def decode(signals, tree):
    """Decode signals into a letter."
    for signal in signals:
        tree = [b for b in tree.branches if b.entry == signal][0]
    leaves = [b for b in tree.branches if not b.branches]
    assert len(leaves) == 1
    return leaves[0].entry
```

```python
def morse(code):
    ...
```

An empty list is a false value
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

**Problem:** Implement `morse` so that `decode` works correctly.

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter.""

    tree = [b for b in tree.branches if b.entry == signal][0]
    leaves = [b for b in tree.branches if not b.branches]
    assert len(leaves) == 1
    return leaves[0].entry

def morse(code):
    ....

>>> t = morse(abcde)
```

An empty list is a false value.

Not used

A:  
B:  
C:  
D:  
E:  

Not used
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

Problem: Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '--.-', 'd': '-..', 'e': '.
```

```python
def decode(signals, tree):
    """Decode signals into a letter.""
    
    >>> t = Morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-', '---', '...', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']

    for signal in signals:
        tree = [b for b in tree.branches if b.entry == signal][0]
    leaves = [b for b in tree.branches if not b.branches]
    assert len(leaves) == 1
    return leaves[0].entry
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

**Problem:** Implement `morse` so that `decode` works correctly.

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-', 'd': '-..', 'e': '.'}
def decode(signals, tree):
    """Decode signals into a letter."
    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-', '[.-]', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    for signal in signals:
        tree = [b for b in tree.branches if b.entry == signal][0]
    leaves = [b for b in tree.branches if not b.branches]
    assert len(leaves) == 1
    return leaves[0].entry
```

```
A: ● ● ● ●
B: ● ● ● ● ●
C: ● ● ● ● ● ●
D: ● ● ● ● ● ● ●
E: ●
```

An empty list is a false value.
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.

```def decode(signals, tree):
    """Decode signals into a letter.""

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '[.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']

    for signal in signals:
        tree = [b for b in tree.branches if b.entry == signal][0]
    leaves = [b for b in tree.branches if not b.branches]
    assert len(leaves) == 1
    return leaves[0].entry
```

```
A: ● ● ● ● ●
B: ● ● ● ●
C: ● ● ● ● ●
D: ● ● ●
E: ●

Not used

A: ● ● ● ● ●
B: ● ● ● ●
C: ● ● ● ● ●
D: ● ● ●
E: ●

... Not used
```

An empty list is a false value