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)))
```

```
Global frame s

First Rest
1

First Rest
2

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

Note: The actual environment diagram is much more complicated.
Environment Diagrams
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)
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)
```

```mermaid
graph TD
    A[func oski(bear)] --> B[func abs(...)]
    B --> C[func cal(berk)]
    C --> D[Return Value]
    D --> E[Global frame]
    E --> F[func oski(bear)]
```

Return Value

Return Value

Return Value

Return Value

Return Value
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

func oski(bear) [parent=G]

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

func abs(...) [parent=G]

func cal(berk) [parent=f1]

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

Return Value

Return Value
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)
```
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)
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)
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`

### Local frames:
1. `f1: oski [parent=G]`
   - `bear`
   - `cal`
   - Return Value
2. `f2: cal [parent=f1]`
   - `berk 2`
   - Return Value
3. `func oski(bear) [parent=G]`
4. `func cal(berk) [parent=f1]`
5. `func λ(ley) [parent=f2]`
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)
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)
```

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!

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

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
```
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
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):
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

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

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

>>> jack

>>> jack.work()  >>> jack

>>> john

>>> john.work()  >>> john

>>> 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

>>> jack.work()

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

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

>>> jack.work()

>>> 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()
jack.greeting = 'Maam'

john = Bourgeoisie()

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

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

>>> jack.work()  # jack <Worker>
    elf:

>>> john.work()  # >>> john.elf.work(john)

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

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

>>> jack.work()

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

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

```python
jack <Worker>
elf: 

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()  
<class Worker>  
greeting: 'Sir'

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

>>> jack.work()  
jack <Worker>  
elf:  
greeting: 'Maam'

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

>>> 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()

jjack <Worker>

greeting: 'Peon'
jelf:

greeting: 'Maam'

>>> john.work()

john <Bourgeoisie>

greeting: 'Maam'

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

>>> john.elf.work(john)

jack <Worker>
egf: 
greeting: 'Maam'

john <Bourgeoisie>
egf: 

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()

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

greeting: 'Maam'
**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'

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

>>> jack
Peon

>>> jack.work()

>>> john.work()

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

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

greeting: 'Peon'

Jack <Worker>

greeting: 'Maam'

John <Bourgeoisie>
```
**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 f'Bourgeoisie.greeting: {self.greeting}'

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

class Bourgeoisie(Worker):
    greeting = 'Maam'
    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
<Worker>
greeting: 'Maam'
elf: john <Bourgeoisie>

greeting: 'Peon'
elf: john <Bourgeoisie>
```

```python
>>> jack.work()
Peon

greeting: 'Peon'
elf: john <Bourgeoisie>
```

```python
>>> john.work()
'I gather wealth'

greeting: 'Maam'
elf: john <Bourgeoisie>
```

```python
>>> john.elf.work(john)
'I gather wealth'

greeting: 'Maam'
elf: 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.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()  # Maam, I work

>>> john.work()  # I gather wealth

>>> john.elf.work(john)  # I gather wealth

>>> jack <Worker>

greeting: 'Peon'

>>> john <Bourgeoisie>

elf: 

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'

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()

>>> john.elf.work(john)

jack <Worker>
  elf: 
greeting: 'Maam'

john <Bourgeoisie>
  elf: 
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

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: john <Bourgeoisie>
greeting: 'Maam'

john <Bourgeoisie>
elf: 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.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: john <Bourgeoisie>
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

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'

<class Worker>
greeting: 'Sir'

ejack <Worker>
egreeting: 'Peon'

<class Bourgeoisie>
greeting: 'Peon'
jack <Worker>
e: 
greeting: 'Maam'

john <Bourgeoisie>
e: 

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
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': '.'}
```

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.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```
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.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```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.\n    >>> 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.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```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.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    if len(leaves) == 1:
        return leaves[0].label
```

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

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

```
decode('.', t)
```
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.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

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

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

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

```python
decode('.', t)
```

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."
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label

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

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."
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
        assert len(leaves) == 1
        return leaves[0].label
```

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

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

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.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]  
    assert len(leaves) == 1
    return leaves[0].label

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.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]  
    assert len(leaves) == 1
    return leaves[0].label
```

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

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

```python
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."
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label

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

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

A:  
B:  
C:  
D:  
E:  

`decode('.', t)`

```python
def decode(signals, tree):
    """Decode signals into a letter."
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```
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.''

    def morse(code):
        ....

    t = morse(abcde)
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

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.""
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
        assert len(leaves) == 1
        return leaves[0].label
```

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

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

A: 
B: 
C: 
D: 
E: 

[tree diagram]

\['e\']
\['-\']

[tree diagram]
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.label == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```
>>> t = morse(abcde)
>>> [decode(s, t) for s in ['-..', '.', '-.-.', '(-)', '-..', '.']]
['d', 'e', 'c', 'a', '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.""
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```python
morse = {'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
def decode(signals, tree):
    """Decode signals into a letter."
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

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

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

(Demo)