CLASSES

Classes are at the heart of object-oriented programming. They allow you to store both information and actions to perform on that information in one place. Understanding classes will allow you to write code that models just about anything in the real world.

ABOUT CLASSES

- What is a class?
- What is an attribute?
- What is a method?
- How do you define a class?

A *class* organizes code that combines data and actions. To use a class, you make an *instance* of the class. An *attribute* is a variable associated with a class, and a *method* is a function associated with a class.

This class represents a trail, such as a hiking trail, with two attributes, `self.dest` and `self.len`, and one method, `__init__()`:

```python
class Trail():
    """A class to represent trails."""

    def __init__(self, dest, len=0):
        self.dest = dest
        self.len = len
```

Python classes generally use initial capital letters. When an instance of a class is made, the `__init__()` method runs automatically. In this code, `__init__()` requires one argument, `dest`, and has one optional parameter, `len`.

The first parameter, `self`, references the current instance of the class. Attaching the values `dest` and `len` to `self` makes them available to all methods in the class and any instance of the class.

METHODS

- How do you define a method?
- How do you make an instance of a class?

After defining a class, you can add methods that allow you to take actions using the class.

This method generates a description of a trail and prints that description:

```python
class Trail():
    """A class to represent trails."""

    def __init__(self, dest, len=0):
        self.dest = dest
        self.len = len

    def describe_trail(self):
        """Print a description of trail."""
        desc = f"This trail goes to {self.dest}."
        if self.len:
            desc += f"\nThe trail is {self.len}km."
        print(desc)
```

Each method is automatically sent a reference to the instance of the class, so each method needs a `self` parameter. Then the method can use any attribute from the class. In this code, the description includes the trail's destination and its length if it's specified.

MAKING INSTANCES

- How do you make an instance of a class?
- How do you access the value of an attribute through an instance?
- How do you use an instance to call methods?

To use class attributes and methods, you must make instances, or *objects*, from the class. The class defines a set of values and actions that represents any trail. An *instance* is a specific set of values that represents a specific trail.

Here we make an instance of `Trail` and call the method `describe_trail()`:

```
verst = Trail("Mt. Verstovia", 4)
print(f"Destination: {verst.dest}")
verst.describe_trail()
```

You provide the class name and include any arguments the `__init__()` method requires. You assign the instance to a variable, and then you can access attributes and call methods using dot notation.

Here's the output:

```
Destination: Mt. Verstovia
This trail goes to Mt. Verstovia.
The trail is 4km.
```

ADDING METHODS

- How do you define a class with multiple methods?

A class can have as many methods as it needs.

Here's `Trail` with a second method defined:

```
class Trail():
    """A class to represent trails."""
        --snip--

    def run_trail(self):
        """Simulate running the trail."""
        print(f"Running to {self.dest}.")
```

Each method has the `self` parameter, giving it access to all the attributes in the class.

Here we make an instance representing a specific trail and call the two methods:

```
verst = Trail("Mt. Verstovia", 4)
verst.describe_trail()
verst.run_trail()
```

Here's the output:

```
This trail goes to Mt. Verstovia.
The trail is 4km.
Running to Mt. Verstovia.
```

MULTIPLE INSTANCES

- How do you make multiple instances from a class?

You can make as many instances as you need from a class.

Here, the same class generates two trails:

```
verst = Trail("Mt. Verstovia", 4)
verst.describe_trail()
verst.run_trail()

ms = Trail("Middle Sister", 10)
ms.describe_trail()
ms.run_trail()
```

Here's the output:

```
This trail goes to Mt. Verstovia.
The trail is 4km.
Running to Mt. Verstovia.
This trail goes to Middle Sister.
The trail is 10km.
Running to Middle Sister.
```

Each instance maintains its own set of attributes.

INHERITANCE

- What is inheritance?
- What is a parent class?
- What is a child class?
- How do you use inheritance?

By using *inheritance*, you can create a new class that inherits the attributes and methods of the original class.

We'll create a class to represent a bike trail: the *parent class* is `Trail`, and the *child class* is `BikeTrail`.

You define a child class like a normal class but include the parent class name in parentheses:

```
class BikeTrail(Trail):
    """Represent a bike trail."""

    def __init__(self, dest, len=0):
        super().__init__(dest, len)
        self.paved = True
        self.bikes_only = True
```

The child class `__init__()` method often needs to call the parent class `__init__()` method by using `super()`. The `super()` function references the parent class and allows you to call methods from the parent class.

The child class can define additional attributes particular to itself. For example the `self.bikes_only` attribute indicates whether the trail is closed to nonbicyclists.

CHILD CLASS METHODS

- How do you write a new method for a child class?

A child class can define its own methods, giving it specific behavior the parent class lacks.

The new BikeTrail method, `ride_trail()`, prints a message that's only appropriate for a bike trail:

```
class BikeTrail(Trail):
    """Represent a bike trail."""

    def __init__(self, dest, len=0):
        --snip--

    def ride_trail(self):
        """Simulate riding the trail."""
        print(f"Riding to {self.dest}.")

cross_trail = BikeTrail("Harbor Mountain", 5)
cross_trail.ride_trail()
```

Here's the output:

```
Riding to Harbor Mountain.
```

OVERRIDING PARENT CLASS METHODS

- How does a child class modify the behavior of a method from the parent class?

A child class can customize the behavior of parent class methods by using super() to access those methods.

This `BikeTrail` class modifies the behavior of the method `run_trail()`:

```
class BikeTrail(Trail):
    """Represent a bike trail."""

    def __init__(self, dest, len=0):
        --snip--

    def run_trail(self):
        """Simulate running the trail."""
        if self.bikes_only:
            print("You can't run this trail!")
        else:
            super().run_trail()

cross_trail = BikeTrail("Harbor Mountain", 5)
cross_trail.run_trail()
```

The method first checks if this is a bikes-only trail; if it is, an appropriate message prints:

```
You can't run this trail!
```

If the trail allows runners, the method calls the parent method `run_trail()`, inheriting the behavior of a regular `Trail`.

STORING CLASSES IN MODULES

- How do you store a class in a module?
- How do you import a class from a module?
- How do you use an imported class?

You store classes in modules and import them the same way as functions.

Here we store the classes `Trail` and `BikeTrail` in the *trails.py* file:

```
class Trail():
    """A class to represent trails."""
    --snip--

class BikeTrail(Trail):
    """Represent a bike trail."""
    --snip--
```

Now we can import these classes into another file:

```
from trails import Trail, BikeTrail

verst = Trail("Mt. Verstovia", 4)
cross_trail = BikeTrail("Harbor Mt", 5)
```

Use dot notation to import the entire module:

```
import trails

verst = trails.Trail("Mt. Verstovia", 4)
cross_trail = trails.BikeTrail("Harbor Mt", 5)
```

Use aliases when importing classes. Store classes to organize your code and make the classes available to multiple programs.