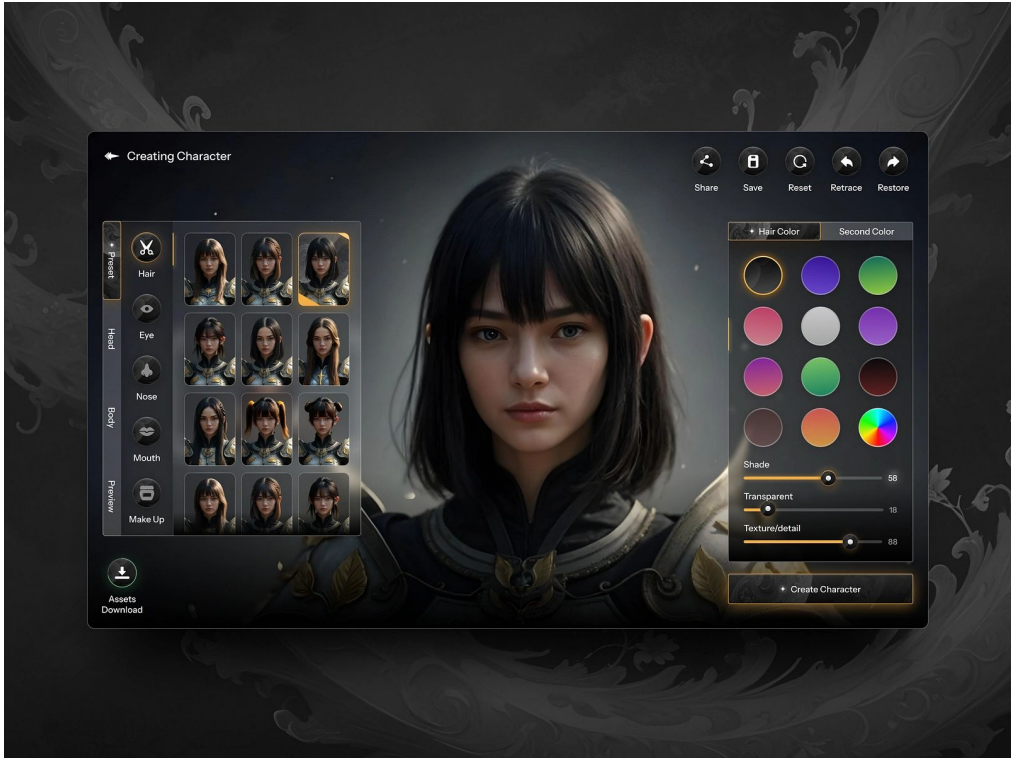
Programming Fundamentals

With Python

Chapter 6

01

Classes and Objects



Introduction

```python
class Person():
    def __init__(self,name, height, age, mass):
        self.name = name
        self.height = height
        self.age = age
        self.mass = mass

    def calculateBMI(self):
        return self.mass/(self.height * self.height)

p1 = Person("Olivia",1.71,28,63)
p2 = Person("Liam",1.83,35,77)

print(f"{p1.name} is {p1.age} years old and has a BMI of {p1.calculateBMI()} kg/m^2")
print(f"{p2.name} is {p2.age} years old and has a BMI of {p2.calculateBMI()} kg/m^2")
```

This is a constructor. This is basically a special method that is called every time you create a new object or instance of the class. It allows you to define the initial values of a instance of a class or object.

This is an instance variable. These are variables that describe the class.

This is a class method that describes what actions a class can take. It changes the data for a specific instance of the class.

This is an object. An object is just 1 instance of a class.

Output

```
Olivia is 28 years old and has a BMI of 21.545090797168363 kg/m^2
Liam is 35 years old and has a BMI of 22.99262444384723 kg/m^2
```

Introduction

```python
class Person():
    def __init__(self,name, height, age, mass):
        self.name = name
        self.height = height
        self.age = age
        self.mass = mass

    def calculateBMI(self):
        return self.mass/(self.height * self.height)

p1 = Person("Olivia",1.71,28,63)
p2 = Person("Liam",1.83,35,77)

print(f"{p1.name} is {p1.age} years old and has a BMI of {p1.calculateBMI()} kg/m^2")
print(f"{p2.name} is {p2.age} years old and has a BMI of {p2.calculateBMI()} kg/m^2")
```

Output

```
Olivia is 28 years old and has a BMI of 21.545090797168363 kg/m^2
Liam is 35 years old and has a BMI of 22.99262444384723 kg/m^2
```

In General

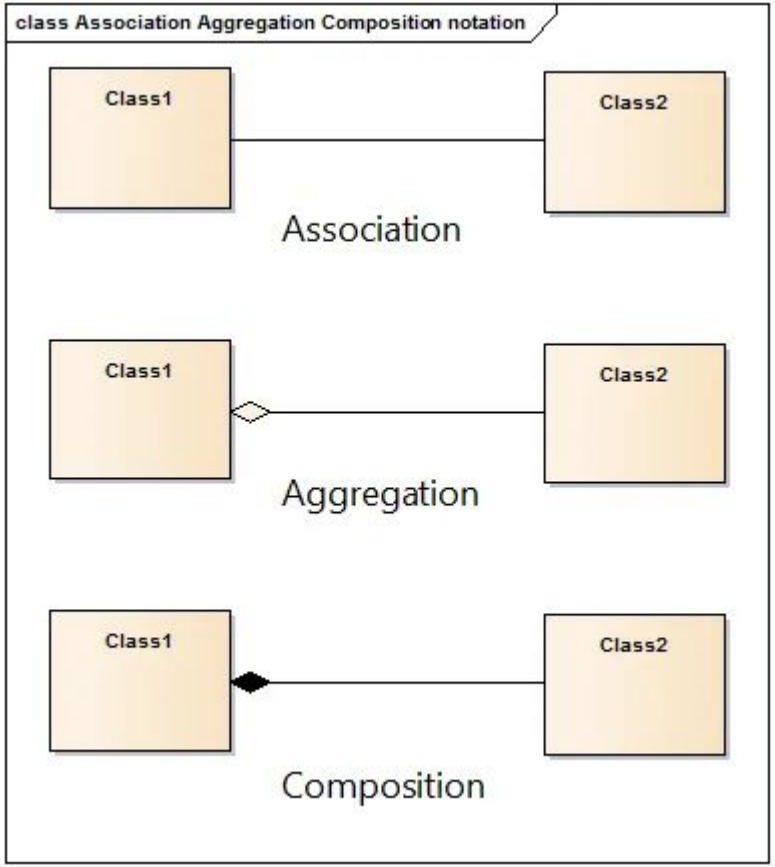
- A class is like a noun (person, place, or thing).
- An instance variable is like an adjective.
- A class method is like a verb.

When should classes be used?

Anytime you have an entity in your program that can be described with data and you need to have that entity’s data change, you would use a class for that.

02

Composition and Aggregation



What is Composition?

Composition is when an instance variable of a class is an object type. This means it is assigned to an instance of another class or its own class.

Composition Example

```
1  class Song():
2      def __init__(self,name,singer):
3          self.name = name
4          self.singer = singer
5          self.next = None
6
7      def setNext(self,next):
8          self.next = next
9
10 s1 = Song("Shape of You","Ed Sheeran")
11 s2 = Song("Rolling in the Deep","Adele")
12 s3 = Song("Levitating","Dua Lipa")
13
14 s1.setNext(s2)
15 s2.setNext(s3)
16 s3.setNext(None)
17
18 print("Here is your playlist!")
19 current = s1
20 i = 0
21
22 while current != None:
23     print(f"{i + 1}. {current.name} by {current.singer}")
24     current = current.next
25     i += 1
```

Output

```
Here is your playlist!
1. Shape of You by Ed Sheeran
2. Rolling in the Deep by Adele
3. Levitating by Dua Lipa
```

Objects are References

```
1  class Song():
2      def __init__(self,name,singer):
3          self.name = name
4          self.singer = singer
5          self.next = None
6
7      def setNext(self,next):
8          self.next = next
9
10 s1 = Song("Shape of You","Ed Sheeran")
11 s2 = Song("Rolling in the Deep","Adele")
12 s3 = Song("Levitating","Dua Lipa")
13
14 s1.setNext(s2)
15 s2.setNext(s3)
16 s3.setNext(None)
17
18 print("Here is your playlist!")
19 current = s1
20 i = 0
21
22 while current != None:
23     print(f"{i + 1}. {current.name} by {current.singer}")
24     current = current.next
25     i += 1
```

Current → 9807

s1.next → 4098

s2.next → 3029

Address	Object
9807	s1
4098	s2
3029	s3

Here is basically what you did:
s1.setNext(4098)
s2.setNext(3029)
s3.setNext(None)

Current = 9807

Objects are References

```
1  class Song():
2      def __init__(self,name,singer):
3          self.name = name
4          self.singer = singer
5          self.next = None
6
7      def setNext(self,next):
8          self.next = next
9
10 s1 = Song("Shape of You","Ed Sheeran")
11 s2 = Song("Rolling in the Deep","Adele")
12 s3 = Song("Levitating","Dua Lipa")
13
14 s1.setNext(s2)
15 s2.setNext(s3)
16 s3.setNext(None)
17
18 print("Here is your playlist!")
19 current = s1
20 i = 0
21
22 while current != None:
23     print(f"{i + 1}. {current.name} by {current.singer}")
24     current = current.next
25     i += 1
```

Current → s1.next → 4098

s2.next → 3029

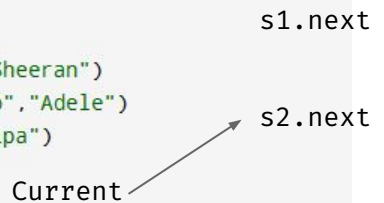
Address	Object
9807	s1
4098	s2
3029	s3

Here is basically what you did:
s1.setNext(4098)
s2.setNext(3029)
s3.setNext(None)

Current = 4098

Objects are References

```
1  class Song():
2      def __init__(self,name,singer):
3          self.name = name
4          self.singer = singer
5          self.next = None
6
7      def setNext(self,next):
8          self.next = next
9
10 s1 = Song("Shape of You","Ed Sheeran")
11 s2 = Song("Rolling in the Deep","Adele")
12 s3 = Song("Levitating","Dua Lipa")
13
14 s1.setNext(s2)
15 s2.setNext(s3)
16 s3.setNext(None)
17
18 print("Here is your playlist!")
19 current = s1
20 i = 0
21
22 while current != None:
23     print(f"{i + 1}. {current.name} by {current.singer}")
24     current = current.next
25     i += 1
```



Address	Object
9807	s1
4098	s2
3029	s3

Here is basically what you did:
s1.setNext(4098)
s2.setNext(3029)
s3.setNext(None)

Current = 3029

Objects are References

```
1  class Song():
2      def __init__(self,name,singer):
3          self.name = name
4          self.singer = singer
5          self.next = None
6
7      def setNext(self,next):
8          self.next = next
9
10 s1 = Song("Shape of You","Ed Sheeran")
11 s2 = Song("Rolling in the Deep","Adele")
12 s3 = Song("Levitating","Dua Lipa")
13
14 s1.setNext(s2)
15 s2.setNext(s3)
16 s3.setNext(None)
17
18 print("Here is your playlist!")
19 current = s1
20 i = 0
21
22 while current != None:
23     print(f"{i + 1}. {current.name} by {current.singer}")
24     current = current.next
25     i += 1
```

Address	Object
9807	s1
4098	s2
3029	s3

s1.next → 4098

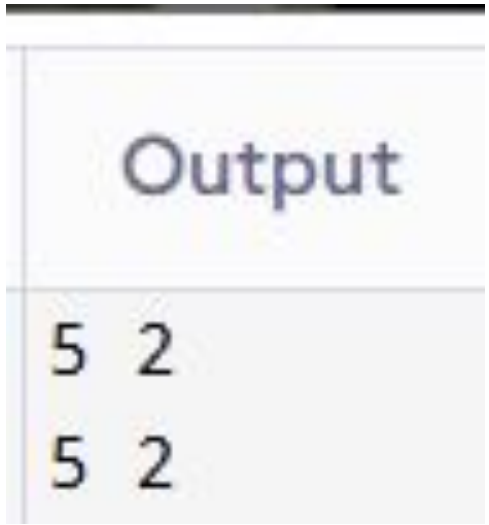
s2.next → 3029

Here is basically what you did:
s1.setNext(4098)
s2.setNext(3029)
s3.setNext(None)

Current = None

Objects are References

```
1  class Point():
2      def __init__(self,x,y):
3          self.x = x
4          self.y = y
5
6  p1 = Point(1,2)
7  p2 = p1
8
9  p2.x = 5
10
11 print(p1.x,p1.y)
12 print(p2.x,p2.y)
```



Both reference the same object.

Objects are References

```
1  import copy
2
3  class Point():
4      def __init__(self,x,y):
5          self.x = x
6          self.y = y
7
8  p1 = Point(1,2)
9
10 p2 = copy.deepcopy(p1)
11
12 p2.x = 5
13
14 print(p1.x,p1.y)
15 print(p2.x,p2.y)
```

Output

```
1 2
5 2
```

Objects are References

```
1  class Point():
2      def __init__(self,x,y):
3          self.x = x
4          self.y = y
5
6  p1 = Point(1,2)
7  p2 = Point(2,3)
8  p3 = Point(2,3)
9  p4 = Point(3,5)
10
11 points = [p1,p2,p4]
12
13 print(p1 in points)
14 print(p3 in points)
```

Output

```
True
False
```

Again, objects are just references. So, think of each object as just an address somewhere different in memory. So, even though p2 and p3 have the same values, they have different addresses. So, the address of p3 was not found in points because p3 was not in points.

What is Aggregation?

Aggregation is when an instance variable of a class is of type list and all the elements in that list are an object type.

Aggregation Example

```python
class School():
    def __init__(self, name):
        self.name = name
        self.students = []

    def addStudents(self,student):
        self.students.append(student)

class Student():
    def __init__(self,name):
        self.name = name
        self.marks = []

    def enterMark(self,mark):
        self.marks.append(mark)

    def calculateAVG(self):
        sum = 0
        for mark in self.marks:
            sum += mark
        return sum/len(self.marks)
```

```python
# Imagine a new school was created
school = School("St. Joan of Arc")

# This school got its first 3 students
s1 = Student("Joseph")
s2 = Student("Mark")
s3 = Student("David")

# Adding the students into their new school
school.addStudents(s1)
school.addStudents(s2)
school.addStudents(s3)

# The 3 students finished their first year and completed 3 classes each.
# Their marks need to be entered

school.students[0].enterMark(0.7)
school.students[0].enterMark(0.8)
school.students[0].enterMark(0.9)

school.students[1].enterMark(0.6)
school.students[1].enterMark(0.4)
school.students[1].enterMark(0.5)

school.students[2].enterMark(0.7)
school.students[2].enterMark(0.8)
school.students[2].enterMark(0.5)
```

NOTE

Self.students is an aggregation.

Self.marks is just an instance variable of type list.

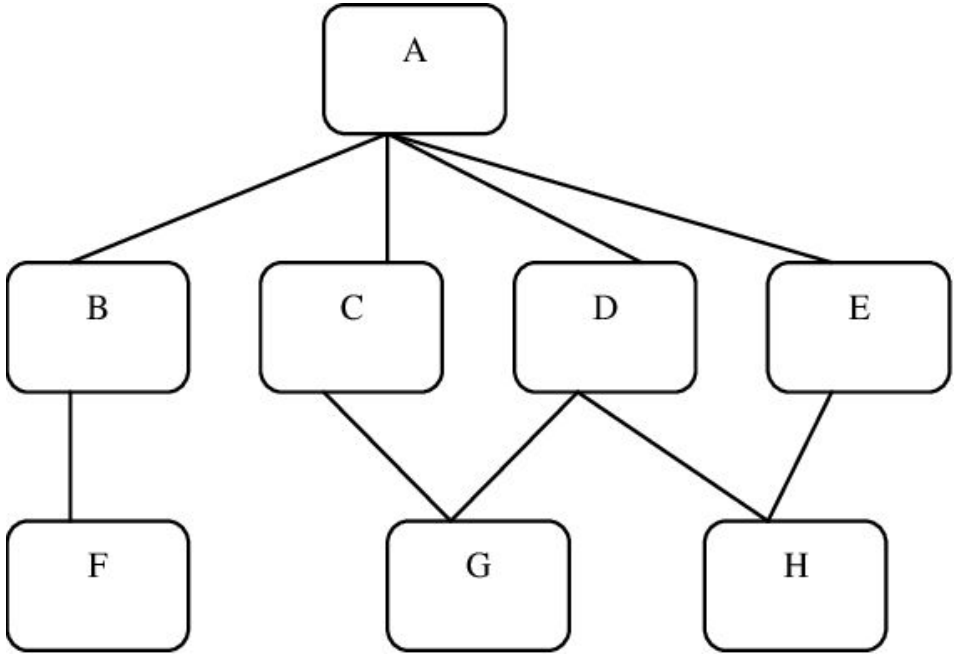
Output

```
Joseph's average this year was a 80.0%
Mark's average this year was a 50.0%
David's average this year was a 67.0%
```

```python
# Get the average for each student
for student in school.students:
    print(f"{student.name}'s average this year was a {round(student.calculateAVG(),2)*100}%")
```

03

Inheritance and Polymorphism



Inheritance vs. Polymorphism

Inheritance is where more than 1 classes have similar properties and/or methods. So, you put the similar properties and/or methods in a superclass and put the different ones in their respective subclasses.

Polymorphism is what allows the calling of a method from any subclass during runtime.

Inheritance vs. Polymorphism

Inheritance is a design concept which makes the code more reusable and extensible by putting common properties and methods in a superclass.

Polymorphism is the ability to go to the right class when a method is instantiated. For example, in this program, the shape class has 2 different forms: Circle and Rectangle. So, when shape1 (type circle) calls area or perimeter, python knows to go to the area() or perimeter() methods for type Circle. This is also true for Rectangle.

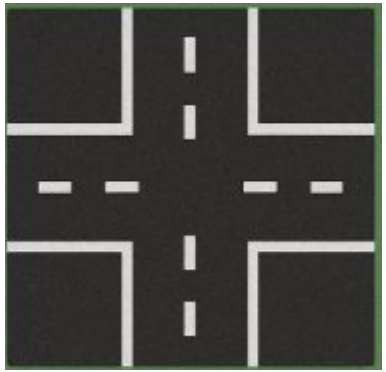
```python
import math

class shape():
    def __init__(self,name):
        self.name = name

    def area(self):
        pass

    def perimeter(self):
        pass
```

```python
class Circle(shape):
    def __init__(self,radius):
        super().__init__("Circle")
        self.radius = radius

    def area(self):
        return math.pi*self.radius*self.radius

    def perimeter(self):
        return 2*math.pi*self.radius

class Rectangle(shape):
    def __init__(self,length,width):
        super().__init__("Rectangle")
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2*self.length + 2*self.width
```

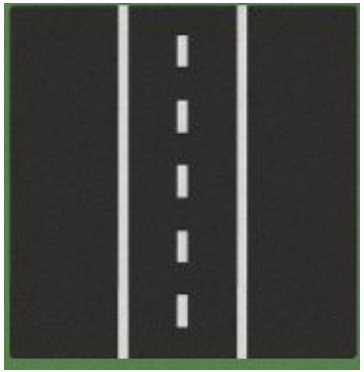
```python
shape1 = Circle(3)
shape2 = Rectangle(6,7)

print(f"The area and perimeter of this {shape1.name} is {shape1.area()} m^2, {shape1.perimeter()} m respectively.")

print(f"The area and perimeter of this {shape2.name} is {shape2.area()} m^2, {shape2.perimeter()} m respectively.")
```

Road Network Program

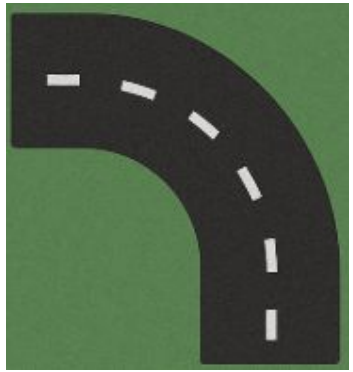
Imagine we are making a program where a user can place any tile type to connect them and make a road. There are 4 tiles types:



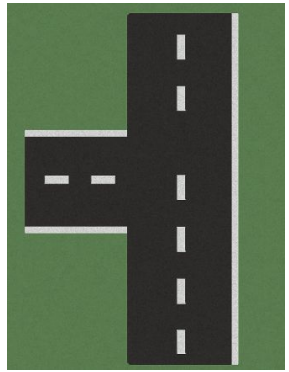
4_way_tile



2_way_tile



turn_tile

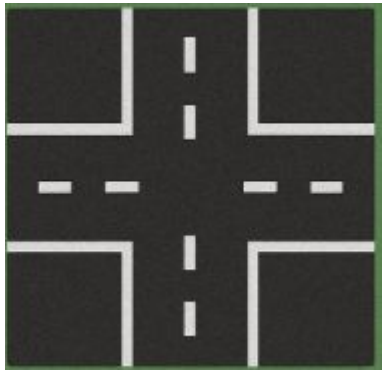


4_way_tile

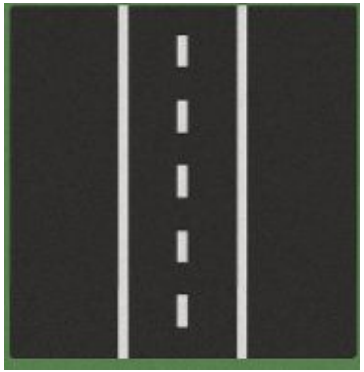
Now you can move the tiles anywhere in the grid. Also, you can rotate all 4 types of tiles, but the edges that are allowed to enter to exit from change depending on the tile type.

Road Network Program

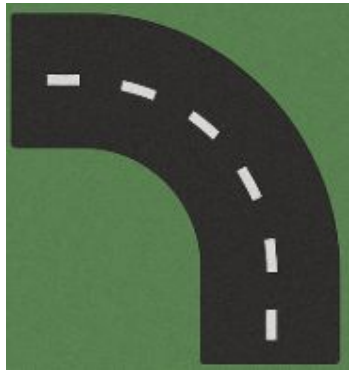
Imagine we are making a program where a user can place any tile type to connect them and make a road. There are 4 tiles types:



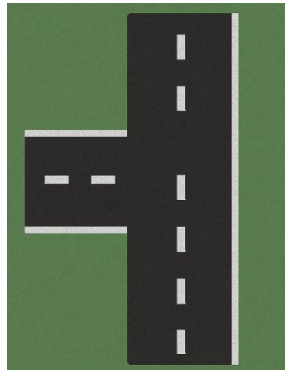
4_way_tile



2_way_tile



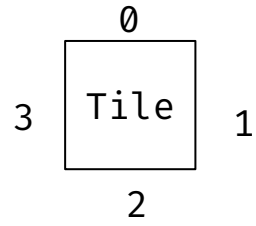
turn_tile

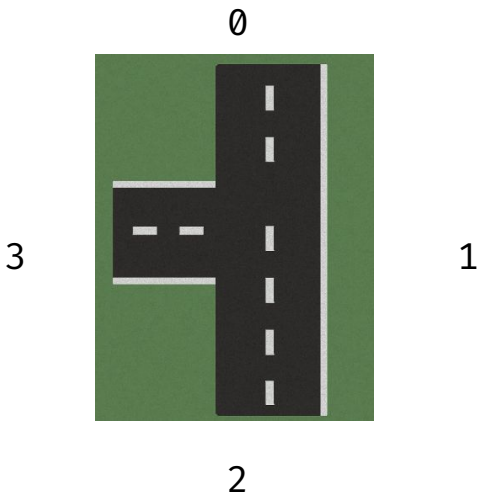


4_way_tile

Now you can move the tiles anywhere in the grid. Also, you can rotate all 4 types of tiles, but the edges that are allowed to enter to exit from change depending on the tile type.

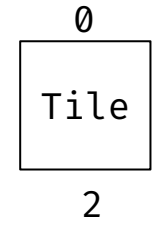
Road Network Program





When the user first instantiates a tile, it will have some image and its edges. Edges is a list of length 4 and the elements are either 0 for False and 1 for True. This list represents what are the edges that a car can enter or exit from for that specific tile. For example, **the 3_way_tile will have Self.edges = [1,0,1,1]**. So, a car can enter or exit from any edge except edge 1 or index 1 in the self.edges list.

Road Network Program



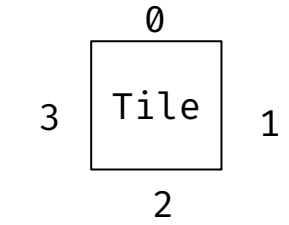

```python
class 4_way_tile():
    def __init__(self,image,edges,position):
        self.img = image
        self.position = position
        self.edges = edges

    def place(x,y):
        # self.position = (x,y)

    def left_rotate():
        # Just return the edges tile

    def right_rotate():
        # Just return the edges tile
```

```python
class 3_way_tile():
    def __init__(self,image,edges,position):
        self.img = image
        self.position = position
        self.edges = edges

    def place(x,y):
        # self.position = (x,y)

    def left_rotate():
        # Rotate Logic
        # shift the 0 to the right
        # then return edges

    def right_rotate():
        # Rotate Logic
        # shift the zero to the left
        # then return edges
```

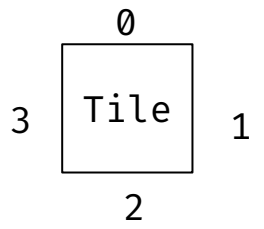
Road Network Program

```python
class 2_way_tile():
    def __init__(self,image,edges,position):
        self.img = image
        self.position = position
        self.edges = edges

    def place(x,y):
        # self.position = (x,y)

    def left_rotate():
        # Rotate Logic
        # if self.edges == [1,0,1,0]:
            # self.edges = [0,1,0,1]
        # else:
            # self.edges = [1,0,1,0]
        # then return edges

    def right_rotate():
        # Rotate Logic
        # if self.edges == [1,0,1,0]:
            # self.edges = [0,1,0,1]
        # else:
            # self.edges = [1,0,1,0]
        # then return edges
```

```python
class turn_tile():
    def __init__(self,image,edges,position):
        self.img = image
        self.position = position
        self.edges = edges

    def place(x,y):
        # self.position = (x,y)

    def left_rotate():
        # Rotate Logic
        # some different logic
        # then return edges

    def right_rotate():
        # Rotate Logic
        # some different logic
        # then return edges
```



Road Network Program



There is a problem with these class structures.

- You are duplicating a lot of code and logic that can be way more condensed.
- Your code will not be easily extensible. This means if I want to add more types of tiles, there will be more logic I have to code.

```python
class 4_way_tile():
    def __init__(self,image,edges,position):
        self.img = image
        self.position = position
        self.edges = edges

    def place(x,y):
        # self.position = (x,y)

    def left_rotate():
        # Just return the edges tile

    def right_rotate():
        # Just return the edges tile
```

```python
class 3_way_tile():
    def __init__(self,image,edges,position):
        self.img = image
        self.position = position
        self.edges = edges

    def place(x,y):
        # self.position = (x,y)

    def left_rotate():
        # Rotate Logic
        # shift the 0 to the right
        # then return edges

    def right_rotate():
        # Rotate Logic
        # shift the zero to the left
        # then return edges
```

All tile types have some position,image, edges, and have the same logic for place() method. In the case of rotation(), they have different logic but they all perform a common operation of rotating a tile.

Road Network Program

Inheritance and Polymorphism

```
class tile():
    def __init__(self,image,edges,position):
        self.img = image
        self.position = position
        self.edges = edges

    def place(x,y):
        # self.position = (x,y)

    def left_rotate():
        pass

    def right_rotate():
        pass
```

```
class 4_way_tile(tile):
    def __init__(self,image,edges,position):
        super().__init__(image,edges,position)

    def left_rotate():
        # Just return the edges tile

    def right_rotate():
        # Just return the edges tile
```

```
class 3_way_tile(tile):
    def __init__(self,image,edges,position):
        super().__init__(image,edges,position)

    def left_rotate():
        # Rotate Logic
        # shift the 0 to the right
        # then return edges

    def right_rotate():
        # Rotate Logic
        # shift the zero to the left
        # then return edges
```

```
class turn_tile(tile):
    def __init__(self,image,edges,position):
        super().__init__(image,edges,position)

    def left_rotate():
        # Rotate Logic
        # some different logic
        # then return edges

    def right_rotate():
        # Rotate Logic
        # some different logic
        # then return edges
```

```
class 2_way_tile(tile):
    def __init__(self,image,edges,position):
        super().__init__(image,edges,position)

    def left_rotate():
        # Rotate Logic
        # if self.edges == [1,0,1,0]:
            # self.edges = [0,1,0,1]
        # else:
            # self.edges = [1,0,1,0]
        # then return edges

    def right_rotate():
        # Rotate Logic
        # if self.edges == [1,0,1,0]:
            # self.edges = [0,1,0,1]
        # else:
            # self.edges = [1,0,1,0]
        # then return edges
```

So, you could call the super class to place the tile. But, When you call left rotate or right rotate, the subclass type you set the object to be will be the class where the rotate method will be called (this is what is called polymorphism).