Introduction to Object-Oriented Programming

Table of contents

See csse1001[1] for course logistics — this note covers Lecture 7C's technical content. See python-classes-and-objects[2] for the full reference on classes, instantiation, and encapsulation.

Today's outline

- Programming paradigms: imperative vs. declarative
- Objects, classes, and `self`
- Instantiation, attributes, equality, and aliasing
- Methods
- Private variables, getters, and setters

Programming paradigms

Imperative programming — the programmer says *how* to do something, by:

1. **Procedural** — grouping instructions into functions.

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-classes-and-objects.pdf

2. **Object-oriented** — grouping instructions into objects that combine data (state, attributes) and behaviour (methods).

(*Imperative*, the adjective, means giving an authoritative command.)

Declarative programming — the programmer says *what* they want, through:

1. **Functional** — a series of function applications.
2. **Logic** — a question about a system of facts and rules.
3. **Mathematical** — optimization.

Object-oriented programming

The fundamental building block of object-oriented programming is the **class** (or **object**). The design principle is to solve a problem by creating objects that interact with one another.

An **object** is a collection of fields/attributes (data comprising the object's state) along with methods (class-scoped functions) that can act on the object itself. An object can reference and change its own state, and has a notion of **self**.

Real-world analogues:

Class	Attributes	Methods
`Dog`	`breed`, `size`, `age`, `colour`	`eat()`, `bark()`, `sleep()`, `fetch()`
`Student`	`name`, `age`, `grades`, `major`	`take_exam()`, `enroll()`, `attend_class()`
`Smartphone`	`brand`, `battery_level`, `is_on`, `storage`	`turn_on()`, `turn_off()`, `make_call()`, `install_app()`

Convention: class names in Python are in `CamelCaps` — `ClassNamesAreLikeThis`.

I: Structure (holding named attributes)

```
class Point():
    def __init__(self):
        self.x = 0
        self.y = 0
```

```
>>> p = Point()          # 'instantiation' of the Point object
>>> p
<__main__.Point object at 0x10a9e0dd8>
>>> type(p)
<class '__main__.Point'>
>>> p.x
0
>>> p.y
0
```

Attributes of an instance are accessed with `.` — these are called **instance variables**.

What is `self`?

Think of the class definition as a *blueprint* with placeholders. `self` has an `x`, `self` has a `y`, and so on — these are the placeholders. When you create an *instance* of the class (e.g. `p`), `self` becomes the actual object you're working with, and each placeholder becomes a real attribute stored in that object. You can create many instances from the same blueprint, each with its own unique values.

```
>>> p = Point()
>>> p.x = 2
>>> p.y = 3
>>> p.x
2
>>> p.y
3
```

Initializing with input

```
class Point():
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

>>> p = Point(2, 3)   # creates a Point and passes 2, 3 to __init__
>>> p.x
2
>>> p.y
3
```

Careful! Don't pass an argument for `self` — Python supplies it automatically.

Equality

```
>>> p = Point(2, 3)
>>> q = Point(2, 3)
>>> p == q
False
```

Two separately-constructed objects with identical attribute values are **not** `==` by default — equality (as opposed to identity) needs to be defined explicitly (covered in a later lecture on magic methods).

Aliasing

```
>>> p = Point(2, 3)
>>> q = p            # q is an alias for the same object as p
>>> q.x = 1
>>> p.x
1
>>> p == q
True
```

Because `q` and `p` are aliases pointing to the *same* object, mutating `q` also changes what `p` sees, and (since it's literally the same object) `p == q` is `True` here.

II: Methods (object-scoped functions)

```
class Person():
    def __init__(self, name: str) -> None:
        self.name = name

    def foo(self) -> str:            # methods must take self as the first argument
        return f"My name is {self.name}."
```

```
>>> p = Person("Slim Shady")
>>> p.foo()
'My name is Slim Shady.'

>>> foo()
NameError: name 'foo' is not defined
```

A method must be accessed via the object using dot notation: `<object_variable>.<method>(<parameters>)`.

```
>>> p = Person("What")
>>> q = Person("Who")
>>> r = Person(f"{2*'chka'} Slim Shady")
>>> p.foo()
'My name is What.'
>>> q.foo()
'My name is Who.'
>>> r.foo()
'My name is chka chka Slim Shady.'
```

Each instance keeps its own independent state.

Exercise: a `Counter` object

Often our objects will be analogous to things that exist in the real world. Create an object called `Counter` that simulates the functionality of a hand-tally counter.

```
class Counter():
    def __init__(self) -> None:
        self._value = 0            # a 'private' variable

    def get_value(self) -> int:    # a 'getter'
        return self._value

    def click(self) -> None:
        self._value = self._value + 1

    def reset(self) -> None:
        self._value = 0
```

```
>>> x = Counter()
>>> x.get_value()
0
>>> x.click()
>>> x.click()
>>> x.click()
>>> x.get_value()
3
>>> x.reset()
```

```
>>> x.get_value()
0
```

Separate instances track their own count independently:

```
>>> x = Counter()
>>> y = Counter()
>>> x.click()
>>> y.click()
>>> x.click()
>>> x.get_value()
2
>>> x.click()
>>> y.get_value()
1
```

Private variables

The leading underscore on `_value` in `Counter` signals that this name is **private** — programmers should never manipulate it directly from outside the object.

Warning: nothing actually *prevents* a user from accessing a private variable in Python:

```
>>> x = Counter()
>>> x.click()
>>> x._value = -10
>>> x.click()
>>> x.get_value()
-9
```

Accessing private variables directly is bad practice.

Setters

It's good practice to use a method — a **setter** — for changing an object's private variables at the user level, so that invalid values can be rejected:

```
class Counter():
    def __init__(self) -> None:
        self._value = 0

    def set_value(self, x: int) -> None:   # a 'setter'
        if x < 0:
            raise ValueError
        self._value = x
```

Classic getters and setters

```
class Person():
    def __init__(self, name):
        self._name = name   # leading underscore = "internal use"

    def get_name(self):
        return self._name

    def set_name(self, value):
        if not value:
            raise ValueError("Name cannot be empty")
        self._name = value

p = Person("Alice")
print(p.get_name())
p.set_name("Sara")
print(p.get_name())
```

Pythonic getters and setters

Python's `@property` decorator lets a getter/setter pair be used with plain attribute-access syntax, rather than explicit `get_`/`set_` method calls:

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

    @property
    def name(self):             # getter
        return self._name
```

```
    @name.setter
    def name(self, value):      # setter
        if not value:
            raise ValueError("Name cannot be empty")
        self._name = value

p = Person("Alice")
print(p.name)         # looks like attribute access (calls the getter)
p.name = "Sara"        # looks like assignment (calls the setter)
print(p.name)
```

Summary

OOP helps organise code using classes and objects.

- **Class**: a blueprint for creating objects.
- **Object**: an instance of a class.
- **Encapsulation**: keep data safe inside classes using attributes and methods.

Next: magic methods.