Attributes

Announcements

Method Calls

Dot Expressions

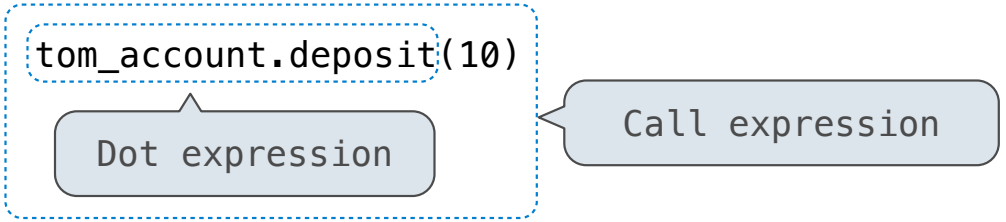
Methods are invoked using dot notation

<expression> . <name>

The <expression> can be any valid Python expression

The <name> is just a name (not a complex expression)

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



(Demo)

Attribute Lookup

Looking Up Attributes by Name

Both instances and classes have attributes that can be looked up by dot expressions

<expression> . <name>

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression

2. <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned

3. If not, <name> is looked up in the class, which yields a class attribute value

4. That value is returned unless it is a function, in which case a bound method is returned instead

Discussion Question: Where's Waldo?

Write an expression **with no quotes or +** that evaluates to 'Waldo'

```
class Town:
   def __init__(self, w, aldo):
        if aldo == 7:
            self.street = {self.f(w): 'Waldo'}

    def f(self, x):
        return x + 1
```

Discussion Question: Where's Waldo?

Write an expression **with no quotes or +** that evaluates to 'Waldo'

```
class Town:
    def __init__(self, w, aldo):
        if aldo == 7:
            self.street = {self.f(w): 'Waldo'}

    def f(self, x):
        return x + 1
```

```
>>> Town(1, 7).street[2]
'Waldo'
```

Discussion Question: Where's Waldo?

Write an expression **with no quotes or +** that evaluates to 'Waldo'

```
class Beach:
    def __init__(self):
        sand = ['Wal', 'do']
        self.dig = sand.pop

    def walk(self, x):
        self.wave = lambda y: self.dig(x) + self.dig(y)
        return self
```

Reminder: s.pop(k) removes and returns the item at index k

Discussion Question: Where's Waldo?

Write an expression **with no quotes or +** that evaluates to 'Waldo'

```
class Beach:
    def __init__(self):
        sand = ['Wal', 'do']
        self.dig = sand.pop

    def walk(self, x):
        self.wave = lambda y: self.dig(x) + self.dig(y)
        return self
```

Reminder: s.pop(k) removes and returns the item at index k

```
>>> Beach().walk(0).wave(0)
'Waldo'
```

Composition

Attribute Composition

Attribute values are themselves objects, which have attributes and methods

Any type of object can be an attribute value of any other type

When reading/writing a complicated expression, think about the type of each sub-expression:

- **self** always has the type of the class being defined
- For **self.x.y: self.x** is an object (an attribute of **self**) with an attribute **y.**
- For **self.f(x).y: self.f(x)** is the return value of method **f,** an object with an attribute **y.**
- For **self.f(x).g(y):** the return value of **self**'s method **f** is an object that has a method **g.**

Example: Message Delivery

```
@dataclass
class Email:
    msg: str
    sender: str
    recipient_name: str

class Server:
    def __init__(self):
        self.clients: dict[str, Client] = {}

    def send(self, email: Email) -> None:
        "Append the email to the inbox of the recipient's client."
        _____________________________________________________________

class Client:
    def __init__(self, server: Server, name: str):
        self.server = server
        self.name = name
        self.inbox: list[Email] = []
        server._____________________

    def compose(self, message: str, recipient: str) -> None:
        "Send an email with the given message to the recipient."
        self.server.send(_____________________________________)
```

A **Client** can **compose** an **Email** to **send** it to the **Server**, which appends it to the **inbox** of the **Client** named as the **recipient.**

Each **Server** has a **clients** dictionary from the **name** of each **Client** to that **Client**, and each **Client** adds itself to **clients.**

```
>>> s = Server()
>>> a = Client(s, 'Alice')
>>> b = Client(s, 'Bob')
>>> a.compose('Hello!', 'Bob')
>>> b.inbox[0].msg
'Hello!'
```

Example: Message Delivery

```
@dataclass
class Email:
    msg: str
    sender: str
    recipient_name: str

class Server:
    def __init__(self):
        self.clients: dict[str, Client] = {}

    def send(self, email: Email) -> None:
        "Append the email to the inbox of the recipient's client."
        self.clients[email.recipient_name].inbox.append(email)

class Client:
    def __init__(self, server: Server, name: str):
        self.server = server
        self.name = name
        self.inbox: list[Email] = []
        server.clients[name] = self

    def compose(self, message: str, recipient: str) -> None:
        "Send an email with the given message to the recipient."
        self.server.send(Email(message, self.name, recipient))
```

Server

dict

Client

list[Email]

Email

A **Client** can **compose** an **Email** to **send** it to the **Server**, which appends it to the **inbox** of the **Client** named as the **recipient.**

Each **Server** has a **clients** dictionary from the **name** of each **Client** to that **Client**, and each **Client** adds itself to **clients.**

```
>>> s = Server()
>>> a = Client(s, 'Alice')
>>> b = Client(s, 'Bob')
>>> a.compose('Hello!', 'Bob')
>>> b.inbox[0].msg
'Hello!'
```

Class Attributes

Class Attributes

Class attributes are "shared" across all instances of a class because they are attributes of the class, not the instance

```
class Account:

    interest = 0.02   # A class attribute

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    # Additional methods would be defined here
```

```
>>> tom_account = Account('Tom')
>>> jim_account = Account('Jim')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
```

The **interest** attribute is ***not*** part of the instance; it's part of the class!

(Demo)

Attribute Assignment Statements

Account class attributes

```
interest: 0.02  0.04  0.05
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

```
balance:  0
holder:   'Jim'
interest: 0.08
```

Instance attributes of tom_account

```
balance:  0
holder:   'Tom'
```

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest
0.04
```

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```