Announcements

☐ HW6 due tomorrow

☐ Ants project out
Mutable Recursive Lists

```python
def mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        return getitem_rlist(contents, value)
    elif message == 'push':
        contents = make_rlist(value, contents)
    elif message == 'pop':
        item = first(contents)
        contents = rest(contents)
        return item
    elif message == 'str':
        return str_rlist(contents)
    return dispatch
```
Building Dictionaries with Lists

Now that we have lists, we can use them to build dictionaries.
Building Dictionaries with Lists

Now that we have lists, we can use them to build dictionaries.

We store key-value pairs as 2-element lists inside another list.
Now that we have lists, we can use them to build dictionaries

We store key-value pairs as 2-element lists inside another list

```python
records = [['cain', 2.79], ['bumgarner', 3.37], ['vogelsong', 3.37], ['lincecum', 5.18], ['zito', 4.15]]
```
Now that we have lists, we can use them to build dictionaries.

We store key-value pairs as 2-element lists inside another list:

```python
records = [['cain', 2.79], ['bumgarner', 3.37], ['vogelsong', 3.37], ['lincecum', 5.18], ['zito', 4.15]]
```

Dictionary operations:
Now that we have lists, we can use them to build dictionaries

We store key-value pairs as 2-element lists inside another list

```python
records = [['cain', 2.79], ['bumgarner', 3.37], ['vogelsong', 3.37], ['lincecum', 5.18], ['zito', 4.15]]
```

Dictionary operations:

- `getitem(key)`: Look at each record until we find a stored key that matches `key`
Building Dictionaries with Lists

Now that we have lists, we can use them to build dictionaries.

We store key-value pairs as 2-element lists inside another list.

```python
records = [['cain', 2.79],
           ['bumgarner', 3.37],
           ['vogelsong', 3.37],
           ['lincecum', 5.18],
           ['zito', 4.15]]
```

Dictionary operations:

• **getitem(key)**: Look at each record until we find a stored key that matches `key`.

• **setitem(key, value)**: Check if there is a record with the given key. If so, change the stored value to `value`. If not, add a new record that stores `key` and `value`. 
Implementing Dictionaries
def dictionary():

Implementing Dictionaries

def dictionary():
    """Return a functional implementation of a dictionary."""
Implementing Dictionaries

def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        pass

def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        pass
def dictionary():
    r"""Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
def dictionary():
    """Return a functional implementation of a dictionary."
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])
    def dispatch(message, key=None, value=None):
Implementing Dictionaries

```python
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])
    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            pass
```

def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])
    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])
    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])
    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []

def getitem(key):
    for k, v in records:
        if k == key:
            return v

def setitem(key, value):
    for item in records:
        if item[0] == key:
            item[1] = value
            return
    records.append([key, value])

def dispatch(message, key=None, value=None):
    if message == 'getitem':
      return getitem(key)
    elif message == 'setitem':
      setitem(key, value)
    elif message == 'keys':
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])
    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
Implementing Dictionaries

def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
def getitem(key):
        for k, v in records:
            if k == key:
                return v

def setitem(key, value):
    for item in records:
        if item[0] == key:
            item[1] = value
            return
    records.append([key, value])
def dispatch(message, key=None, value=None):
    if message == 'getitem':
        return getitem(key)
    elif message == 'setitem':
        setitem(key, value)
    elif message == 'keys':
        return tuple(k for k, _ in records)
    elif message == 'values':
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])
    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
        elif message == 'values':
            return tuple(v for _, v in records)
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])
    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
        elif message == 'values':
            return tuple(v for _, v in records)
        return dispatch

return dispatch
def dictionary():
    """Return a functional implementation of a dictionary."""
    records = []
    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])
    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
        elif message == 'values':
            return tuple(v for _, v in records)
        return dispatch

Question: Do we need a nonlocal statement here?
Dispatch Dictionaries
Enumerating different messages in a conditional statement isn't very convenient:
Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:

• Equality tests are repetitive
Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:

• Equality tests are repetitive

• We can't add new messages without writing new code
Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:

- Equality tests are repetitive
- We can't add new messages without writing new code

A dispatch dictionary has messages as keys and functions (or data objects) as values.
Enumerating different messages in a conditional statement isn't very convenient:

• Equality tests are repetitive
• We can't add new messages without writing new code

A dispatch dictionary has messages as keys and functions (or data objects) as values.

Dictionaries handle the message look-up logic; we concentrate on implementing useful behavior.
An Account as a Dispatch Dictionary
def account(balance):
def account(balance):
    """Return an account that is represented as a dispatch dictionary."""
def account(balance):
    """Return an account that is represented as a dispatch dictionary."""

def withdraw(amount):

def account(balance):
    """Return an account that is represented as a dispatch dictionary."""

def withdraw(amount):
    if amount > dispatch['balance']:
def account(balance):
    """Return an account that is represented as a
    dispatch dictionary."""

    def withdraw(amount):
        if amount > dispatch['balance']:
            return 'Insufficient funds'
def account(balance):
    """Return an account that is represented as a dispatch dictionary."""

def withdraw(amount):
    if amount > dispatch['balance']:
        return 'Insufficient funds'
    dispatch['balance'] -= amount
An Account as a Dispatch Dictionary

```python
def account(balance):
    """Return an account that is represented as a dispatch dictionary."""

    def withdraw(amount):
        if amount > dispatch['balance']:
            return 'Insufficient funds'
        dispatch['balance'] -= amount
        return dispatch['balance']
```
def account(balance):
    """Return an account that is represented as a dispatch dictionary."""

def withdraw(amount):
    if amount > dispatch['balance']:
        return 'Insufficient funds'
    dispatch['balance'] -= amount
    return dispatch['balance']

def deposit(amount):
An Account as a Dispatch Dictionary

def account(balance):
    """Return an account that is represented as a dispatch dictionary."""
    
def withdraw(amount):
        if amount > dispatch['balance']:
            return 'Insufficient funds'
        dispatch['balance'] -= amount
        return dispatch['balance']
    
def deposit(amount):
        dispatch['balance'] += amount
def account(balance):
    """Return an account that is represented as a dispatch dictionary."""

def withdraw(amount):
    if amount > dispatch['balance']:
        return 'Insufficient funds'
    dispatch['balance'] -= amount
    return dispatch['balance']

def deposit(amount):
    dispatch['balance'] += amount
    return dispatch['balance']
def account(balance):
    """Return an account that is represented as a dispatch dictionary."""

def withdraw(amount):
    if amount > dispatch['balance']:
        return 'Insufficient funds'
    dispatch['balance'] -= amount
    return dispatch['balance']

def deposit(amount):
    dispatch['balance'] += amount
    return dispatch['balance']

dispatch = {'balance': balance, 'withdraw': withdraw, 'deposit': deposit}
An Account as a Dispatch Dictionary

def account(balance):
    """Return an account that is represented as a dispatch dictionary."""

def withdraw(amount):
    if amount > dispatch['balance']:
        return 'Insufficient funds'
    dispatch['balance'] -= amount
    return dispatch['balance']

def deposit(amount):
    dispatch['balance'] += amount
    return dispatch['balance']

display = {'balance': balance, 'withdraw': withdraw, 'deposit': deposit}

return display
def account(balance):
    
    """Return an account that is represented as a dispatch dictionary."""

    def withdraw(amount):
        if amount > dispatch['balance']:
            return 'Insufficient funds'
        dispatch['balance'] -= amount
        return dispatch['balance']

    def deposit(amount):
        dispatch['balance'] += amount
        return dispatch['balance']

    dispatch = {'balance': balance, 'withdraw': withdraw, 'deposit': deposit}

    return dispatch
The Story So Far About Data
The Story So Far About Data

**Data abstraction**: Enforce a separation between how data values are represented and how they are used.
The Story So Far About Data

**Data abstraction**: Enforce a separation between how data values are represented and how they are used.

**Abstract data types**: A representation of a data type is valid if it satisfies certain behavior conditions.
The Story So Far About Data

**Data abstraction**: Enforce a separation between how data values are represented and how they are used.

**Abstract data types**: A representation of a data type is valid if it satisfies certain behavior conditions.

**Message passing**: We can organize large programs by building components that relate to each other by passing messages.
The Story So Far About Data

**Data abstraction**: Enforce a separation between how data values are represented and how they are used.

**Abstract data types**: A representation of a data type is valid if it satisfies certain behavior conditions.

**Message passing**: We can organize large programs by building components that relate to each other by passing messages.

**Dispatch functions/dictionaries**: A single object can include many different (but related) behaviors that all manipulate the same local state.
The Story So Far About Data

**Data abstraction**: Enforce a separation between how data values are represented and how they are used.

**Abstract data types**: A representation of a data type is valid if it satisfies certain behavior conditions.

**Message passing**: We can organize large programs by building components that relate to each other by passing messages.

**Dispatch functions/dictionaries**: A single object can include many different (but related) behaviors that all manipulate the same local state.

(All of these techniques can be implemented using only functions and assignment.)
Object-Oriented Programming
Object-Oriented Programming

A method for organizing modular programs
Object-Oriented Programming

A method for organizing modular programs

• Abstraction barriers
Object-Oriented Programming

A method for organizing modular programs

• Abstraction barriers
• Message passing
Object-Oriented Programming

A method for organizing modular programs

• Abstraction barriers
• Message passing
• Bundling together information and related behavior
Object-Oriented Programming

A method for organizing modular programs

• Abstraction barriers
• Message passing
• Bundling together information and related behavior

A metaphor for computation using distributed state
Object-Oriented Programming

A method for organizing modular programs

- Abstraction barriers
- Message passing
- Bundling together information and related behavior

A metaphor for computation using distributed state

- Each *object* has its own local state.
Object-Oriented Programming

A method for organizing modular programs
- Abstraction barriers
- Message passing
- Bundling together information and related behavior

A metaphor for computation using distributed state
- Each *object* has its own local state.
- Each object also knows how to manage its own local state, based on the messages it receives.
Object-Oriented Programming

A method for organizing modular programs

• Abstraction barriers
• Message passing
• Bundling together information and related behavior

A metaphor for computation using distributed state

• Each *object* has its own local state.
• Each object also knows how to manage its own local state, based on the messages it receives.
• Several objects may all be instances of a common type.
Object-Oriented Programming

A method for organizing modular programs

• Abstraction barriers
• Message passing
• Bundling together information and related behavior

A metaphor for computation using distributed state

• Each *object* has its own local state.
• Each object also knows how to manage its own local state, based on the messages it receives.
• Several objects may all be instances of a common type.
• Different types may relate to each other as well.
Object-Oriented Programming

A method for organizing modular programs

- Abstraction barriers
- Message passing
- Bundling together information and related behavior

A metaphor for computation using distributed state

- Each *object* has its own local state.
- Each object also knows how to manage its own local state, based on the messages it receives.
- Several objects may all be instances of a common type.
- Different types may relate to each other as well.

Specialized syntax & vocabulary to support this metaphor
A class serves as a template for its instances.
A *class* serves as a template for its *instances*.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.
A class serves as a template for its instances.

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
```
A class serves as a template for its instances.

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
```
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
```
A class serves as a template for its instances.

**Idea**: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
```
A *class* serves as a template for its *instances*.

**Idea**: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```
A class serves as a template for its instances.

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

Idea: All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

**Idea:** All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.

```python
>>> a.deposit(15)
```
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
>>> a.deposit(15)
15
```

**Idea:** All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

**Idea:** All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.

```
>>> a.deposit(15)
15
>>> a.withdraw(10)
```
Classes

A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

**Idea:** All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.

```python
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
```
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

**Idea:** All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.

```
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
```
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

**Idea:** All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.

```python
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
```
Classes

A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

**Idea:** All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.

```python
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
>>> a.withdraw(10)
```
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
>>> a.withdraw(10)
'Insufficient funds'
```

**Idea:** All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.
Classes

A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
```

**Better idea:** All bank accounts share a "withdraw" method.

```python
>>> a.withdraw(10)
'Insufficient funds'
```
The Class Statement
The Class Statement

class <name>(<base class>):
    <suite>
The Class Statement

class <name>(<base class>):
    <suite>

Next lecture
A class statement creates a new class and binds that class to `<name>` in the first frame of the current environment.
A class statement **creates** a new class and **binds** that class to `<name>` in the first frame of the current environment.

Statements in the `<suite>` create attributes of the class.
A class statement **creates** a new class and **binds** that class to `<name>` in the first frame of the current environment.

Statements in the `<suite>` create attributes of the class.

As soon as an instance is created, it is passed to `__init__`, which is an attribute of the class.
A class statement **creates** a new class and **binds** that class to `<name>` in the first frame of the current environment.

Statements in the `<suite>` create attributes of the class.

As soon as an instance is created, it is passed to **`__init__`**, which is an attribute of the class.

```python
class Account(object):
```
A class statement \texttt{creates} a new class and \texttt{binds} that class to \texttt{name} in the first frame of the current environment.

Statements in the \texttt{suite} create attributes of the class.

As soon as an instance is created, it is passed to \texttt{__init__}, which is an attribute of the class.

```python
class Account(object):
    def __init__(self, account_holder):
```

\texttt{Next lecture}
A class statement **creates** a new class and **binds** that class to `<name>` in the first frame of the current environment.

Statements in the `<suite>` create attributes of the class.

As soon as an instance is created, it is passed to `__init__`, which is an attribute of the class.

```python
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
```
The Class Statement

A class statement **creates** a new class and **binds** that class to `<name>` in the first frame of the current environment.

Statements in the `<suite>` create attributes of the class.

As soon as an instance is created, it is passed to `__init__`, which is an attribute of the class.

```python
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```
Initialization
**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```
Initialization

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

When a class is called:
**Initialization**

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created:
**Initialization**

**Idea**: All bank accounts have a balance and an account holder; the Account class should add those attributes.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created:

2. The constructor `__init__` of the class is called with the new object as its first argument (called `self`), along with additional arguments provided in the call expression.
Initialization

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created:
2. The constructor `__init__` of the class is called with the new object as its first argument (called `self`), along with additional arguments provided in the call expression.
Initialization

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created:
2. The constructor `__init__` of the class is called with the new object as its first argument (called `self`), along with additional arguments provided in the call expression.

```python
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```
**Initialization**

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created:
2. The constructor `__init__` of the class is called with the new object as its first argument (called `self`), along with additional arguments provided in the call expression.

```python
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```
Initialization

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created:

2. The constructor `__init__` of the class is called with the new object as its first argument (called `self`), along with additional arguments provided in the call expression.

```python
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```
Object Identity
Object Identity

Every object that is an instance of a user-defined class has a unique identity:
Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
```
Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
```

Identity testing is performed by "is" and "is not" operators:
Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
```

Identity testing is performed by "is" and "is not" operators:

```python
>>> a is a
True
>>> a is not b
True
```
Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
```

Identity testing is performed by "is" and "is not" operators:

```python
>>> a is a
True
>>> a is not b
True
```

Binding an object to a new name using assignment does not create a new object:
Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
```

Identity testing is performed by "is" and "is not" operators:

```python
>>> a is a
True
>>> a is not b
True
```

Binding an object to a new name using assignment does not create a new object:

```python
>>> c = a
>>> c is a
True
```
Methods
Methods

Methods are defined in the suite of a class statement
Methods

Methods are defined in the suite of a class statement

```python
class Account(object):
```
Methods are defined in the suite of a class statement

```python
class Account(object):
    def __init__(self, account_holder):
```
Methods

Methods are defined in the suite of a class statement

```python
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
```
Methods

Methods are defined in the suite of a class statement

class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
Methods

Methods are defined in the suite of a class statement

class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
Methods

Methods are defined in the suite of a class statement

class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
Methods

Methods are defined in the suite of a class statement

```python
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```
Methods

Methods are defined in the suite of a class statement

```python
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
```

Methods are defined in the suite of a class statement

class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
Methods

Methods are defined in the suite of a class statement

class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
Methods are defined in the suite of a class statement

class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
Methods

Methods are defined in the suite of a class statement

class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
Methods

Methods are defined in the suite of a class statement

```python
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

These def statements create function objects as always, but their names are bound as attributes of the class.
Invoking Methods
Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state.
Invoking Methods

All invoked methods have access to the object via the `self` parameter, and so they can all access and manipulate the object's state.

```python
class Account(object):
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
    return self.balance
```
Invoking Methods

All invoked methods have access to the object via the `self` parameter, and so they can all access and manipulate the object's state.

```python
class Account(object):
    
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Called with two arguments
All invoked methods have access to the object via the `self` parameter, and so they can all access and manipulate the object's state.

```python
class Account(object):
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Dot notation automatically supplies the first argument to a method.
Invoking Methods

All invoked methods have access to the object via the `self` parameter, and so they can all access and manipulate the object's state.

```python
class Account(object):
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
```

Dot notation automatically supplies the first argument to a method.
All invoked methods have access to the object via the `self` parameter, and so they can all access and manipulate the object's state.

```python
class Account(object):
    ...def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Dot notation automatically supplies the first argument to a method.

```python
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
```
Dot Expressions
Objects receive messages via dot notation
Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

<expression> . <name>
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

\(<\text{expression}> \ . \ <\text{name}>\)

The \(<\text{expression}>\) can be any valid Python expression
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

\(<\text{expression}\> \ . \ <\text{name}\>

The \(<\text{expression}\>\) can be any valid Python expression

The \(<\text{name}\>\) must be a simple name
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

\(<\text{expression}\> \ . \ <\text{name}\>\)

The \(<\text{expression}\>\) can be any valid Python expression

The \(<\text{name}\>\) must be a simple name

Evaluates to the value of the attribute looked up by \(<\text{name}\>\) in the object that is the value of the \(<\text{expression}\>\)
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

\texttt{<expression> . <name>}

The \texttt{<expression>} can be any valid Python expression

The \texttt{<name>} must be a simple name

Evaluates to the value of the attribute \texttt{looked up} by \texttt{<name>}
in the object that is the value of the \texttt{<expression>}

\texttt{tom_account.deposit(10)}
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

\(<expression> . <name>\)

The \(<expression>\) can be any valid Python expression

The \(<name>\) must be a simple name

Evaluates to the value of the attribute \texttt{looked up} by \(<name>\)
in the object that is the value of the \(<expression>\)

\(\texttt{tom\_account.deposit(10)}\)

Dot expression
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

\[
<\text{expression}> \ . \ <\text{name}>
\]

The \texttt{<expression>} can be any valid Python expression

The \texttt{<name>} must be a simple name

Evaluates to the value of the attribute \texttt{looked up} by \texttt{<name>} in the object that is the value of the \texttt{<expression>}

\begin{minipage}{0.4\textwidth}
\begin{verbatim}
tom_account.deposit(10)
\end{verbatim}
\end{minipage} \hspace{0.05\textwidth} \begin{minipage}{0.3\textwidth}
\begin{itemize}
\item \textbf{Dot expression}
\item \textbf{Call expression}
\end{itemize}
\end{minipage}
Accessing Attributes
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary.
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```python
>>> getattr(tom_account, 'balance')
```
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```python
>>> getattr(tom_account, 'balance')
10
```
Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary.

```python
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
```
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```python
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```python
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```

`getattr` and dot expressions look up a name in the same way
Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary:

```python
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```

`getattr` and dot expressions look up a name in the same way.

Looking up an attribute name in an object may return:
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```python
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```

`getattr` and dot expressions look up a name in the same way

Looking up an attribute name in an object may return:

- One of its instance attributes, or
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```python
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```

`getattr` and dot expressions look up a name in the same way

Looking up an attribute name in an object may return:

- One of its instance attributes, or
- One of the attributes of its class
Methods and Functions
Methods and Functions

Python distinguishes between:
Methods and Functions

Python distinguishes between:

• *Functions*, which we have been creating since the beginning of the course, and
Methods and Functions

Python distinguishes between:

- *Functions*, which we have been creating since the beginning of the course, and

- *Bound methods*, which couple together a function and the object on which that method will be invoked.
Methods and Functions

Python distinguishes between:

• *Functions*, which we have been creating since the beginning of the course, and

• *Bound methods*, which couple together a function and the object on which that method will be invoked.

\[
\text{Object } + \text{ Function } = \text{ Bound Method}
\]
Methods and Functions

Python distinguishes between:

• *Functions*, which we have been creating since the beginning of the course, and

• *Bound methods*, which couple together a function and the object on which that method will be invoked.

\[
\text{Object + Function} = \text{Bound Method}
\]

```python
>>> type(Account.deposit)
```
Methods and Functions

Python distinguishes between:

• *Functions*, which we have been creating since the beginning of the course, and

• *Bound methods*, which couple together a function and the object on which that method will be invoked.

\[
\text{Object + Function} = \text{Bound Method}
\]

```python
>>> type(Account.deposit)
<class 'function'>
```
Python distinguishes between:

- **Functions**, which we have been creating since the beginning of the course, and
- **Bound methods**, which couple together a function and the object on which that method will be invoked.

\[
\text{Object + Function = Bound Method}
\]

```python
type(Account.deposit)
<class 'function'>
type(tom_account.deposit)
```
Methods and Functions

Python distinguishes between:

- *Functions*, which we have been creating since the beginning of the course, and
- *Bound methods*, which couple together a function and the object on which that method will be invoked.

\[ \text{Object} + \text{Function} = \text{Bound Method} \]

```python
>>> type(Account.deposit)
<class 'function'>
>>> type(tom_account.deposit)
<class 'method'>
```
Python distinguishes between:

- **Functions**, which we have been creating since the beginning of the course, and
- **Bound methods**, which couple together a function and the object on which that method will be invoked.

\[
\text{Object} + \text{ Function} = \text{Bound Method}
\]

```python
>>> type(Account.deposit)
<class 'function'>
>>> type(tom_account.deposit)
<class 'method'>
```

```python
>>> Account.deposit(tom_account, 1001)
```
Python distinguishes between:

- **Functions**, which we have been creating since the beginning of the course, and
- **Bound methods**, which couple together a function and the object on which that method will be invoked.

\[
\text{Object} + \text{Function} = \text{Bound Method}
\]

```python
>>> type(Account.deposit)
<class 'function'>
>>> type(tom_account.deposit)
<class 'method'>
```

```python
>>> Account.deposit(tom_account, 1001)
1011
```
Methods and Functions

Python distinguishes between:

• *Functions*, which we have been creating since the beginning of the course, and

• *Bound methods*, which couple together a function and the object on which that method will be invoked.

    Object + Function = Bound Method

    >>> type(Account.deposit)
    <class 'function'>
    >>> type(tom_account.deposit)
    <class 'method'>

    >>> Account.deposit(tom_account, 1001)
    1011
    >>> tom_account.deposit(1000)
Methods and Functions

Python distinguishes between:

- **Functions**, which we have been creating since the beginning of the course, and
- **Bound methods**, which couple together a function and the object on which that method will be invoked.

\[
\text{Object} \ + \ \text{Function} \ = \ \text{Bound Method}
\]

```python
>>> type(Account.deposit)
<class 'function'>
>>> type(tom_account.deposit)
<class 'method'>

>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1000)
2011
```
Methods and Currying
Methods and Currying

Earlier, we saw currying, which converts a function that takes in multiple arguments into multiple chained functions.
Methods and Currying

Earlier, we saw *currying*, which converts a function that takes in multiple arguments into multiple chained functions.

```python
def curry(f):
    def outer(x):
        def inner(*args):
            return f(x, *args)
        return inner
    return outer
```
Methods and Currying

Earlier, we saw currying, which converts a function that takes in multiple arguments into multiple chained functions.

```python
def curry(f):
    def outer(x):
        def inner(*args):
            return f(x, *args)
        return inner
    return outer

>>> add2 = curry(add)(2)
>>> add2(3)
5
```
Methods and Currying

Earlier, we saw currying, which converts a function that takes in multiple arguments into multiple chained functions.

The same procedure can be used to create a bound method from a function

```python
def curry(f):
    def outer(x):
        def inner(*args):
            return f(x, *args)
        return inner
    return outer

>>> add2 = curry(add)(2)
>>> add2(3)
5
```
Methods and Currying

Earlier, we saw currying, which converts a function that takes in multiple arguments into multiple chained functions.

The same procedure can be used to create a bound method from a function

```python
def curry(f):
    def outer(x):
        def inner(*args):
            return f(x, *args)
        return inner
    return outer
```

```python
>>> add2 = curry(add)(2)
>>> add2(3)
5
```

```python
>>> tom_deposit = curry(Account.deposit)(tom_account)
>>> tom_deposit(1000)
3011
```