Introduction to Object-Oriented Programming

Table of contents

Outline 1
Programming paradigms 1
Object-oriented programming 2
I: Structure (holding named attributes) 2
II: Methods (object-scoped functions) 5
Exercise: a `Counter` object 6
Private variables 7
Setters 8
Summary 9

Lecture 7C. Course logistics are in csse1001[1]. The full reference on classes, instantiation, and encapsulation is python-classes-and-objects[2].

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* what to do, 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 *declares* 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.

In one line: **imperative** is *how* to do it, **declarative** is *what* to do. It is climbing rung by rung to the Moon versus simply floating there.

See programming-paradigms[3] for the full reference, including where each paradigm shows up across this course.

Object-oriented programming

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

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

Real-world analogues are in Figure 1. The split into a data half and a behaviour half is the whole idea.

Convention: names of classes (i.e. objects) in Python are in `CamelCaps`. That is, `ClassNamesAreLikeThis`.

I: Structure (holding named attributes)

```
class Point():                  # Point() is the class name
    def __init__(self):         # the initializer method
        self.x = 0              # -.
        self.y = 0              # -'  attributes
```

[3] courses/csse1001/programming-paradigms.pdf

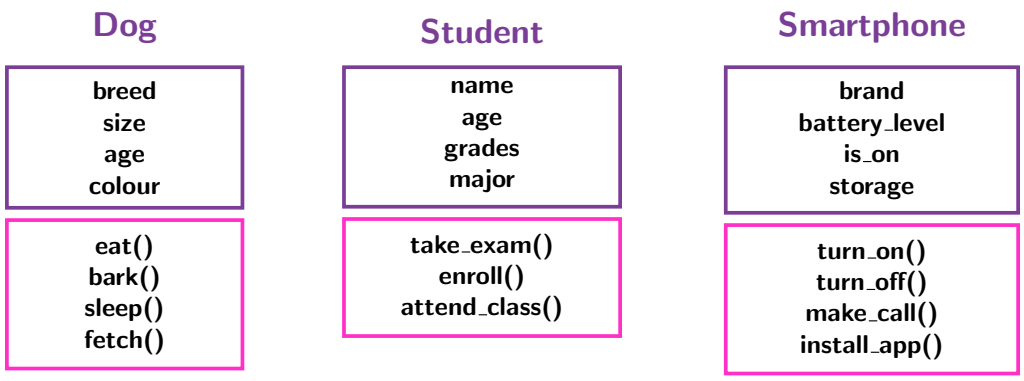

Figure 1: Three real-world classes. Each class is a name above two stacked boxes: the attributes (what it knows) on top and the methods (what it does) beneath. The attributes box is outlined in the dark purple used for class names and the methods box in bright magenta, so the two halves are told apart by outline colour, not just by position.

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

`Point()` is the class name, `def __init__(self):` is the initializer method, and the two `self.` lines are the attributes. `p.x` accesses an attribute of an instance with `.`; these are called instance variables.

What is `self`?

Think of the class definition as a **blueprint** with placeholders or actual items. `self` has an `address`, `self` has `num_bedrooms`, `self` has `num_bathrooms`, `self` has a `garage`, `self` has `has_solar_panels` — these are the placeholders. When you create an *instance* of that class (e.g. `beach_house`), `self` becomes the actual object you're working with, and each of these becomes a real attribute stored in the object when it's created. You can create many instances using the same blueprint, each with its own unique values.

(The analogy is inspired by an example from MIT6.100L.)

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

Initializing with input

Instantiating an object with input instead:

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

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

Do not pass an argument for `self` — Python handles it automatically.

Equality and aliasing

Two separately-constructed objects with identical attribute values are **not** `==` by default; two *names for the same object* are. The difference is a difference in the picture of memory, not in the code that looks at it — see Figure 2.

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

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

```
1
>>> p == q           # because these aliases point to the same object
True
```

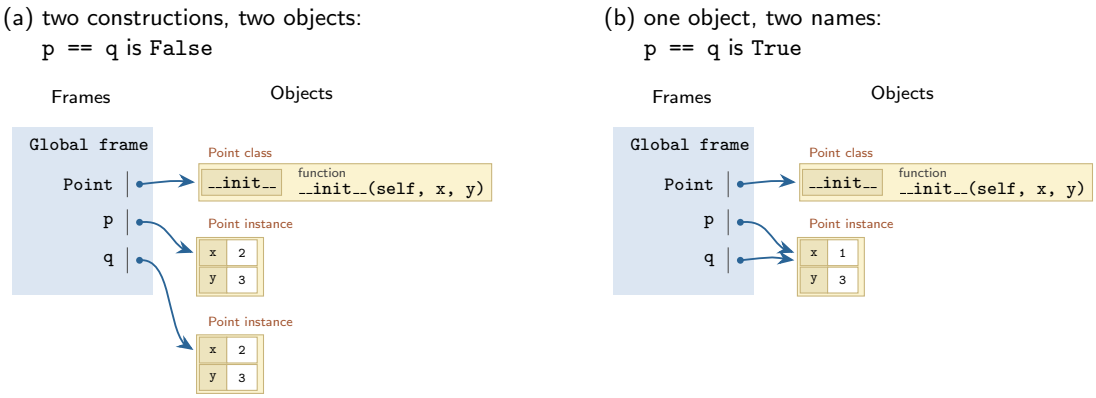

Figure 2: Equality and aliasing side by side, in Python Tutor notation. In (a) q = Point(2, 3) builds a second object, so p and q arrow off to two separate Point instances that happen to hold equal values, and p == q is False. In (b) q = p binds a second name to the one object, so both arrows converge on it — that convergence is the whole notation for aliasing — and mutating q.x is visible as p.x, and p == q is True. Both panels also show Point itself bound in the Global frame to the class object holding **init**, which is what makes Point(2, 3) callable at all.

Equality (as opposed to identity) has to be defined explicitly — that is what `__eq__` is for, and it comes in the next lecture, 2025-09-16-dunder-methods[4].

II: Methods (object-scoped functions)

```
class Person():
    def __init__(self, name: str) -> None:   # a method for initializing
        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.'
```

[4] courses/csse1001/2025-09-16-dunder-methods.pdf

```
>>> foo()                # you must access foo via the object
NameError: name 'foo' is not defined
```

A method is reached by **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.

> `f"{2*'chka'} Slim Shady"` actually evaluates to `"chkachka Slim Shady"` — for the output above the repeated string would need its own trailing space, `f"{2*'chka '}Slim Shady"`.

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-held tally counter.

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

    def get_value(self) -> int:    # this is called 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 — each `Counter()` call builds its own object with its own `_value`, exactly as the two `Point` instances in Figure 2 (a) do:

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

Private variables

The underscore on `_value` in the `Counter` object is used to denote this name as **private**. This signals that programmers should *never* manipulate this value outside the object.

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

```
>>> x = Counter()
>>> x.get_value()
0
>>> x.click()
>>> x._value = -10     # accessing private variables is bad practice
>>> x.click()
>>> x.get_value()
-9
```

Setters

It is good practice to use *methods* for changing an object's private variables at user level. We usually enable this through a `setter` method which *sets* a private value:

```
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
```

The `raise` is the point of routing writes through a method: it is the only place the class gets to reject a value before it becomes state. (See python-exceptions[5].)

Classic getters and setters

```
# Classic implementation of 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())
```

[5] courses/csse1001/python-exceptions.pdf

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 — the validation still runs, but the call sites read like ordinary attribute reads and writes:

```
# Pythonic implementation of getters and setters
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)          # Sara
```

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 — 2025-09-16-dunder-methods[6].

[6] courses/csse1001/2025-09-16-dunder-methods.pdf