Python descriptors:

Python descriptors are a powerfull feature that allow you to manage the behavior of attributes in a class. They are often used to define custom behavior for `getting`, `setting` and `deleting` attributes.

A descriptor is any object that implements at least one of the following methods:

`Methods:`

- `__get__(self, instance, owner):`
- `__set__(self, instance, value):`
- `__delete__(self, instance):`

Descriptors are a `protocol` that allows objects to define how attributes are accessed or modified.

`Basic descriptor defination:`

A descriptor calss must implement at least one of the descriptor methods (`__get__`,`__set__`, `__delete__`).

```
class Descriptor:
    def __init__(self, value=None):
        self.value = value

    def __get__(self, instance, owner):
        print("Getting value")
        return self.value
```

`Implementing All Descriptor Methods:`

```
class Descriptor:
    def __init__(self, value=None):
        self.value = value

    def __get__(self, instance, owner):
        print("Getting value")
        return self.value

    def __set__(self, instance, value):
        print("Setting value")
        self.value = value

    def __delete__(self, instance):
        print("Deleting value")
        del self.value
```

`Using Descriptors in a Class:`

```
class MyClass:
    attr = Descriptor(10)

obj = MyClass()
print(obj.attr)  # Triggers __get__
obj.attr = 20     # Triggers __set__
del obj.attr      # Triggers __delete__
```

Combining Multiple Descriptors:

```
class DescriptorA:
    def __get__(self, instance, owner):
        return "DescriptorA value"

class DescriptorB:
    def __get__(self, instance, owner):
        return "DescriptorB value"

class MyClass:
    attr_a = DescriptorA()
    attr_b = DescriptorB()

obj = MyClass()
print(obj.attr_a)  # "DescriptorA value"
print(obj.attr_b)  # "DescriptorB value"
```

Descriptors with Instance State:

If you need a descriptor to have `state` that's specific to an `instance`, you can use the __init__ method of the descriptor to initialize it:

```
class Descriptor:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, owner):
        return f"Descriptor for {self.name} is {instance.__dict__.get(self.name,
'undefined')}"

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

class MyClass:
    attr1 = Descriptor('attr1')
    attr2 = Descriptor('attr2')

obj = MyClass()
```

```
obj.attr1 = 10
print(obj.attr1)  # "Descriptor for attr1 is 10"
```

Property vs Descriptor:

While descriptors are powerful, Python's property decorator can be simpler for many use cases. Here's how to use property:

```
class MyClass:
    def __init__(self):
        self._attr = None

    @property
    def attr(self):
        return self._attr

    @attr.setter
    def attr(self, value):
        self._attr = value

    @attr.deleter
    def attr(self):
        del self._attr

obj = MyClass()
obj.attr = 10
print(obj.attr)
del obj.attr
```

Validation and Type Checking:

```
class PositiveInteger:
    def __init__(self, value=0):
        self.value = value

    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError("Value must be positive")
        self.value = value

class MyClass:
    attr = PositiveInteger()

obj = MyClass()
obj.attr = 10   # Works fine
```

```
print(obj.attr)  # 10
obj.attr = -5    # Raises ValueError
```

Computed Properties:

Descriptors can be used to create computed properties that are calculated dynamically based on other attributes:

```
class ComputedProperty:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, owner):
        return self.func(instance)

class MyClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @ComputedProperty
    def sum(self):
        return self.x + self.y

obj = MyClass(3, 4)
print(obj.sum)  # 7
```

Lazy Initialization:

Descriptors can be used to implement lazy initialization, where an attribute is computed or fetched only when needed:

```
class LazyProperty:
    def __init__(self, func):
        self.func = func
        self.cache_name = f"_{func.__name__}_cache"

    def __get__(self, instance, owner):
        if not hasattr(instance, self.cache_name):
            value = self.func(instance)
            setattr(instance, self.cache_name, value)
        return getattr(instance, self.cache_name)

class MyClass:
    def __init__(self, data):
        self.data = data

    @LazyProperty
    def expensive_computation(self):
```

```
        # Simulate a costly computation
        print("Computing...")
        return sum(self.data)

obj = MyClass([1, 2, 3, 4, 5])
print(obj.expensive_computation)  # Computes and prints "Computing...", then 15
print(obj.expensive_computation)  # Prints 15, without computing again
```

Delegation Pattern:

Descriptors can delegate attribute access to another object. This pattern can be useful for encapsulating behavior or delegating responsibilities:

```
class Delegate:
    def __init__(self):
        self._data = {}

    def __getattr__(self, item):
        return self._data.get(item)

    def __setattr__(self, key, value):
        if key == '_data':
            super().__setattr__(key, value)
        else:
            self._data[key] = value

class Container:
    def __init__(self):
        self.delegate = Delegate()

    def __getattr__(self, item):
        return getattr(self.delegate, item)

    def __setattr__(self, key, value):
        if key == 'delegate':
            super().__setattr__(key, value)
        else:
            setattr(self.delegate, key, value)

container = Container()
container.some_attr = "Hello"
print(container.some_attr)  # "Hello"
```

Binding Descriptors to Different Classes:

```
class SharedDescriptor:
    def __init__(self, value):
        self.value = value
```

```
    def __get__(self, instance, owner):
        return self.value

class ClassA:
    shared = SharedDescriptor("Shared Value")

class ClassB:
    shared = SharedDescriptor("Shared Value")

a = ClassA()
b = ClassB()

print(a.shared)  # "Shared Value"
print(b.shared)  # "Shared Value"
```

Dynamic Attribute Creation:

```
class DynamicAttributes:
    def __init__(self):
        self._data = {}

    def __getattr__(self, name):
        if name in self._data:
            return self._data[name]
        else:
            raise AttributeError(f"Attribute {name} not found")

    def __setattr__(self, name, value):
        if name == '_data':
            super().__setattr__(name, value)
        else:
            self._data[name] = value

    def add_attribute(self, name, value):
        self._data[name] = value

obj = DynamicAttributes()
obj.add_attribute('dynamic_attr', 42)
print(obj.dynamic_attr)  # 42
```

Summary:

- `Descriptors` are objects that manage attribute access.
- Implement at least one of __get__, __set__, or __delete__.
- `Properties` can be simpler and are often used in place of descriptors.
- Descriptors are useful for scenarios where you need more control over attribute access, such as data validation or computed properties.
- `Validation:` Ensure attributes meet specific criteria.
- `Computed Properties:` Create dynamic attributes based on other attributes.

- `Lazy Initialization:` Delay the computation of an attribute until it is accessed.
- `Delegation:` Delegate attribute management to another object.
- `Reusable Descriptors:` Share descriptors across multiple classes.
- `Dynamic Attributes:` Manage attributes that are created at runtime.