Classes

Part 3 – Data Encapsulation

by

Lilian Blot

Class methods have only one specific difference from ordinary functions, an extra parameter at the beginning of the parameter list

but we do **not** give a value for this parameter when we call the method.

this particular variable **refers** to the object **itself**,

by convention, it is given the name **`self`**.

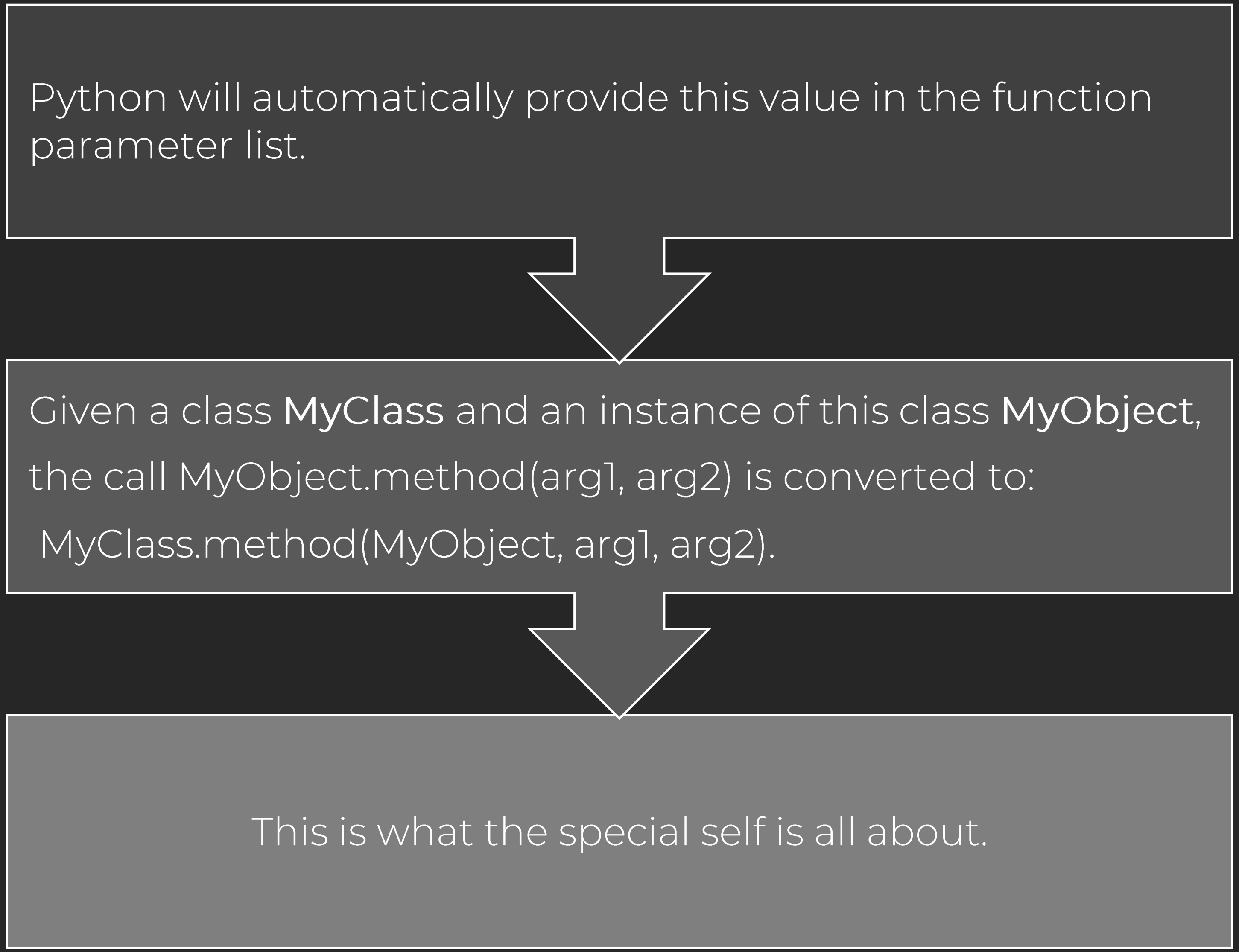
Although, we can give any name for this parameter, it is ***strongly*** *recommended* that we use the name **`self`**.

Any other name is definitely **frowned upon**.

There are many advantages to using a standard name

any reader of your program will immediately recognize that it is the object variable, some IDE automatically add the parameter.

The self parameter



Python will automatically provide this value in the function parameter list.

Given a class **MyClass** and an instance of this class **MyObject**, the call MyObject.method(arg1, arg2) is converted to:

MyClass.method(MyObject, arg1, arg2).

This is what the special self is all about.

Encapsulation as Information hiding

Data belonging to one object is hidden from other objects.

Know what an object can do, not how it does it.

Information hiding increases the level of *independence*.

Independence of modules is important for large systems and maintenance.

Information Hiding

- looking back at our first attempt, all Attributes are public

Code

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return ('<'+ str(self.x) + ', ' + self.y + ‘>’)
```

No Information Hiding

interpreter

```
>>> p = Point(1,2)
>>> p.x
1
>>> p.x = '10’
>>> p
'10'
```

Protected vs Public Attributes

“protected” instance variables that cannot be accessed except from inside an object don’t exist in Python.

there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member).

Such attribute, function or method should be considered an implementation detail and subject to change without notice.

There is no completely private attributes in Python, however there is limited support for such a mechanism, called *name mangling*.

Information Hiding

- Adding data encapsulation to our first attempt

Code

```
class Point:

    def __init__(self, x=0, y=0):
        self._x = x
        self._y = y

    def __str__(self):
        return ('<'+ str(self._x) + ', ' + self._y + ‘>’)
```

The two instance variables _x and _y are by convention **protected**.

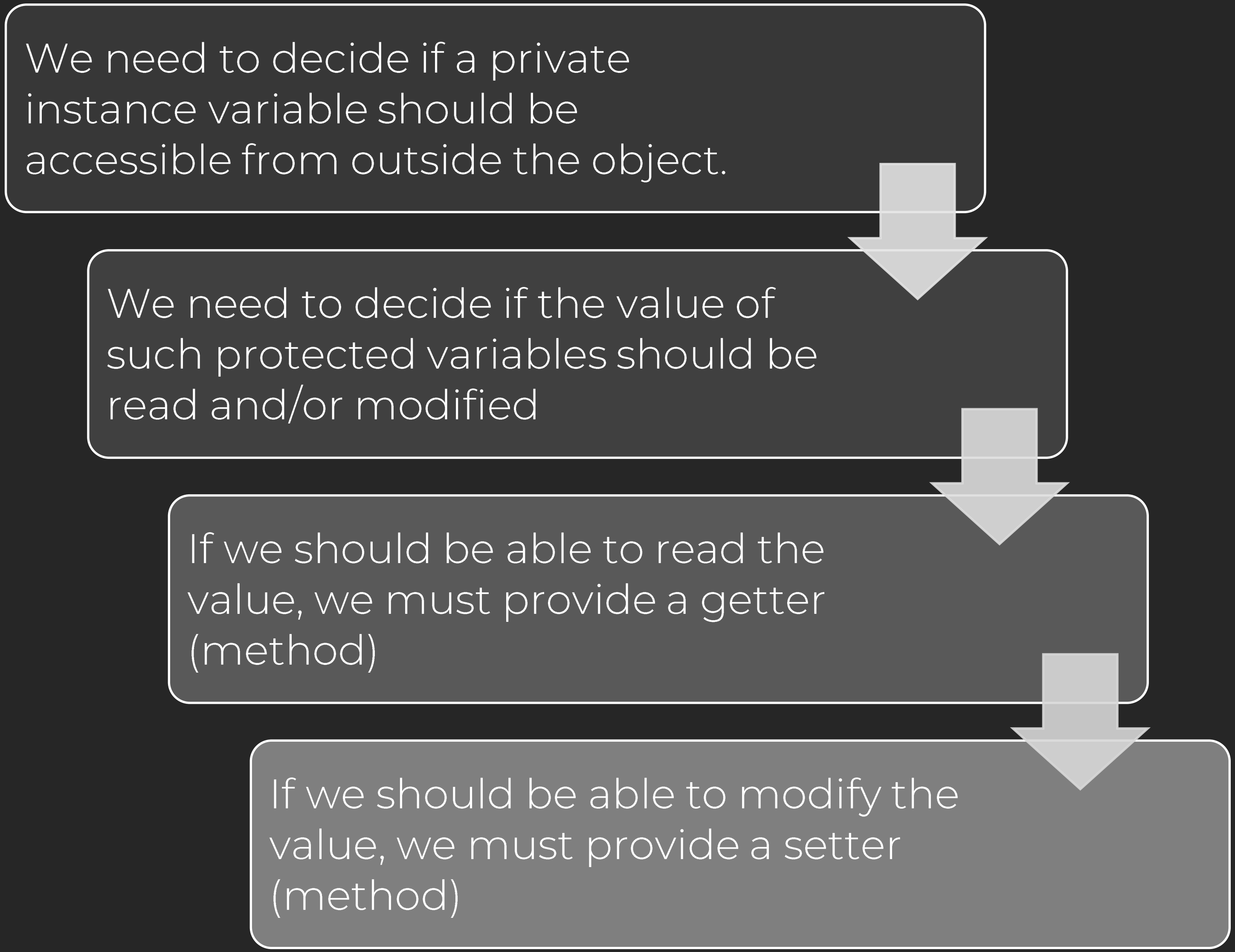
- Despite the convention, both instance variables are not strictly protected.

interpreter

```
>>> p = Point(1,2)
>>> p._x
1
>>> p._x = '10’
>>> p
'10'
```

The instance variable _x can be accessed and modified (not strictly protected then).

Accessing "Protected" instance variables



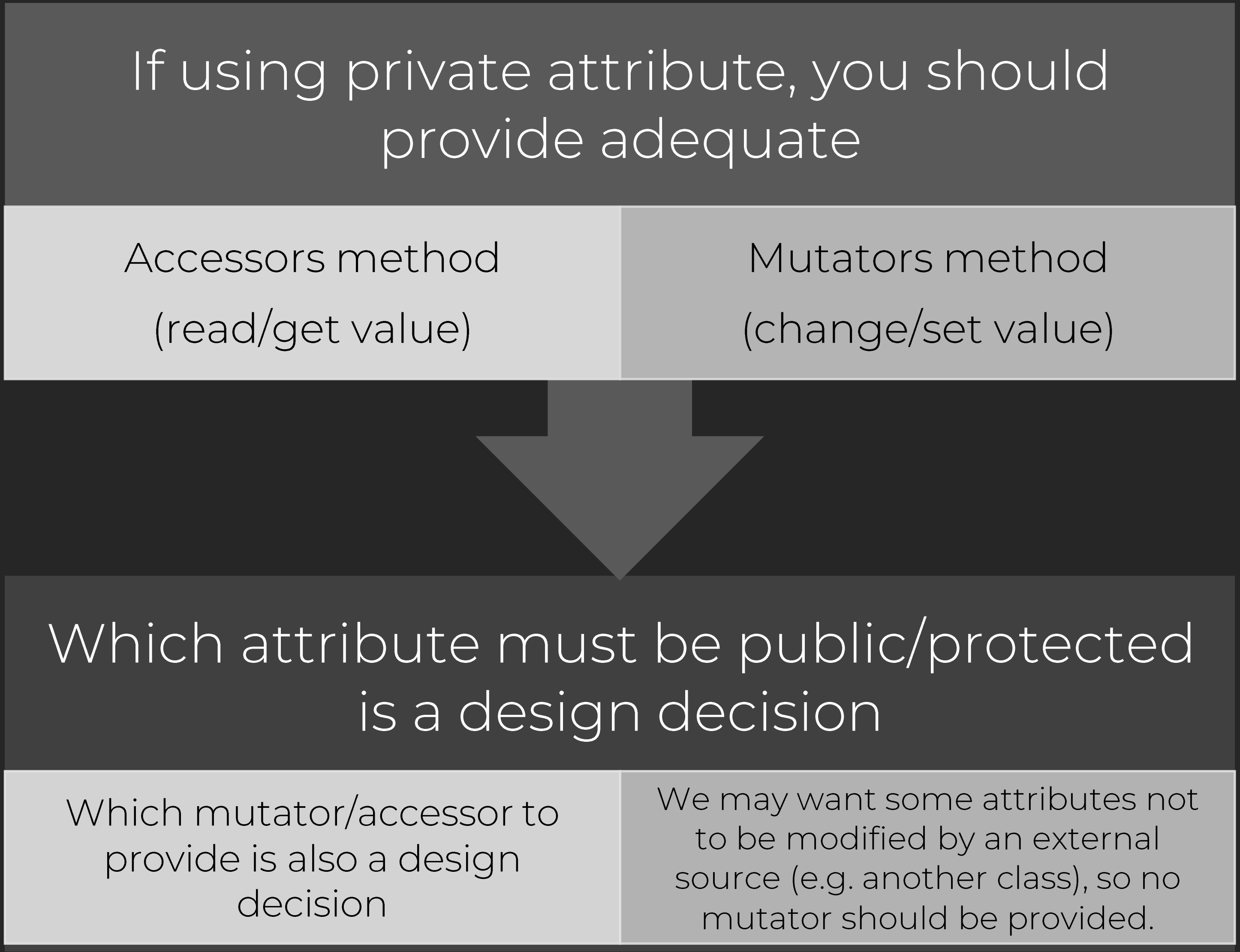
We need to decide if a private instance variable should be accessible from outside the object.

We need to decide if the value of such protected variables should be read and/or modified

If we should be able to read the value, we must provide a getter (method)

If we should be able to modify the value, we must provide a setter (method)

Protected vs Public Attributes



If using private attribute, you should provide adequate

Accessors method (read/get value)

Mutators method (change/set value)

Which attribute must be public/protected is a design decision

Which mutator/accessor to provide is also a design decision

We may want some attributes not to be modified by an external source (e.g. another class), so no mutator should be provided.

Information Hiding

adding data encapsulation to our first attempt (the Python way)

Code

```python
class Point:

    def __init__(self, x=0, y=0):
        self._x = x
        self._y = y

    @property
    def x(self): # this is the getter
        return self._x

    @x.setter
    def x(self, value): # this is the setter
        if isinstance(value, str):
            self._x = float(value)
        elif isinstance(value, float) or isinstance(value, int):
            self._x = float(value)
        else:
            raise TypeError(…)
```

Note that the name of the two methods are the same. It is the name of the property x.

The decorator @property is used to define a getter for property named x.

The decorator @x.setter is used to define a setter for the property named x.