Describing Descriptors

PyCon Sri Lanka 2023

by Rodrigo Girão Serrão

Rodrigo 🐍🚀

@mathsppblog

Rodrigo 🐍🚀

@mathsppblog

Textualize

mathspp.com

Slides:

github.com/mathspp/talks

Attributes

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

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

Attribute name depends on other attributes.

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

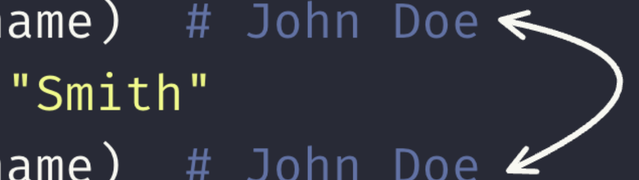
```python
john = Person("John", "Doe")
print(john.name)  # John Doe
```

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

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

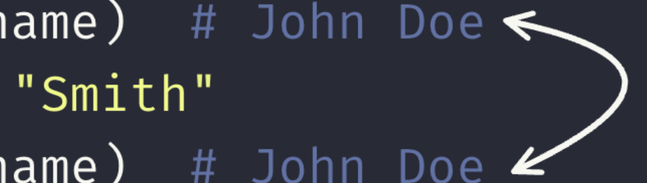
```python
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.name = f"{first} {last}"
```

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



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

```python
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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Accessing the attribute calls the method.

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

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

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

Accessing the attribute calls the method.

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

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

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

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

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

```python
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.

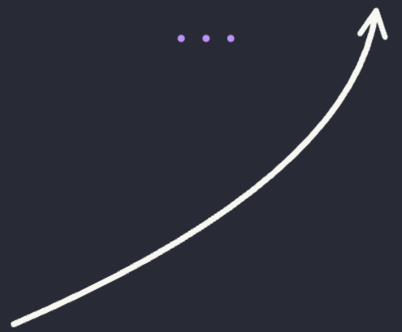
`@property` “turns methods into attributes.”

Your first descriptor

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

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

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

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

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

class NameDescriptor:
    ...
```



We create a class for the descriptor.

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

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

```python
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`.

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

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

```python
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...

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

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

```python
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}"
```

```
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}"
```

`__get__` in action

... 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`.

```python
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")
```


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

```python
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'
```

```python
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"
```

```python
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)
```

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

```
class Colour:
    def __init__(self, hex_repr):
        self.hex = hex_repr

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

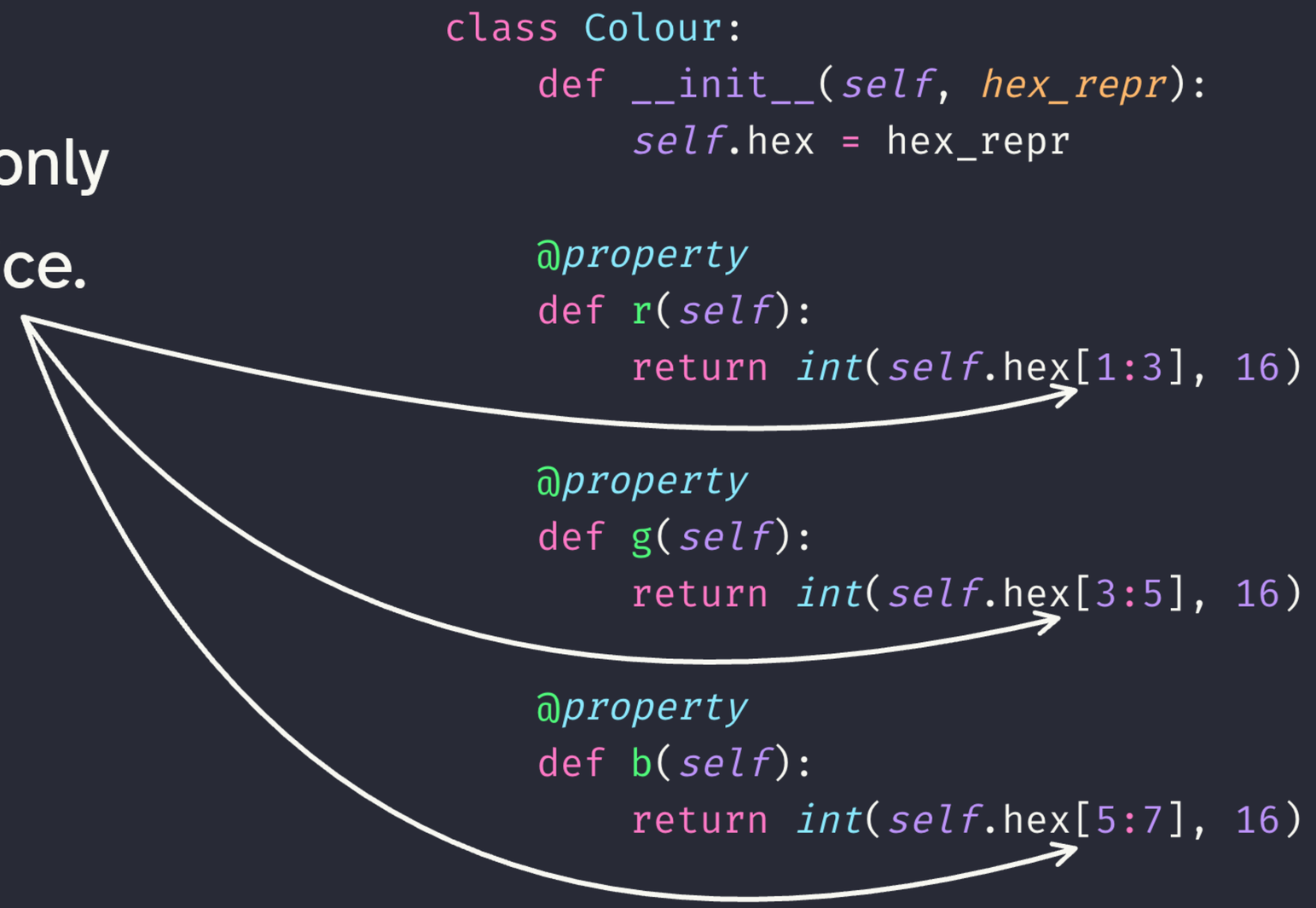
```
>>> red = Colour("#ff0000")
>>> red.r
255
>>> red.g, red.b
(0, 0)
```

```
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)
```

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

```
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.



```
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)
```

```python
class Colour:
    r = RGBComponent(...)
    g = RGBComponent(...)
    b = RGBComponent(...)

    def __init__(self, hex_repr):
        self.hex = hex_repr

class RGBComponent:
    ...
```

A class for the descriptor.

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

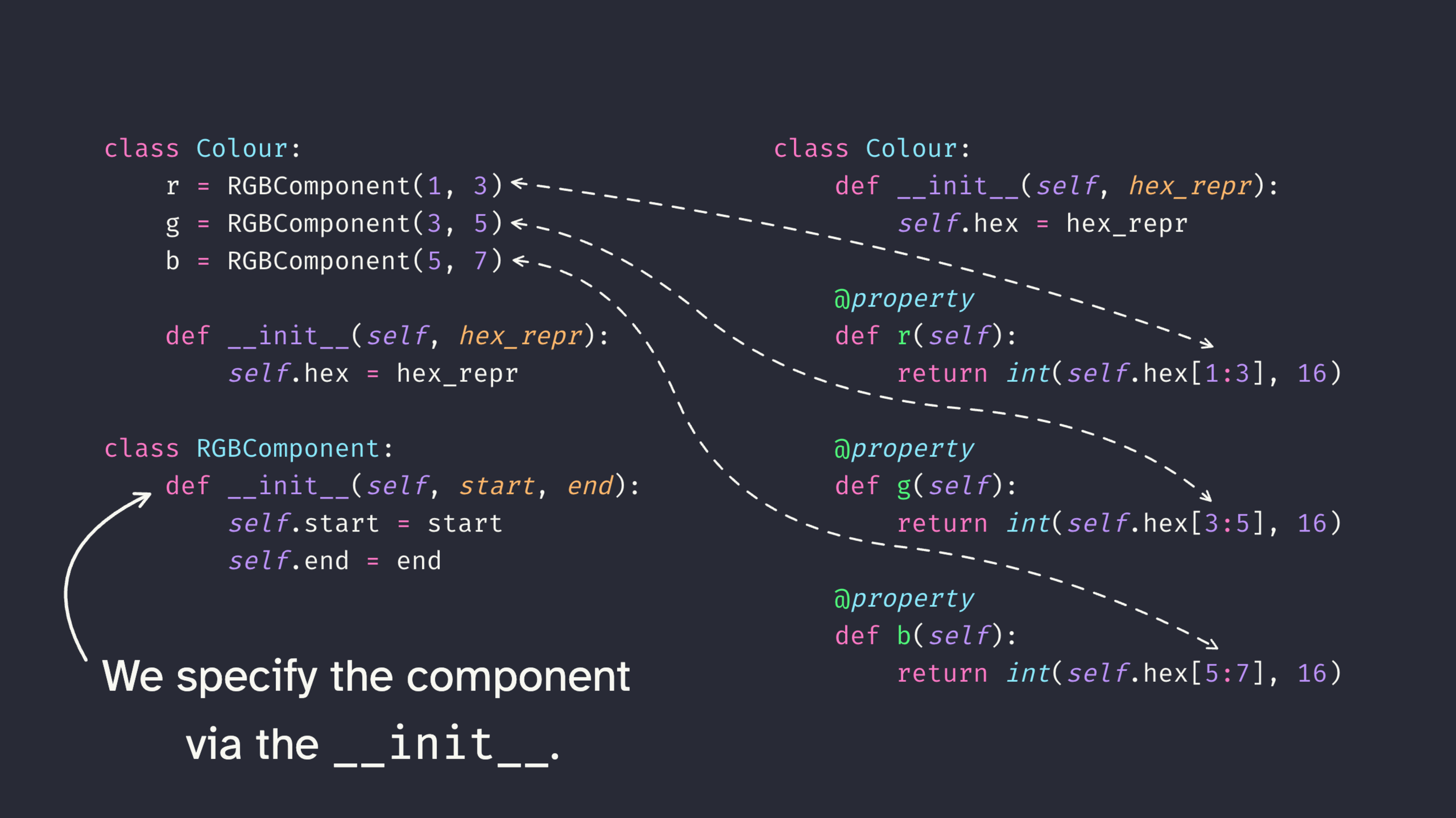
We specify the component via the `__init__`.

```
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 Colour:
    r = RGBComponent(1, 3)
    g = RGBComponent(...)
    b = RGBComponent(...)

    def __init__(self, hex_repr):
        self.hex = hex_repr

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

```
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
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 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

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

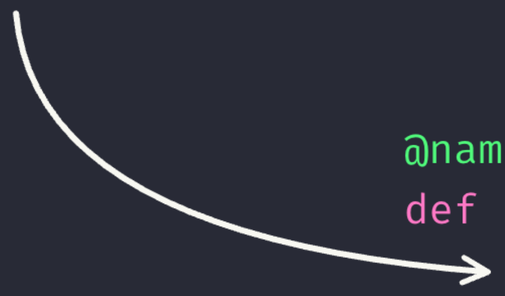
Dynamic attribute assignment

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

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

```python
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`.



```python
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

Dynamic attribute assignment with descriptors

```
class Person:
    ...

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

```
class Person:
    ...

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

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

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

__set__ in action

```
class Person:
    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")
```

```
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?

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

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

Leading _ is a convention:

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

```python
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?

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
```

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

    def __set_name__(self, cls, name):
        ...
```

```python
class Person:
    first = ReadOnlyAttr()
    last = ReadOnlyAttr()

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

We can use the dunder method `__set_name__`.

Read-only attributes with descriptors:

`__set_name__`

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

    def __set_name__(self, cls, name):
        ...
```

```python
class Person:
    first = ReadOnlyAttr()
    last = ReadOnlyAttr()

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

We can use the dunder method `__set_name__`.

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

    def __set_name__(self, cls, name):
        ...
```

```python
class Person:
    first = ReadOnlyAttr()
    last = ReadOnlyAttr()

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

`__set_name__` notifies the descriptor of its attribute's name.

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

    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
```

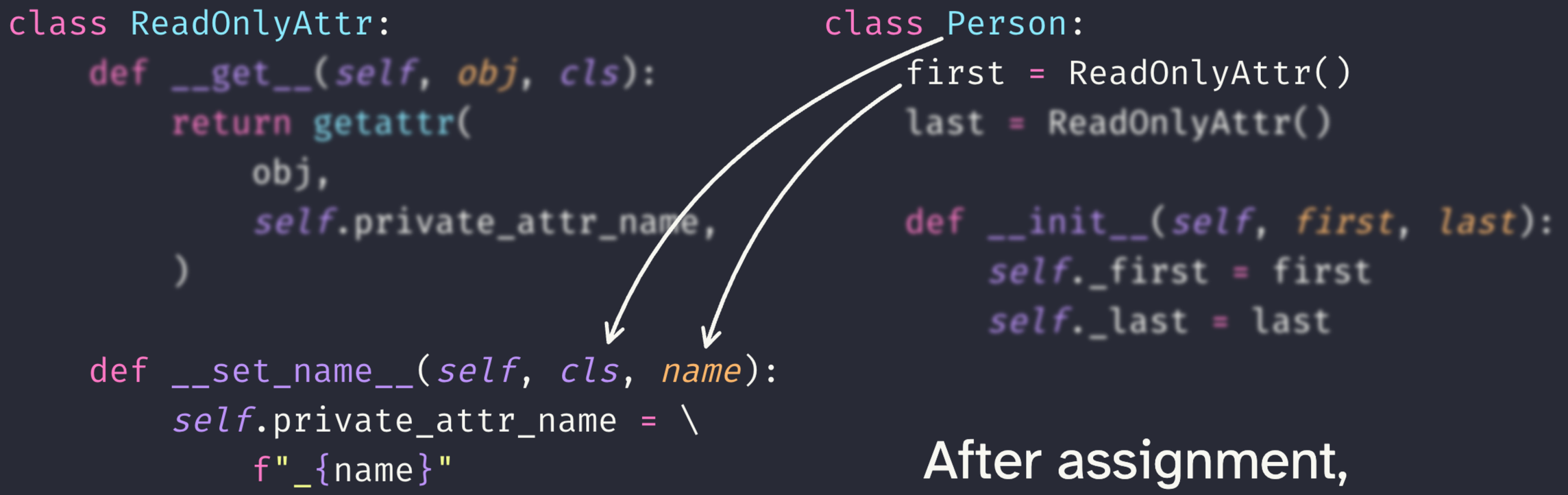
`self` is the descriptor,

not the person.

```python
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}"
```

```python
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.



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!


Hell no!

I showed you enough
to get you started.

Challenges

Challenges:

- add `__set__` to `RGBComponent`

Challenges:

- add `__set__` to `RGBComponent`
- add `__set__` to `ReadOnlyAttr` (make it error)

Challenges:

- add `__set__` to `RGBComponent`
- add `__set__` to `ReadOnlyAttr` (make it error)
- create `LogAttrAccess` descriptor

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

mathspp.com/pydonts

presented with snappify*

*no affiliation, they're just awesome

rodrigo@mathspp.com