61A Lecture 22
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

>>> 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)))
```

![Diagram of recursive list structure]

*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)))
```

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
```

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
>>> s.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
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)

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

```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)
odsi(abs)
```

```
Global frame
oski

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

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

- `func oski(bear)[parent=G]`
- `func λ(ley)[parent=f2]`
- `func cal(berk)[parent=f1]`

---

```
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)
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)
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)
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)
```
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)
    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
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'
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
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'
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'
```
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()  
>>> 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

>>> 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'

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

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

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

>>> john.work()

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

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

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

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

greeting: 'Peon'

>>> john.work()

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

greeting: 'Maam'

>>> 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'

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

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

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)

jack <Worker>
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()
I gather wealth

>>> john.work()

>>> john.elf.work(john)

jack <Worker>
greeting: 'Peon'

john <Bourgeoisie>
greeting: 'Maam'

elf: 
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'

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

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

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

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

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

>>> jack
Peon

>>> jack.work()

>>> john.work()

>>> 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()
'Sir, I work'

>>> jack
Peon

>>> jack.work()

>>> john.work()

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

- `Worker`
- `Bourgeoisie`
- `jack <Worker>`
- `john <Bourgeoisie>`
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

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)

<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
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()

>>> john.elf.work(john)

<class Worker>
greeting: 'Sir'

<class Bourgeoisie>
greeting: 'Peon'

jack <Worker>
elf: <Worker>
greeting: 'Maam'

john <Bourgeoisie>
elf: <Bourgeoisie>
```

Land Owners

Instance attributes are found before class attributes; class attributes are inherited
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
'I gather wealth'

>>> john.elf.work(john)

<class Worker>
greeting: 'Sir'

<class Bourgeoisie>
greeting: 'Peon'

ejack <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
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: 
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 f'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'

<class Bourgeoisie>:
    greeting: 'Peon'

jack <Worker>:
    elf: 
    greeting: 'Maam'

john <Bourgeoisie>:
    elf: 

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>●●</td>
</tr>
<tr>
<td>B:</td>
<td>●●●●</td>
</tr>
<tr>
<td>C:</td>
<td>●●●● ●</td>
</tr>
<tr>
<td>D:</td>
<td>●●● ●</td>
</tr>
<tr>
<td>E:</td>
<td>●</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
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': '.}
```
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.root == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].root
```
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.root == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].root
```

```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.

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

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

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

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

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

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

```python
morse('.')
```

A: ⬣ ⬣ ⬣ ⬣
B: ⬣ ⬣ ⬣ ⬣ ⬣ ⬣
C: ⬣ ⬣ ⬣ ⬣ ⬣ ⬣
D: ⬣ ⬣ ⬣ ⬣ ⬣ ⬣
E: ⬣

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

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

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

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

```python
def Morse(code):
    ....
    decode('.', t)
```

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

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

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

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

A: • • •
B: • • • •
C: • • • • •
D: • • • •
E: •

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

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

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

```python
code = {'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.root == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
        assert len(leaves) == 1
        return leaves[0].root
    return

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

---

def morse(code):
    ....

```

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

... ?

'e'
```

---

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.

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

```python
A: 🔑
B: 🔒
C: 🔒
D: 🔒
E: 🔒

\[
\begin{array}{l}
\cdots \\
\text{e} \\
\text{‘} \\
\end{array}
\]
```
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.root == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
        assert len(leaves) == 1
        return leaves[0].root
```

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

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

... ?

'e'

'a'

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

```
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

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

(Demo)