

# Describing Descriptors

PyCon Ireland 2023

by Rodrigo Girão Serrão



Rodrigo 🐍🚀  
🐦 @mathsppblog



Rodrigo 🐍🚀

🐦 @mathsppblog

Textualize  
[mathspp.com](http://mathspp.com)

# Code

# Summary

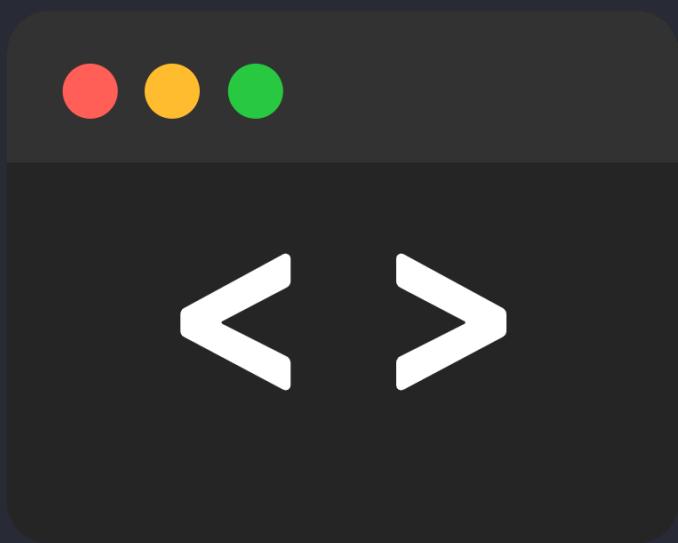
- `__get__` runs on attribute access;

- `__get__` runs on attribute access;
- `__set__` runs on attribute assignment;

- `__get__` runs on attribute access;
- `__set__` runs on attribute assignment; and
- `__set_name__` notifies descriptor of attribute name;

# presented with snappify\*

\*no affiliation, they're just awesome





Backup talk slides ->

# Attributes

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
        self.name = f"{first} {last}"
```

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
        self.name = f"{first} {last}"
```

Attribute name  
depends on other  
attributes.

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
        self.name = f"{first} {last}"
```

```
john = Person("John", "Doe")  
print(john.name) # John Doe
```

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
        self.name = f"{first} {last}"
```

```
john = Person("John", "Doe")  
print(john.name) # John Doe  
john.last = "Smith"  
print(john.name) # ???
```

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
        self.name = f"{first} {last}"
```

```
john = Person("John", "Doe")  
print(john.name) # John Doe  
john.last = "Smith"  
print(john.name) # John Doe
```



```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
        self.name = f"{first} {last}"
```

```
john = Person("John", "Doe")  
print(john.name) # John Doe ←  
john.last = "Smith"  
print(john.name) # John Doe ←
```

## Person.name:

- has dependencies; but
- only computed once.

# Dynamic attribute computation

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
        self.name = f"{self.first} {self.last}"
```

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
    def name(self):  
        return f"{self.first} {self.last}"
```

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
    @property  
    def name(self):  
        return f"{self.first} {self.last}"
```

```
class Person:                                john = Person("John", "Doe")
    def __init__(self, first, last):
        self.first = first
        self.last = last

@property
def name(self):
    return f"{self.first} {self.last}"
```

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last
```

*@property*

```
def name(self):  
    return f"{self.first} {self.last}"
```

```
john = Person("John", "Doe")  
print(john.name)
```

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last
```

*@property*

```
def name(self): <--  
    return f"{self.first} {self.last}"
```

```
john = Person("John", "Doe")  
print(john.name)
```

Accessing the  
attribute calls  
the method.

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last
```

*@property*

```
def name(self): <--  
    return f"{self.first} {self.last}"
```

```
john = Person("John", "Doe")  
print(john.name) # John Doe
```

Accessing the  
attribute calls  
the method.

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last
```

*@property*

```
def name(self):  
    return f"{self.first} {self.last}"
```

```
john = Person("John", "Doe")  
print(john.name) # John Doe  
john.last = "Smith"  
print(john.name) # ???
```

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
    @property  
    def name(self): <--  
        return f"{self.first} {self.last}"
```

```
john = Person("John", "Doe")  
print(john.name) # John Doe  
john.last = "Smith"  
print(john.name) # John Smith
```



Because we call the method, the value is recomputed.

@property “turns methods into attributes.”

Your first  
descriptor

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
    @property  
    def name(self):  
        return f"{self.first} {self.last}"
```

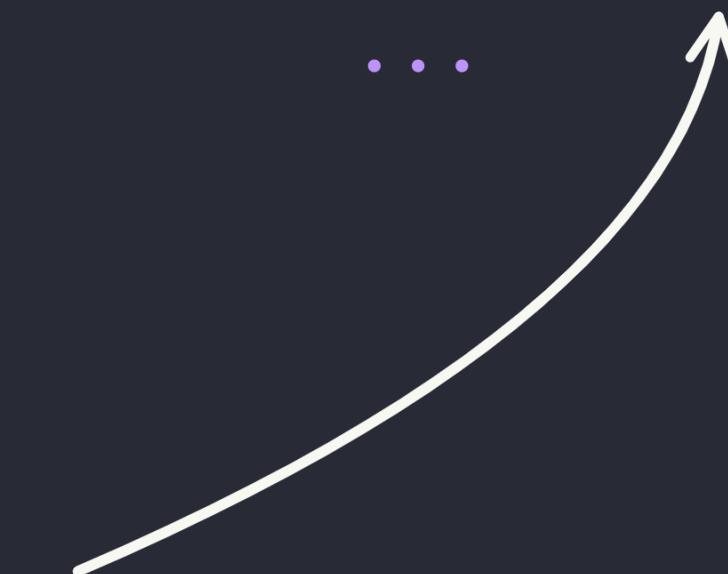
```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last
```

```
@property  
def name(self):  
    return f"{self.first} {self.last}"
```

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last
```

```
class NameDescriptor:
```

```
...
```



We create a class for the descriptor.

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
    @property  
    def name(self):  
        return f"{self.first} {self.last}"
```

```
class Person:  
    name = NameDescriptor()  
  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
class NameDescriptor:  
    ...
```



We assign the descriptor to  
the class attribute name.

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
    @property  
    def name(self):  
        return f"{self.first} {self.last}"
```

```
class Person:  
    name = NameDescriptor()  
  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
class NameDescriptor:  
    def __get__(self, person, cls):  
        ...
```



The dunder method `__get__` is called  
when we try to “get” the attribute...

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
    @property  
    def name(self):  
        return f"{self.first} {self.last}"
```

```
class Person:  
    name = NameDescriptor()  
  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
class NameDescriptor:  
    def __get__(self, person, cls):  
        return f"{person.first} {person.last}"
```

... so it must return the computed value.

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
    @property  
    def name(self):  
        return f"{self.first} {self.last}"
```

# get in action

```
class Person:  
    name = NameDescriptor()  
  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
class NameDescriptor:  
    def __get__(self, person, cls):  
        return f"{person.first} {person.last}"
```

... so it must return the computed value.

```
class Person:  
    name = NameDescriptor()  
  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
class NameDescriptor:  
    def __get__(self, person, cls):  
        print("Inside __get__") ←  
        return f"{person.first} {person.last}"
```

Added a helper print.

```
class Person:                                     >>> john = Person("John", "Doe")
    name = NameDescriptor()

    def __init__(self, first, last):
        self.first = first
        self.last = last

class NameDescriptor:
    def __get__(self, person, cls):
        print("Inside __get__")
        return f"{person.first} {person.last}"
```

```
class Person:  
    name = NameDescriptor() <--  
  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
class NameDescriptor:  
    def __get__(self, person, cls):  
        print("Inside __get__")  
        return f"{person.first} {person.last}"  
  
    >>> john = Person("John", "Doe")  
    >>> john.name
```



```
class Person:  
    name = NameDescriptor() ←  
  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
class NameDescriptor:  
    def __get__(self, person, cls):  
        print("Inside __get__")  
        return f"{person.first} {person.last}"  
  
    >>> john = Person("John", "Doe")  
    >>> john.name
```

The diagram illustrates the flow of the 'name' attribute. It starts with the assignment 'name = NameDescriptor()' in the Person class. A curved arrow points from this assignment to the 'name' attribute in the Person class definition. Another curved arrow points from the 'name' attribute to the \_\_get\_\_ method of the NameDescriptor class. This visualizes how the attribute reference is resolved through the descriptor protocol.

```
class Person:  
    name = NameDescriptor()  
  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
  
class NameDescriptor:  
    def __get__(self, person, cls):  
        print("Inside __get__")  
        return f"{person.first} {person.last}"
```

```
>>> john = Person("John", "Doe")  
>>> john.name  
Inside __get__  
'John Doe'
```

```
class Person:  
    name = NameDescriptor()  
  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
  
class NameDescriptor:  
    def __get__(self, person, cls):  
        print("Inside __get__")  
        return f"{person.first} {person.last}"
```

```
>>> john = Person("John", "Doe")  
>>> john.name  
Inside __get__  
'John Doe'  
>>> john.last = "Smith"
```

```
class Person:  
    name = NameDescriptor()  
  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
class NameDescriptor:  
    def __get__(self, person, cls):  
        print("Inside __get__")  
        return f"{person.first} {person.last}"
```

```
>>> john = Person("John", "Doe")  
>>> john.name  
Inside __get__  
'John Doe'  
>>> john.last = "Smith"  
>>> john.name  
Inside __get__  
'John Smith'
```

The descriptor's `__get__` runs  
when we access the attribute.

A more general  
descriptor

```
>>> red = Colour("#ff0000")
```

```
>>> red = Colour("#ff0000")
>>> red.r
255
```

```
>>> red = Colour("#ff0000")
>>> red.r
255
>>> red.g, red.b
(0, 0)
```

```
class Colour:  
    def __init__(self, hex_repr):  
        self.hex = hex_repr  
  
>>> red = Colour("#ff0000")  
>>> red.r  
255  
>>> red.g, red.b  
(0, 0)
```

```
class Colour:  
    def __init__(self, hex_repr):  
        self.hex = hex_repr  
  
>>> red = Colour("#ff0000")  
>>> red.r  
255  
>>> red.g, red.b  
(0, 0)
```

*property*

```
def r(self):  
    return int(self.hex[1:3], 16)
```

```
class Colour:  
    def __init__(self, hex_repr):  
        self.hex = hex_repr  
  
>>> red = Colour("#ff0000")  
>>> red.r  
255  
>>> red.g, red.b  
(0, 0)  
  
@property  
def r(self):  
    return int(self.hex[1:3], 16)  
  
@property  
def g(self):  
    return int(self.hex[3:5], 16)
```

```
class Colour:  
    def __init__(self, hex_repr):  
        self.hex = hex_repr  
  
>>> red = Colour("#ff0000")  
>>> red.r  
255  
>>> red.g, red.b  
(0, 0)  
  
    @property  
    def r(self):  
        return int(self.hex[1:3], 16)  
  
    @property  
    def g(self):  
        return int(self.hex[3:5], 16)  
  
    @property  
    def b(self):  
        return int(self.hex[5:7], 16)
```

The properties only  
differ in the slice.

```
class Colour:  
    def __init__(self, hex_repr):  
        self.hex = hex_repr  
  
    @property  
    def r(self):  
        return int(self.hex[1:3], 16)  
  
    @property  
    def g(self):  
        return int(self.hex[3:5], 16)  
  
    @property  
    def b(self):  
        return int(self.hex[5:7], 16)
```

The properties only  
differ in the slice.

We can replace  
them with a single  
descriptor.

```
class Colour:  
    def __init__(self, hex_repr):  
        self.hex = hex_repr  
  
    @property  
    def r(self):  
        return int(self.hex[1:3], 16)  
  
    @property  
    def g(self):  
        return int(self.hex[3:5], 16)  
  
    @property  
    def b(self):  
        return int(self.hex[5:7], 16)
```

```
class Colour:  
    r = # ...  
    g = # ...  
    b = # ...  
  
    def __init__(self, hex_repr):  
        self.hex = hex_repr
```

Class attributes that  
will be assigned the  
descriptor.

```
class Colour:  
    def __init__(self, hex_repr):  
        self.hex = hex_repr  
  
    @property  
    def r(self):  
        return int(self.hex[1:3], 16)  
  
    @property  
    def g(self):  
        return int(self.hex[3:5], 16)  
  
    @property  
    def b(self):  
        return int(self.hex[5:7], 16)
```

```
class Colour:  
    r = RGBComponent(...)  
    g = RGBComponent(...)  
    b = RGBComponent(...)  
  
def __init__(self, hex_repr):  
    self.hex = hex_repr
```

```
class Colour:  
    def __init__(self, hex_repr):  
        self.hex = hex_repr  
  
    @property  
    def r(self):  
        return int(self.hex[1:3], 16)  
  
    @property  
    def g(self):  
        return int(self.hex[3:5], 16)  
  
    @property  
    def b(self):  
        return int(self.hex[5:7], 16)
```

A class for the  
descriptor.

```
class Colour:  
    r = RGBComponent(...)  
    g = RGBComponent(...)  
    b = RGBComponent(...)  
  
def __init__(self, hex_repr):  
    self.hex = hex_repr
```

```
class Colour:  
    def __init__(self, hex_repr):  
        self.hex = hex_repr  
  
    @property  
    def r(self):  
        return int(self.hex[1:3], 16)  
  
    @property  
    def g(self):  
        return int(self.hex[3:5], 16)  
  
    @property  
    def b(self):  
        return int(self.hex[5:7], 16)
```



We specify the component  
via the `__init__`.

```
class Colour:  
    r = RGBComponent(1, 3)  
    g = RGBComponent(...)  
    b = RGBComponent(...)  
  
def __init__(self, hex_repr):  
    self.hex = hex_repr
```

```
class Colour:  
    def __init__(self, hex_repr):  
        self.hex = hex_repr  
  
    @property  
    def r(self):  
        return int(self.hex[1:3], 16)  
  
    @property  
    def g(self):  
        return int(self.hex[3:5], 16)  
  
    @property  
    def b(self):  
        return int(self.hex[5:7], 16)
```



We specify the component  
via the `__init__`.

```
class Colour:  
    r = RGBComponent(1, 3)  
    g = RGBComponent(3, 5)  
    b = RGBComponent(5, 7)  
  
def __init__(self, hex_repr):  
    self.hex = hex_repr
```

```
class RGBComponent:  
    def __init__(self, start, end):  
        self.start = start  
        self.end = end
```

We specify the component  
via the `__init__`.

```
class Colour:  
    def __init__(self, hex_repr):  
        self.hex = hex_repr  
  
    @property  
    def r(self):  
        return int(self.hex[1:3], 16)  
  
    @property  
    def g(self):  
        return int(self.hex[3:5], 16)  
  
    @property  
    def b(self):  
        return int(self.hex[5:7], 16)
```

```
class RGBComponent:  
    def __init__(self, start, end):  
        self.start = start  
        self.end = end
```

```
class RGBComponent:  
    def __init__(self, start, end):  
        self.start = start  
        self.end = end  
  
    def __get__(self, colour, cls):  
        return int(  
            colour.hex[self.start:self.end],  
            16,  
        )
```



\_\_get\_\_ replaces the three properties.

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last
```

# Dynamic attribute assignment

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
    @property  
    def name(self):  
        return f"{self.first} {self.last}"
```

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
    @property  
    def name(self):  
        return f"{self.first} {self.last}"  
  
    @name.setter  
    def name(self, new_name):  
        self.first, self.last = new_name.split()
```

Properties can be used  
to set attributes, too.

Added a helper print.

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
    @property  
    def name(self):  
        return f"{self.first} {self.last}"  
  
    @name.setter  
    def name(self, new_name):  
        print("Setting the name!")  
        self.first, self.last = new_name.split()
```

```
>>> john = Person("John", "Doe")
```

```
class Person:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
  
    @property  
    def name(self):  
        return f"{self.first} {self.last}"  
  
    @name.setter  
    def name(self, new_name):  
        print("Setting the name!")  
        self.first, self.last = new_name.split()
```

```
>>> john = Person("John", "Doe")
>>> john.name = "Charles Smith"
Setting the name!
```

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def name(self):
        return f"{self.first} {self.last}"

    @name.setter
    def name(self, new_name):
        print("Setting the name!")
        self.first, self.last = new_name.split()
```

```
>>> john = Person("John", "Doe")
>>> john.name = "Charles Smith"
Setting the name!
>>> john.first
'Charles'
>>> john.last
'Smith'
```

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def name(self):
        return f"{self.first} {self.last}"

    @name.setter
    def name(self, new_name):
        print("Setting the name!")
        self.first, self.last = new_name.split()
```

```
class Person:  
    ...  
  
    @name.setter  
    def name(self, new_name):  
        self.first, self.last = \  
            new_name.split()
```

# Dynamic attribute assignment

```
class Person:  
    @name.setter  
    def name(self, new_name):  
        self._name = new_name.split()
```

# Dynamic attribute assignment with descriptors

```
class Person:
```

```
    ...
```

```
        def __setattr__(self, name, value):
```

```
            if name == 'name':
```

```
                self._name = value
```

```
            else:
```

```
                setattr(self, name, value)
```

```
class Person:  
    ...  
  
    @name.setter  
    def name(self, new_name):  
        self.first, self.last = \  
            new_name.split()
```

```
class Person:  
    name = NameDescriptor()  
    ...  
  
class Person:  
    ...  
  
    @name.setter  
    def name(self, new_name):  
        self.first, self.last = \  
            new_name.split()
```

```
class NameDescriptor:  
    ...
```

```
class Person:  
    ...
```

```
@name.setter  
def name(self, new_name):  
    self.first, self.last = \  
        new_name.split()
```

```
class Person:  
    name = NameDescriptor()  
    ...
```

```
class NameDescriptor:  
    ...  
    def __set__(self, person, new_name):  
        ...
```



\_\_set\_\_ is responsible for setting attributes.

```
class Person:  
    ...  
  
    @name.setter  
    def name(self, new_name):  
        self.first, self.last = \  
            new_name.split()
```

```
class Person:  
    name = NameDescriptor()  
    ...  
  
class NameDescriptor:  
    ...  
  
    def __set__(self, person, new_name):  
        person.first, person.last = \  
            new_name.split()
```



It mirrors the setter method.

```
class Person:  
    name = NameDescriptor()  
    ...
```

```
class NameDescriptor:  
    ...
```

# \_\_set\_\_ in action

```
def __set__(self, person, new_name):  
    print("Inside __set__")  
    person.first, person.last = \  
        new_name.split()
```

```
class Person:  
    name = NameDescriptor()  
    ...  
  
class NameDescriptor:  
    ...  
  
    def __set__(self, person, new_name):  
        print("Inside __set__")  
        person.first, person.last = \  
            new_name.split()
```

```
class Person:                                     >>> john = Person("John", "Doe")
    name = NameDescriptor()
    ...
    ...
class NameDescriptor:
    ...
def __set__(self, person, new_name):
    print("Inside __set__")
    person.first, person.last = \
        new_name.split()
```

```
class Person:  
    name = NameDescriptor()  
    ...
```

```
class NameDescriptor:  
    ...
```

```
def __set__(self, person, new_name):  
    print("Inside __set__")  
    person.first, person.last = \  
        new_name.split()
```

```
>>> john = Person("John", "Doe")  
>>> john.name = "Charles Smith"
```



```
class Person:  
    name = NameDescriptor()  
    ...
```

```
class NameDescriptor:  
    ...
```

```
def __set__(self, person, new_name):  
    print("Inside __set__")  
    person.first, person.last = \  
        new_name.split()
```

```
>>> john = Person("John", "Doe")  
>>> john.name = "Charles Smith"  
Inside __set__
```

Assignment triggered  
the dunder method  
\_\_set\_\_.

```
class Person:  
    name = NameDescriptor()  
    ...  
  
class NameDescriptor:  
    ...  
  
    def __set__(self, person, new_name):  
        print("Inside __set__")  
        person.first, person.last = \  
            new_name.split()
```

```
>>> john = Person("John", "Doe")  
>>> john.name = "Charles Smith"  
Inside __set__  
>>> john.first  
'Charles'  
>>> john.last  
'Smith'
```

The descriptor's `__set__` runs  
when we assign to an attribute.

# Read-only attributes

How?

```
class Person:  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```

```
class Person:  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```



Leading `_` is a convention:

- `_first` is “private”
- `_last` is “private”

```
class Person:  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```

*@property*  
def first(self):  
 return self.\_first



We add a property around the  
“private” attribute.

```
class Person:  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last  
  
    @property  
    def first(self):  
        return self._first  
  
    @property  
    def last(self):  
        return self._last
```

```
>>> john = Person("John", "Doe")
>>> john.first = "Charles"
AttributeError: ...
```

Trying to assign to  
the attribute raises  
an error...

```
class Person:
    def __init__(self, first, last):
        self._first = first
        self._last = last

    @property
    def first(self):
        return self._first

    @property
    def last(self):
        return self._last
```

Can we make it better?

```
class Person:  
    first = ReadOnlyAttr()  
    last = ReadOnlyAttr()  
  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last  
  
    class ReadOnlyAttr:  
        def __get__(self, instance, owner):  
            return instance._first  
        def __set__(self, instance, value):  
            raise AttributeError("Can't set attribute")
```

# Read-only attributes with descriptors

```
class Person:  
    first = ReadOnlyAttr()  
    last = ReadOnlyAttr()  
  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```

class ReadOnlyAttr:

...



Descriptor for read-only attributes.

```
class ReadOnlyAttr:  
    def __get__(self, obj, cls):  
        ...
```



How can the descriptor  
know which attribute  
we're accessing?

```
class Person:  
    first = ReadOnlyAttr()  
    last = ReadOnlyAttr()  
  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```

```
class ReadOnlyAttr:  
    def __get__(self, obj, cls):  
        ...  
  
    def __set_name__(self, cls, name):  
        ...
```

We can use the dunder  
method `__set_name__`.

```
class Person:  
    first = ReadOnlyAttr()  
    last = ReadOnlyAttr()  
  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```

# Read-only attributes

## with descriptors:

We can use the dunder  
method `__set_name__`

```
class ReadOnlyAttr:  
    def __init__(self, origin):  
        self._name = origin  
        self._value = None  
  
    def __get__(self, instance, owner):  
        return self._value  
  
    def __set__(self, instance, value):  
        if self._value is not None:  
            raise ValueError("Can't change value")  
        self._value = value  
  
class Person:  
    first = ReadOnlyAttr("first")  
    last = ReadOnlyAttr()  
  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```

```
class ReadOnlyAttr:  
    def __get__(self, obj, cls):  
        ...  
  
    def __set_name__(self, cls, name):  
        ...
```

We can use the dunder  
method `__set_name__`.

```
class Person:  
    first = ReadOnlyAttr()  
    last = ReadOnlyAttr()  
  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```

```
class ReadOnlyAttr:  
    def __get__(self, obj, cls):  
        ...  
  
    def __set_name__(self, cls, name):  
        ...
```

\_\_set\_name\_\_ notifies  
the descriptor of its  
attribute's name.

```
class Person:  
    first = ReadOnlyAttr()  
    last = ReadOnlyAttr()  
  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```

```
class ReadOnlyAttr:  
    def __get__(self, obj, cls):  
        ...  
  
    def __set_name__(self, cls, name):  
        self.private_attr_name = \  
            f"_{name}"
```

self is the descriptor,  
not the person.

```
class Person:  
    first = ReadOnlyAttr()  
    last = ReadOnlyAttr()  
  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```

```
class ReadOnlyAttr:  
    def __get__(self, obj, cls):  
        return getattr(←  
            obj,  
            self.private_attr_name,  
        )  
  
    def __set_name__(self, cls, name):  
        self.private_attr_name = \  
            f"_{name}"
```

```
class Person:  
    first = ReadOnlyAttr()  
    last = ReadOnlyAttr()  
  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```

Get the value with the  
private attribute name.

```
class ReadOnlyAttr:  
    def __get__(self, obj, cls):  
        return getattr(  
            obj,  
            self.private_attr_name,  
        )  
  
    def __set_name__(self, cls, name):  
        self.private_attr_name = \  
            f"_ {name}"
```

```
class Person:  
    first = ReadOnlyAttr()  
    last = ReadOnlyAttr()  
  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```



After assignment,  
\_\_set\_name\_\_ receives the  
class and the attribute name.

```
class ReadOnlyAttr:  
    def __get__(self, obj, cls):  
        return getattr(  
            obj,  
            self.private_attr_name,  
        )  
  
    def __set_name__(self, cls, name):  
        self.private_attr_name = \  
            f"_{name}"
```

```
class Person:  
    first = ReadOnlyAttr()  
    last = ReadOnlyAttr()  
  
    def __init__(self, first, last):  
        self._first = first  
        self._last = last
```

The ReadOnlyAttr instance gets  
private\_attr\_name = "\_first"

`__set_name__` makes descriptors “aware”  
of the attributes they were assigned to.

*That's all folks!*

A blurred, out-of-focus photograph of a person's face. The person has dark hair and appears to be wearing glasses. The lighting is dramatic, with strong red and blue hues, creating a moody and abstract effect.

Hell no!

I showed you enough  
to get you started.

**What's next?**

# What's next?

- descriptor protocol

## What's next?

- descriptor protocol
- property, staticmethod, and classmethod

## What's next?

- descriptor protocol
- property, staticmethod, and classmethod
- descriptors, methods, and self
- ...

# References

# References

- <https://mathspp.com/blog/pydonts/properties>
- <https://mathspp.com/blog/pydonts/describing-descriptors>\*
- <https://docs.python.org/3/howto/descriptor.html>
- <https://tushar.lol/post/descriptors/>

\*link from the future

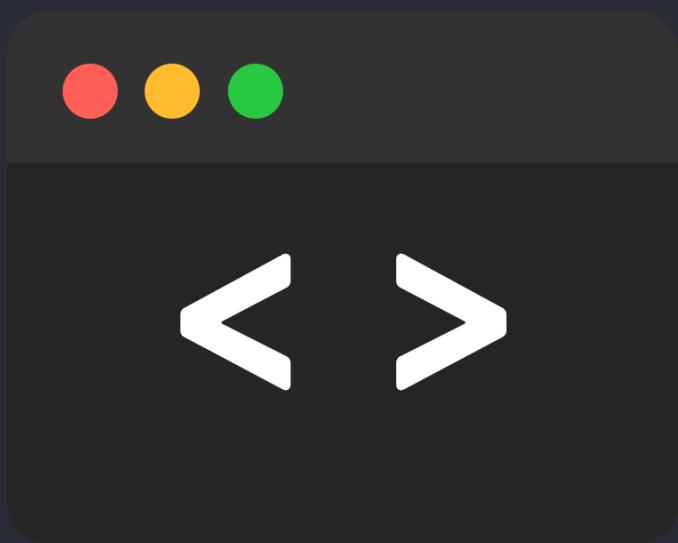
# Pydon'ts

Write elegant  code

[mathspp.com/pydnts](https://mathspp.com/pydnts)

# presented with snappify\*

\*no affiliation, they're just awesome



rodrigo@mathsp.com