Demystifying Python Metaclasses

Eric D. Wills, Ph.D.
Instructor, University of Oregon
Director of R&D, Vizme Inc

What is a metaclass?

- A metaclass specifies the fundamental attributes for a class, such as the class name, parent class(es), and class variables.
- A class constructor creates instances of that class; a metaclass constructor creates classes of that metaclass.

Agenda

- Review Python classes
- Introduce Python metaclasses
- Use case: Dynamic class parenting
- Use case: Dynamic properties

Agenda

- Review Python classes
- Introduce Python metaclasses
- Use case: Dynamic class parenting
- Use case: Dynamic properties

What is a class?

- A class is typically a template for the state and behavior of a real-world phenomenon.
- A constructor method creates specific instances of the class.
- For example, a Car class might specify the heading, velocity, and position state of each car instance and provide methods for accelerating, decelerating, and turning the instance.

Class example

```
class Car(object):
    _MAX_VELOCITY = 100.0

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

    @property
    def velocity(self):
        return self._velocity

    def accelerate(self, acceleration, deltaTime):
        self._velocity += acceleration*deltaTime
        if self._velocity > self.__class__._MAX_VELOCITY:
            self._velocity = self.__class__._MAX_VELOCITY

car = Car(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Class example

```
class Car(object):
    _MAX_VELOCITY = 100.0

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

    @property
    def velocity(self):
        return self._velocity

    def accelerate(self, acceleration, deltaTime):
        self._velocity += acceleration*deltaTime
        if self._velocity > self.__class__._MAX_VELOCITY:
            self._velocity = self.__class__._MAX_VELOCITY

car = Car(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Class example

- `class Car(object):`
  - Defines a class named *Car*. The ultimate parent class for all classes is *object*.

Class example

```
class Car(object):
    _MAX_VELOCITY = 100.0

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

    @property
    def velocity(self):
        return self._velocity

    def accelerate(self, acceleration, deltaTime):
        self._velocity += acceleration*deltaTime
        if self._velocity > self.__class__._MAX_VELOCITY:
            self._velocity = self.__class__._MAX_VELOCITY

car = Car(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Class example

- `_MAX_VELOCITY = 100.0`
  - This is a class variable. All instances of the class share a single copy of this variable. Therefore, a change to the variable’s value by one instance affects all instances.

Class example

```
class Car(object):
    _MAX_VELOCITY = 100.0

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

    @property
    def velocity(self):
        return self._velocity

    def accelerate(self, acceleration, deltaTime):
        self._velocity += acceleration*deltaTime
        if self._velocity > self.__class__._MAX_VELOCITY:
            self._velocity = self.__class__._MAX_VELOCITY

car = Car(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Class example

-
  ```
  def __init__(self, initialVelocity):
      self._velocity = initialVelocity
  ```
  - This is the constructor. When passed an initial velocity as an argument, the _*velocity* instance variable is assigned to the value of the *initialVelocity* parameter.

Class example

```
class Car(object):
    _MAX_VELOCITY = 100.0

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

    @property
    def velocity(self):
        return self._velocity

    def accelerate(self, acceleration, deltaTime):
        self._velocity += acceleration*deltaTime
        if self._velocity > self.__class__._MAX_VELOCITY:
            self._velocity = self.__class__._MAX_VELOCITY

car = Car(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Class example

-
  ```
  @property
  def velocity(self):
      return self._velocity
  ```
  - This is an instance getter. This method uses the built-in *@property* decorator to specify that this method should be called to return a value when the public variable *velocity* is accessed. The method returns the current velocity of the instance.

Class example

```
class Car(object):
    _MAX_VELOCITY = 100.0

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

    @property
    def velocity(self):
        return self._velocity

    def accelerate(self, acceleration, deltaTime):
        self._velocity += acceleration*deltaTime
        if self._velocity > self.__class__._MAX_VELOCITY:
            self._velocity = self.__class__._MAX_VELOCITY

car = Car(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Class example

-
  ```
  def accelerate(self, acceleration, deltaTime):
      self._velocity += acceleration*deltaTime
      if self._velocity > self.__class__._MAX_VELOCITY:
          self._velocity = self.__class__._MAX_VELOCITY
  ```
  - This is an instance method. When called on an instance, the velocity of that instance is updated according to the input parameters and then capped at the max velocity.

Class example

```
class Car(object):
    _MAX_VELOCITY = 100.0

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

    @property
    def velocity(self):
        return self._velocity

    def accelerate(self, acceleration, deltaTime):
        self._velocity += acceleration*deltaTime
        if self._velocity > self.__class__._MAX_VELOCITY:
            self._velocity = self.__class__._MAX_VELOCITY
```

```
car = Car(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Class example

-
  ```
  car = Car(10.0)
  print(car.velocity)
  car.accelerate(100.0, 1.0)
  print(car.velocity)
  ```
  - Prints 10 followed by 100.

Class example

```
class Car(object):
    _MAX_VELOCITY = 100.0

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

    @property
    def velocity(self):
        return self._velocity

    def accelerate(self, acceleration, deltaTime):
        self._velocity += acceleration*deltaTime
        if self._velocity > self.__class__._MAX_VELOCITY:
            self._velocity = self.__class__._MAX_VELOCITY

car = Car(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Agenda

- Review Python classes
- Introduce Python metaclasses
- Use case: Dynamic class parenting
- Use case: Dynamic properties

What is a metaclass again?

- A metaclass specifies the fundamental attributes for a class, such as the class name, parent class(es), and class variables.
- A class constructor creates instances of that class; a metaclass constructor creates classes of that metaclass.

How is this useful?

- Imagine that we now want to create several types of cars, each with a different max velocity. We have a few options:
  - Add a max velocity to the constructor parameters:
    - This would require that we pass the max velocity as an argument each time we create an instance.
  - Create a new class for each type of car:
    - Requires creating a new class just to overload a variable.
  - Use a metaclass to create classes dynamically:
    - Can then use a factory method to create classes at runtime.

Metaclass example

```
class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        return type.__new__(cls, name, bases, attrs)

    @staticmethod
    def createCarClass(carType, maxVelocity):
        return CarMeta('Car_' + carType, (Car,),
                       {'_MAX_VELOCITY':maxVelocity})

Car_Corolla = CarMeta.createCarClass('Corolla', 80.0)
car         = Car_Corolla(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Metaclass example

```
class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        return type.__new__(cls, name, bases, attrs)

    @staticmethod
    def createCarClass(carType, maxVelocity):
        return CarMeta('Car_' + carType, (Car,),
                       {'_MAX_VELOCITY':maxVelocity})

Car_Corolla = CarMeta.createCarClass('Corolla', 80.0)
car         = Car_Corolla(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Metaclass example

- `class CarMeta(type):`
  - Defines a metaclass named *CarMeta*. The ultimate parent class for all metaclasses is *type*.

Metaclass example

```
class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        return type.__new__(cls, name, bases, attrs)

    @staticmethod
    def createCarClass(carType, maxVelocity):
        return CarMeta('Car_' + carType, (Car,),
                       {'_MAX_VELOCITY':maxVelocity})

Car_Corolla = CarMeta.createCarClass('Corolla', 80.0)
car         = Car_Corolla(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Metaclass example

-
  ```
  def __new__(cls, name, bases, attrs):
      return type.__new__(cls, name, bases, attrs)
  ```
  - The constructor can modify the name, base class(es), and class variables for the class being created. None of these need be modified for this example, though, so the constructor is a trivial passthrough.

Metaclass example

```
class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        return type.__new__(cls, name, bases, attrs)

    @staticmethod
    def createCarClass(carType, maxVelocity):
        return CarMeta('Car_' + carType, (Car,),
                       {'_MAX_VELOCITY':maxVelocity})

Car_Corolla = CarMeta.createCarClass('Corolla', 80.0)
car         = Car_Corolla(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Metaclass example

-
  ```
  @staticmethod
  def createCarClass(carType, maxVelocity):
      return CarMeta('Car_' + carType, (Car,),
                   {'_MAX_VELOCITY':maxVelocity})
  ```

  – This method uses the built-in *@staticmethod* decorator to specify that this method is called on the metaclass itself and not classes of the metaclass. The method returns a new class of the metaclass by passing the class name (a string), parent classes (a tuple), and class variables (a dict) to the metaclass constructor.

Metaclass example

```
class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        return type.__new__(cls, name, bases, attrs)

    @staticmethod
    def createCarClass(carType, maxVelocity):
        return CarMeta('Car_' + carType, (Car,),
                       {'_MAX_VELOCITY':maxVelocity})

Car_Corolla = CarMeta.createCarClass('Corolla', 80.0)
car         = Car_Corolla(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Metaclass example

-
  ```
  Car_Corolla = CarMeta.createCarClass('Corolla', 80.0)
  car         = Car_Corolla(10.0)
  print(car.velocity)
  car.accelerate(100.0, 1.0)
  print(car.velocity)
  ```
  - Prints 10 followed by 80.

Metaclass example

```
class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        return type.__new__(cls, name, bases, attrs)

    @staticmethod
    def createCarClass(carType, maxVelocity):
        return CarMeta('Car_' + carType, (Car,),
                       {'_MAX_VELOCITY':maxVelocity})

Car_Corolla = CarMeta.createCarClass('Corolla', 80.0)
car         = Car_Corolla(10.0)
print(car.velocity)
car.accelerate(100.0, 1.0)
print(car.velocity)
```

Agenda

- Review Python classes
- Introduce Python metaclasses
- Use case: Dynamic class parenting
- Use case: Dynamic properties

Dynamic class parenting

- Imagine that we would like to create several similar (or even identical) classes, each of which with a different parent class. For example, we might want to:
  - Create multiple versions of a class, each parented to a different class hierarchy:
    - Useful when extending multiple classes from an API (or multiple APIs).
    - Dynamic parent class can be specified using the child's class name, module path, class variables, etc.

Dynamic parenting example

```
class Car(object):
    pass

class Corolla(object):
    _MAX_VELOCITY = 80.0

class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        return type.__new__(cls, name, bases, attrs)

    @staticmethod
    def createCarClass(carType):
        return CarMeta('Car_' + carType,
                       (Car, globals()[carType]), {})

Car_Corolla = CarMeta.createCarClass('Corolla')
print Car_Corolla._MAX_VELOCITY
```

Dynamic parenting example

```
class Car(object):
    pass

class Corolla(object):
    _MAX_VELOCITY = 80.0

class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        return type.__new__(cls, name, bases, attrs)

    @staticmethod
    def createCarClass(carType):
        return CarMeta('Car_' + carType,
                       (Car, globals()[carType]), {})

Car_Corolla = CarMeta.createCarClass('Corolla')
print Car_Corolla._MAX_VELOCITY
```

Dynamic parenting example

-
  ```
  class Car(object):
      pass

  class Corolla(object):
      _MAX_VELOCITY = 80.0
  ```
  - Define a Car base class to represent the state and behavior of all car classes.
  - Also define a number of possible parent classes (just one here) with state and behavior specific to one type of car.

Dynamic parenting example

```
class Car(object):
    pass

class Corolla(object):
    _MAX_VELOCITY = 80.0

class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        return type.__new__(cls, name, bases, attrs)

    @staticmethod
    def createCarClass(carType):
        return CarMeta('Car_' + carType,
                       (Car, globals()[carType]), {})

Car_Corolla = CarMeta.createCarClass('Corolla')
print Car_Corolla._MAX_VELOCITY
```

Dynamic parenting example

-
  ```
  @staticmethod
  def createCarClass(carType):
      return CarMeta('Car_' + carType,
                     (Car, globals()[carType]),
                     {})
  ```
  - When creating a class of the metaclass, lookup the class with name equal to the value of *carType* and add that class as a parent class to the class to be created.

Dynamic parenting example

```
class Car(object):
    pass

class Corolla(object):
    _MAX_VELOCITY = 80.0

class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        return type.__new__(cls, name, bases, attrs)

    @staticmethod
    def createCarClass(carType):
        return CarMeta('Car_' + carType,
                               (Car, globals()[carType]), {})

Car_Corolla = CarMeta.createCarClass('Corolla')
print Car_Corolla._MAX_VELOCITY
```

Dynamic parenting example

-
  ```
  Car_Corolla = CarMeta.createCarClass('Corolla')
  print Car_Corolla._MAX_VELOCITY
  ```
  - Prints 80.

Dynamic parenting example

```
class Car(object):
    pass

class Corolla(object):
    _MAX_VELOCITY = 80.0

class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        return type.__new__(cls, name, bases, attrs)

    @staticmethod
    def createCarClass(carType):
        return CarMeta('Car_' + carType,
                       (Car, globals()[carType]), {})

Car_Corolla = CarMeta.createCarClass('Corolla')
print Car_Corolla._MAX_VELOCITY
```

At Vizme…

- We primarily use SQLAlchemy as our ORM:
  - A canonical model class is maintained for each table which specifies the columns of the table and provides convenience methods.
  - New child classes of these canonical classes are created dynamically and parented to the SQLAlchemy base class for the database session.
    - Allows multiple database sessions with a single model class per table.

Agenda

- Review Python classes
- Introduce Python metaclasses
- Use case: Dynamic class parenting
- Use case: Dynamic properties

Dynamic properties

- Imagine that we have a class with some instance variables and that we want to execute a similar operation when getting or setting any of the variables. For example, we might want to:
    - Set a dirty bit when the state of the instance changes.
    - Perform queries based on foreign keys outside a model's database.
    - Select from multiple config values based on server state.
    - Retrieve config values from a file or memory.

Dynamic property example (part 1)

```
class CarGetter(object):
    def __init__(self, name):
        self._name = name

    def __call__(self, owner):
        return getattr(owner, self._name)

class CarSetter(object):
    def __init__(self, name):
        self._name = name

    def __call__(self, owner, value):
        print "invalidate!"
        return setattr(owner, self._name, value)
```

Dynamic property example (part 1)

```
class CarGetter(object):
    def __init__(self, name):
        self._name = name

    def __call__(self, owner):
        return getattr(owner, self._name)

class CarSetter(object):
    def __init__(self, name):
        self._name = name

    def __call__(self, owner, value):
        print "invalidate!"
        return setattr(owner, self._name, value)
```

Dynamic property example (part 1)

-
  ```
  class CarGetter(object):
      def __init__(self, name):
          self._name = name
  ```
  – Create a callable class to serve as the property getter. The constructor stores the name of the property for use when the class is called.

Dynamic property example (part 1)

```
class CarGetter(object):
    def __init__(self, name):
        self._name = name

    def __call__(self, owner):
        return getattr(owner, self._name)

class CarSetter(object):
    def __init__(self, name):
        self._name = name

    def __call__(self, owner, value):
        print "invalidate!"
        return setattr(owner, self._name, value)
```

Dynamic property example (part 1)

-
  ```
  def __call__(self, owner):
      return getattr(owner, self._name)
  ```
  - When the class is called we’re implicitly given access to the owner instance, so simply return the value of the specified variable from that instance.

Dynamic property example (part 1)

```
class CarGetter(object):
    def __init__(self, name):
        self._name = name

    def __call__(self, owner):
        return getattr(owner, self._name)

class CarSetter(object):
    def __init__(self, name):
        self._name = name

    def __call__(self, owner, value):
        print "invalidate!"
        return setattr(owner, self._name, value)
```

Dynamic property example (part 1)

-
  ```
  def __call__(self, owner, value):
      print "invalidate!"
      return setattr(owner, self._name, value)
  ```
  - When the class is called we first invalidate the instance (or just indicate that we could do so, in this case). We're again implicitly given access to the owner instance, so we can update the value of the specified variable on that instance.

Dynamic property example (part 1)

```
class CarGetter(object):
    def __init__(self, name):
        self._name = name

    def __call__(self, wrappedSelf):
        return getattr(wrappedSelf, self._name)

class CarSetter(object):
    def __init__(self, name):
        self._name = name

    def __call__(self, wrappedSelf, value):
        print "invalidate!"
        return setattr(wrappedSelf, self._name, value)
```

Dynamic property example (part 2)

```
class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        for name in attrs['_instanceVars']:
            attrs[name[1:]] = property(CarGetter(name),
                                       CarSetter(name))
        return type.__new__(cls, name, bases, attrs)

class Car(object):
    __metaclass__ = CarMeta

    _instanceVars = ['_velocity']

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

car = Car(10)
car.velocity = 100
print(car.velocity)
```

Dynamic property example (part 2)

```
class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        for name in attrs['_instanceVars']:
            attrs[name[1:]] = property(CarGetter(name),
                                       CarSetter(name))
        return type.__new__(cls, name, bases, attrs)

class Car(object):
    __metaclass__ = CarMeta

    _instanceVars = ['_velocity']

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

car = Car(10)
car.velocity = 100
print(car.velocity)
```

Dynamic property example (part 2)

-
  ```
  def __new__(cls, name, bases, attrs):
      for name in attrs['_instanceVars']:
          attrs[name[1:]] = property(CarGetter(name),
                                     CarSetter(name))
      return type.__new__(cls, name, bases, attrs)
  ```
  - When creating a new class of the metaclass, loop through all variable names in the _*instanceVars* class variable of the class being created. For each private variable name in _*instanceVars*, use the built-in *property* function to create a public property based on new CarGetter and CarSetter instances.

Dynamic property example (part 2)

```
class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        for name in attrs['_instanceVars']:
            attrs[name[1:]] = property(CarGetter(name),
                                       CarSetter(name))
        return type.__new__(cls, name, bases, attrs)

class Car(object):
    __metaclass__ = CarMeta

    _instanceVars = ['_velocity']

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

car = Car(10)
car.velocity = 100
print(car.velocity)
```

Dynamic property example (part 2)

-
  ```
  __metaclass__ = CarMeta

  _instanceVars = ['_velocity']
  ```
  - Specify *CarMeta* as the metaclass for the *Car* class. This is an alternate method for specifying a metaclass which ensures that the metaclass constructor is called implicitly when the class is created.
  - Also define the list of instance-variable names for which dynamic properties will be created.

Dynamic property example (part 2)

```
class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        for name in attrs['_instanceVars']:
            attrs[name[1:]] = property(CarGetter(name),
                                       CarSetter(name))
        return type.__new__(cls, name, bases, attrs)

class Car(object):
    __metaclass__ = CarMeta

    _instanceVars = ['_velocity']

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

car = Car(10)
car.velocity = 100
print(car.velocity)
```

Dynamic property example (part 2)

-
  ```
  car = Car(10)
  car.velocity = 100
  print(car.velocity)
  ```
  - Prints “invalidate!” followed by 100.

Dynamic property example (part 2)

```
class CarMeta(type):
    def __new__(cls, name, bases, attrs):
        for name in attrs['_instanceVars']:
            attrs[name[1:]] = property(CarGetter(name),
                                       CarSetter(name))
        return type.__new__(cls, name, bases, attrs)

class Car(object):
    __metaclass__ = CarMeta

    _instanceVars = ['_velocity']

    def __init__(self, initialVelocity):
        self._velocity = initialVelocity

car = Car(10)
car.velocity = 100
print(car.velocity)
```

At Vizme…

- Our ORM models use metaclasses to create properties dynamically:
  - Columns representing foreign keys outside the model's database are assigned property getters which perform the necessary query.
  - Searchable columns are assigned property setters which enforce coherence between the database and search index.
- We maintain a config system which assigns a property getter for each private variable:
  - The getter typically retrieves the appropriate value from Memcached.
    - Allows config values to be modified without restarting the web server.

In summary…

- Metaclasses provide yet another means for avoiding code duplication by dynamically creating similar classes and methods.
- Metaclasses allow classes to crosscut multiple aspects by dynamically parenting to distinct classes.
- Metaclasses aren’t *that* mysterious!

Questions?