Revised: 7/27/2020

Classes

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

Lecture Topics

- Basics of Object Oriented Design
    - Objects
    - Classes
- Creating Classes and Instances of Objects
    - Instance Variables/Fields
- Initializers
- Functions in Classes
    - Accessors
    - Mutators
        - Data Validation
- Copying Instances
- String Representation
- Class Diagrams

Colors/Fonts

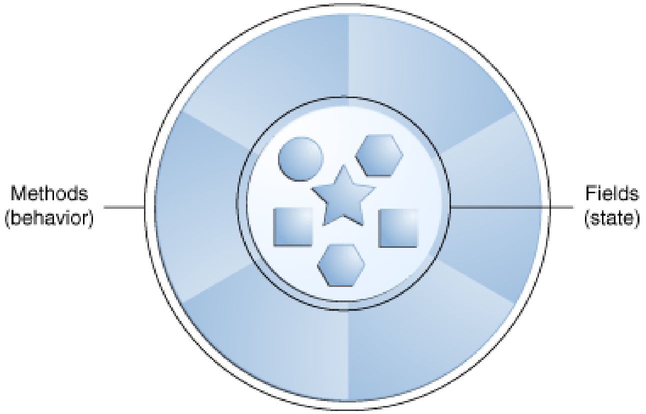
Global Variable Names	– Brown
Local Variable Names	– Lt Blue
Literals	– Blue
Keywords	– Orange
Operators/Punctuation	– Black
Functions	– Purple
Parameters	– Gold
Comments	– Gray
Modules	– Pink
Object/Class Names	– Green

Source Code	– **Consolas**
Output	– Courier New

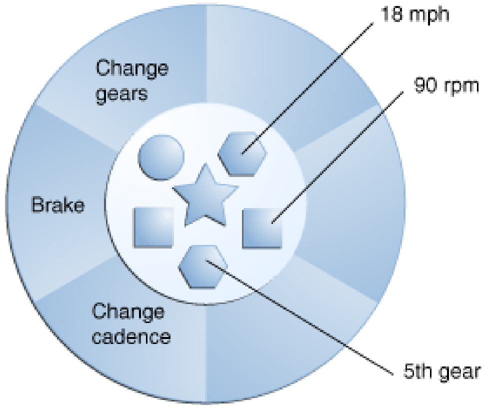
What is Object Oriented Design (OOD)

- ***Object Oriented Design*** is a programming paradigm where we model code after how it would act in the real-world.

Software Object



Bicycle Object



What are Objects?

- A software object is *conceptually* similar to real world objects.
- Real world objects all have two characteristics:
  - They have ***attributes***- properties that make something unique.
    - A bicycle's attributes could be its speed, color, tire size, etc.
  - They have ***behaviors***- actions that something can do.
    - A bicycle's behaviors could be pedaling, braking, changing gear, etc.

- When we model a software object, it too has attributes and behaviors.
  - Objects store their attributes in variables referred to as ***fields.***
  - Objects expose their behaviors as ***functions.***

Abstraction

- ***Abstraction*** is an OOD principle that software objects are able to function as individual entities.
  - They can, for example, return information about their attributes by communicating with other objects.
  - Perform any internal operations that other objects need not be concerned with.

- Our Bicycle object example should allow other objects to know information about itself, like what color it is or how fast its moving.
  - But the more complex details, like the code that handles the bike's speed, remains hidden.

Encapsulation

- ***Encapsulation*** is an OOD principle that suggests we design classes so that all relevant data (attributes) and behaviors (functions) are together.

- Only attributes and behaviors relevant to the object should be in the object.
  - Other data or functions not related to the object's use should be placed in other objects.
  - For example, it wouldn't make sense for a Bicycle object to have a fuel level attribute (but would, perhaps, make sense in a Moped object.)

What are Classes?

- A ***class*** is the blueprint from which individual objects are created.

- It is the *source code* of our object.
  - The object is the idea, the code in the class is the implementation of the idea.

Classes

- A ***class declaration*** or *class header* for an object named “Bicycle” is shown below.
- The source code contained in a class is the ***class definition*** or *class body*.

```
class Bicycle :
```

The __init__ Function

- The ***initializer*** is called when a new object is created from the class.

- All functions in a class must have a self parameter.
  - Functions can have additional parameters but the first must always be self.

```
class Bicycle :
    def __init__(self) :
```

Indent one tab.

Declaring Instance Variables

- An ***instance variable*** is a variable that represents an object’s attribute/field.
  - The data stored in an instance variable is unique to each instance of an object.
  - Instance variables are normally declared in the initializer.

```
class Bicycle :

    def __init__(self) :
        self.gear = 0
        self.speed = 0
        self.color = "x"
```

```
class Bicycle :

    def __init__(self) :
        gear = 1
        speed = 1
        color = "Red"
```

Creates local variables →

- Instance variables are global variables.
  - They will be accessible by all functions in the class.
- Instance variables begin with the **self** keyword.
  - The self keyword is how an object refers to itself.

Classes as a Module

- In most circumstances, your class's code is in a separate source code file.

- The same rules will apply to classes as they do modules.
  - The class will need to be imported in order to create instances in other programs.

Creating an Instance of an Object

- In a second class named bicycletest.py, we will instantiate a Bicycle object in its main method (shown below).
  - ***Instantiation*** is the term used when you create an *instance* of an object.

bicycle.py

```
class Bicycle :

  def __init__(self) :
    self.gear = 0
    self.speed = 0
    self.color = "x"
```

bicycletest.py

```
from bicycle import Bicycle


def main() :
  test_bike = Bicycle()

main()
```

Class Name

Accessing an Instance’s Fields

- We can access an instance’s fields using dot notation.

bicycle.py

```
class Bicycle :

  def __init__(self) :
    self.gear = 0
    self.speed = 0
    self.color = "x"
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle()
  print(test_bike.gear)
  test_bike.gear = 5
  print(test_bike.gear)

main()
```

```
0

5
```

Accessing an Instance’s Fields

bicycle.py

```
class Bicycle :

  def __init__(self) :
    self.gear = 0
    self.speed = 0
    self.color = "x"
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle()
  print(test_bike.gear)
  test_bike.gear = 5
  print(test_bike.gear)
  test_bike.speed = 8
  test_bike.color = "Blue"
  print(test_bike.speed)
  print(test_bike.color)

main()
```

```
0
5
8
Blue
```

Accessing an Instance’s Fields

- An instance field’s value is unique from other instances.

bicycle.py

```
class Bicycle :

  def __init__(self) :
    self.gear = 0
    self.speed = 0
    self.color = "x"
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike1 = Bicycle()
  test_bike2 = Bicycle()
  test_bike1.gear = 5
  test_bike2.gear = 4
  print(test_bike1.gear)
  print(test_bike2.gear)

main()
```

```
5
4
```

The Initializer

- The initializer may accept arguments.
  - The first parameter must always be the self keyword.

```python
class Bicycle :

  def __init__(self, gear_in) :
    self.gear = gear_in
    self.speed = 0
    self.color = "x"
```

The Initializer

bicycle.py

```
class Bicycle :

  def __init__(self, gear_in) :
    self.gear = gear_in
    self.speed = 0
    self.color = "x"
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(6)
  print(test_bike.gear)
  print(test_bike.speed)
  print(test_bike.color)

main()
```

```
6
0
x
```

The Initializer

bicycle.py (Code portion)

```
class Bicycle :

  def __init__(self,
               gear_in,
               speed_in) :
  self.gear = gear_in
  self.speed = speed_in
  self.color = "x"
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(6, 3)
  print(test_bike.gear)
  print(test_bike.speed)
  print(test_bike.color)

main()
```

```
6
3
x
```

The Initializer

bicycle.py

```
class Bicycle :

  def __init__(self,
               gear_in,
               speed_in,
               color_in) :
    self.gear = gear_in
    self.speed = speed_in
    self.color = color_in
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(6, 3, "Blue")
  print(test_bike.gear)
  print(test_bike.speed)
  print(test_bike.color)

main()
```

```
6
3
Blue
```

Functions in Classes

- Most functions (*behaviors*) in an object are either an accessor function or a mutator function.
- An **accessor function** retrieves (or “gets”) data from an object.
  - Colloquially called a “getter” function.
- A **mutator function** changes (or “sets”) data in an object.
  - Colloquially called a “setter” function.
- The use of accessors and mutators allows the object to specify how its data is accessed or changed.

Functions in Classes

- It is not always a good idea to access a field directly, because there is no way to specify what data can be assigned to the field.
  - This is why we normally use functions to get and set data.

bicycle.py

```
class Bicycle :

  def __init__(self,
               gear_in,
               speed_in,
               color_in) :
    self.gear = gear_in
    self.speed = speed_in
    self.color = color_in
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(6, 3, "Blue")
  test_bike.gear = "abcd"
  test_bike.gear = [7, 3, 100]

main()
```

Accessor Functions

- An accessor function should simply return data from an object.
  - The function’s name should describe the data it returns.
  - They normally don’t accept arguments.

bicycle.py

```
class Bicycle :

  def __init__(self,
               gear_in,
               speed_in,
               color_in) :
    self.gear = gear_in
    self.speed = speed_in
    self.color = color_in

  def getgear(self) :
    return self.gear
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(6, 3, "Blue")
  print(test_bike.getgear())

main()
```

```
6
```

Accessor Functions

bicycle.py

```
class Bicycle :


  def getgear(self) :
    return self.gear

  def getspeed(self) :
    return self.speed

  def getcolor(self) :
    return self.color
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(6, 3, "Blue")
  print(test_bike.getgear())
  print(test_bike.getspeed())
  print(test_bike.getcolor())

main()
```

```
6
3
Blue
```

Mutator Functions

- An mutator function accepts data/arguments to give to an object.
  - The function’s name should describe the data it accepts.
  - Rarely will need to return a value.

bicycle.py

```
class Bicycle :


  def setgear(self, gear_in) :
    self.gear = gear_in
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(6, 3 , "Blue")
  test_bike.setgear(5)
  print(test_bike.getgear())

main()
```

```
5
```

Mutator Functions

bicycle.py

```
class Bicycle :


  def setgear(self, gear_in) :
    self.gear = gear_in

  def setspeed(self, speed_in) :
    self.speed = speed_in

  def setcolor(self, color_in) :
    self.color = color_in
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(6, 3, "Blue")
  test_bike.setgear(5)
  test_bike.setspeed(7)
  test_bike.setcolor("Green")
  print(test_bike.getgear())
  print(test_bike.getspeed())
  print(test_bike.getcolor())

main()
```

```
5
7
Green
```

Data Validation

- It’s always a good idea to validate data that is passed to a function.

- Mutators can check that the data passed to it is…
  - The right type of data.
  - In a range of acceptable values.
    - For example, the data is between the values 1 and 10.

Mutator Functions

- Perhaps we want the setgear function to only accept values between 1 and 10.
  - If the value passed in is outside of that range, we'll raise an exception.
  - It will be up to the code that called the getgear function to handle the exception.

bicycle.py

```
class Bicycle :


 def setgear(self, gear_in) :
  if gear_in >= 1 and gear_in <= 10 :
   self.gear = gear_in
  else :
   raise ValueError("Value not in 1-10")

```

Mutator Functions

bicycle.py

```
class Bicycle :


  def setgear(self, gear_in) :
    if gear_in >= 1 and gear_in <= 10 :
      self.gear = gear_in
    else :
      raise ValueError("Value not in 1-10")
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(6, 3 , "Blue")
  test_bike.setgear(5)
  print(test_bike.getgear())

main()
```

```
5
```

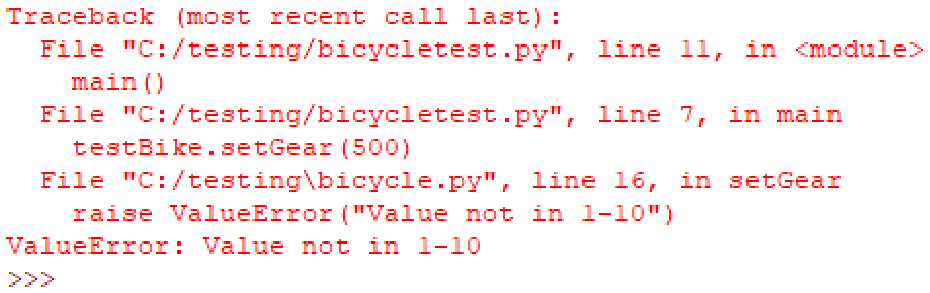
Mutator Functions

bicycle.py

```
class Bicycle :


  def setgear(self, gear_in) :
    if gear_in >= 1 and gear_in <= 10 :
      self.gear = gear_in
    else :
      raise ValueError("Value not in 1-10")

```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(6, 3, "Blue")
  test_bike.setgear(500)
  print(test_bike.getgear())

main()
```



```
Traceback (most recent call last):
  File "C:/testing/bicycletest.py", line 11, in <module>
    main()
  File "C:/testing/bicycletest.py", line 7, in main
    testBike.setGear(500)
  File "C:/testing\bicycle.py", line 16, in setGear
    raise ValueError("Value not in 1-10")
ValueError: Value not in 1-10
>>>
```

Testing a Variable’s Type

- Python’s built-in **isinstance** function determines if a variable’s data is a certain type.
  - The first argument is the variable to test.
  - The second argument is the type.

```python
value1 = 5
value2 = "Test"
value3 = [5, 6, 7]

if isinstance(value1, int) :
  print("value1 is an int")

if isinstance(value2, float) :
  print("value2 is a float")

if isinstance(value3, set) :
  print("value3 is a set")
```

```
value1 is an int
```

Common Types:

```
int
float
bool
str
list
tuple
set
dict
```

Testing a Variable’s Type

- Mutators can validate the type of data passed and raise an exception if it is the wrong type.

bicycle.py

```
class Bicycle :

  def setgear(self, gear_in) :
    if not isinstance(gear_in, int) :
      raise TypeError("Value not an int")

    if gear_in >= 1 and gear_in <= 10 :
      self.gear = gear_in
    else :
      raise ValueError("Value not in 1-10")

```

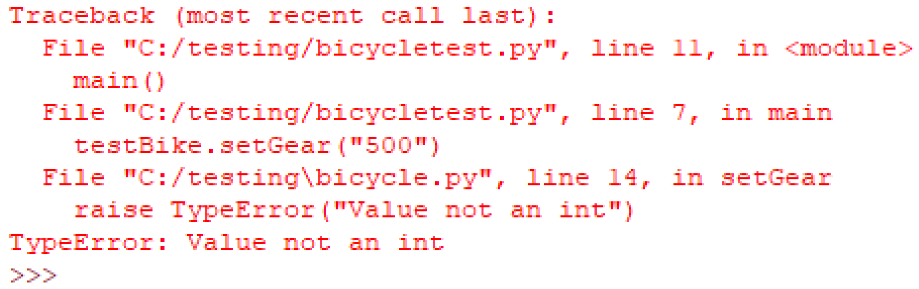
Testing a Variable’s Type

bicycle.py

```
class Bicycle :

  def setgear(self, gear_in) :
    if not isinstance(gear_in, int) :
      raise TypeError("Value not an int")

    if gear_in >= 1 and gear_in <= 10 :
      self.gear = gear_in
    else :
      raise ValueError("Value not in 1-10")
```

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(6, 3 , "Blue")
  test_bike.setgear("500")
  print(test_bike.getgear())

main()
```



```
Traceback (most recent call last):
  File "C:/testing/bicycletest.py", line 11, in <module>
    main()
  File "C:/testing/bicycletest.py", line 7, in main
    testBike.setGear("500")
  File "C:/testing\bicycle.py", line 14, in setGear
    raise TypeError("Value not an int")
TypeError: Value not an int
>>>
```

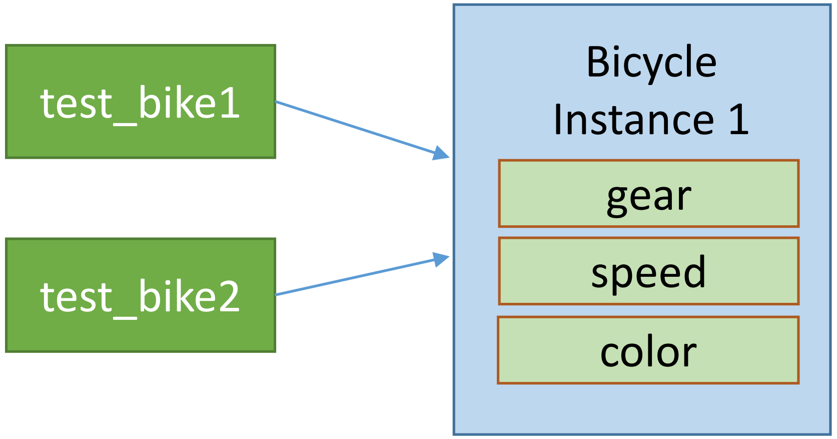
Copying Instances

- An instance of an object can be shallow copied or deep copied.

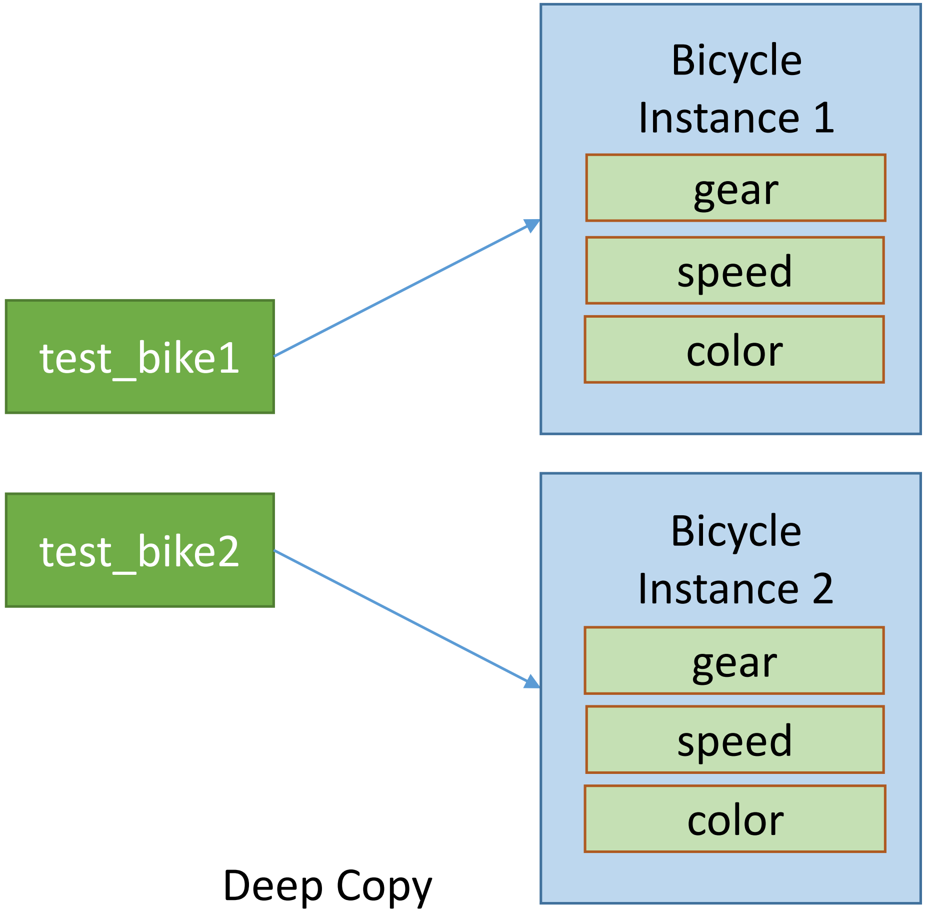
- Shallow Copy: The ***reference*** to data at a location in memory is copied from one variable to a different variable. In essence, both variables reference the same data/object in memory, NOT their own.

- Deep Copy: The ***data*** referenced by one variable is copied to a new location in memory, and is then referenced by a different variable.

Copying Instances



Shallow Copy



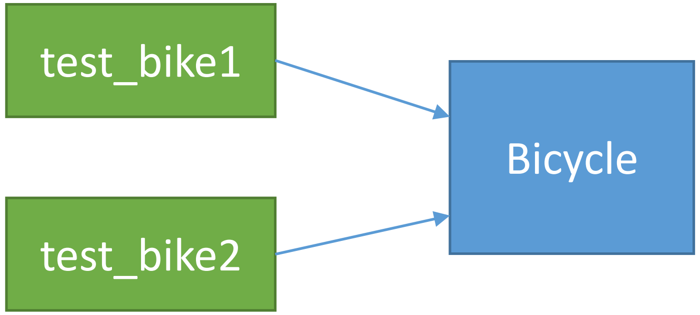
Deep Copy

Shallow Copying Instances

- Use the assignment operator (=) to shallow copy an instance.
- Remember, the shallow copy is not a new instance.
    - The new variable will point to the same instance in memory.

```
from bicycle import Bicycle

def main() :
  test_bike1 = Bicycle(6, 3, "Blue")
  test_bike2 = test_bike1
  test_bike2.setgear(4)
  print(test_bike1.getgear())

main()
```

```
4
```



Deep Copying Instances

- A deep copy gives us an entirely new instance with the current state of the instance we wish to copy.
  - All fields of the new instance should have the same values as the original instance.

- Writing a copy function is common technique to deep copy instances.
  - The function should return a new instance with all of the new instance’s fields set to the same values as the original instance.

Copy Functions

- This copy function will return a new instance of a Bicycle object, using the class’s own fields as arguments to the initializer of the new instance.

bicycle.py

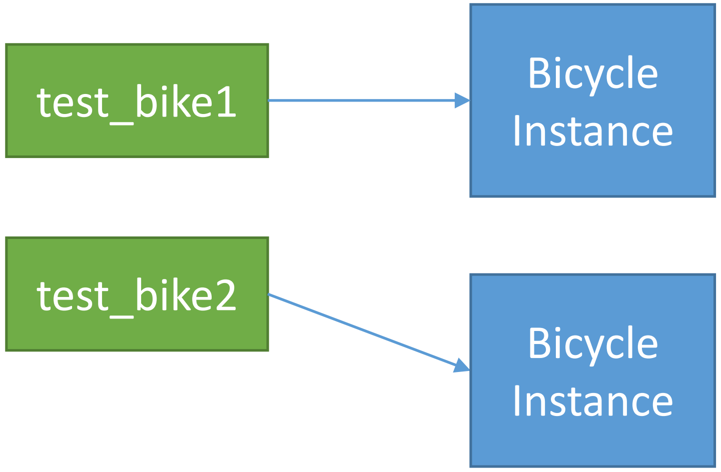
```
class Bicycle :

  def __init__(self, gear_in, speed_in, color_in) :
    self.gear = gear_in
    self.speed = speed_in
    self.color = color_in


  def copy(self) :
    return Bicycle(self.gear, self.speed, self.color)
```

Copy Functions

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike1 = Bicycle(6, 3, "Blue")
  test_bike1.setgear(2)
  test_bike1.setspeed(7)
  test_bike1.setcolor("Green")
  test_bike2 = test_bike1.copy()
  test_bike2.setgear(5)
  print(test_bike1.getgear())
  print(test_bike2.getgear())

main()
```

```
2
5
```



test_bike1

Bicycle Instance

test_bike2

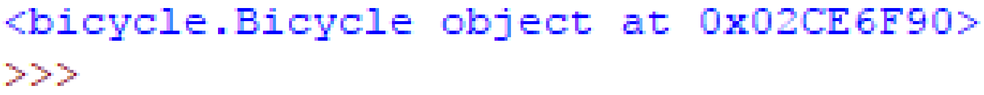
Bicycle Instance

String Representation

- When we pass an instance to the print function, it prints the object’s name/module and its memory address.

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(6, 3, "Blue")
  print(test_bike)

main()
```



```
<bicycle.Bicycle object at 0x02CE6F90>
>>>
```

__str__ Functions

- When a class has a __str__ function, it will be called when instances of the class are passed to the print function.

bicycle.py

```
class Bicycle :


  def __str__(self) :
    toString = "Bicycle Information: \n"
    toString += "Current Gear: " + str(self.gear) + "\n"
    toString += "Current Speed: " + str(self.speed) + "\n"
    toString += "Color: " + self.color

    return toString
```

String Representation

- Now, when we pass the instance to the print function, it uses the string returned by the __str__ function.

bicycletest.py

```python
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(5, 7, "Blue")
  print(test_bike)

main()
```

```
Bicycle Information:
Current Gear: 5
Current Speed: 7
Color: Blue
```

String Representation

- We will get the same string returned if we first passed the instance to the built-in str function.

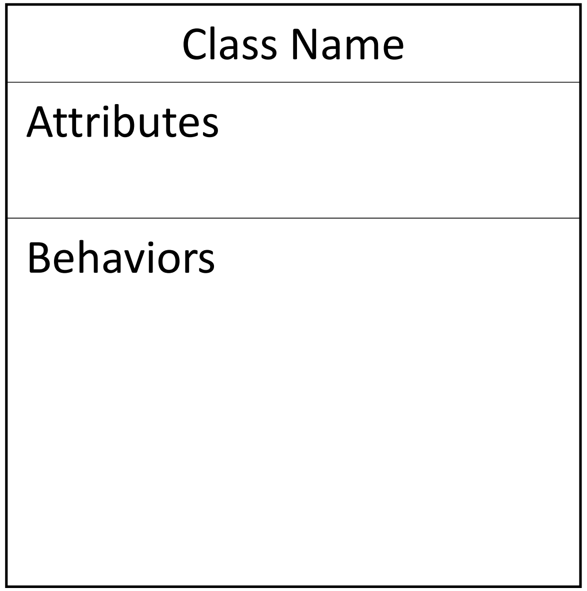
bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle(5, 7, "Blue")
  strValue = str(test_bike)
  print(strValue)

main()
```

```
Bicycle Information:
Current Gear: 5
Current Speed: 7
Color: Blue
```

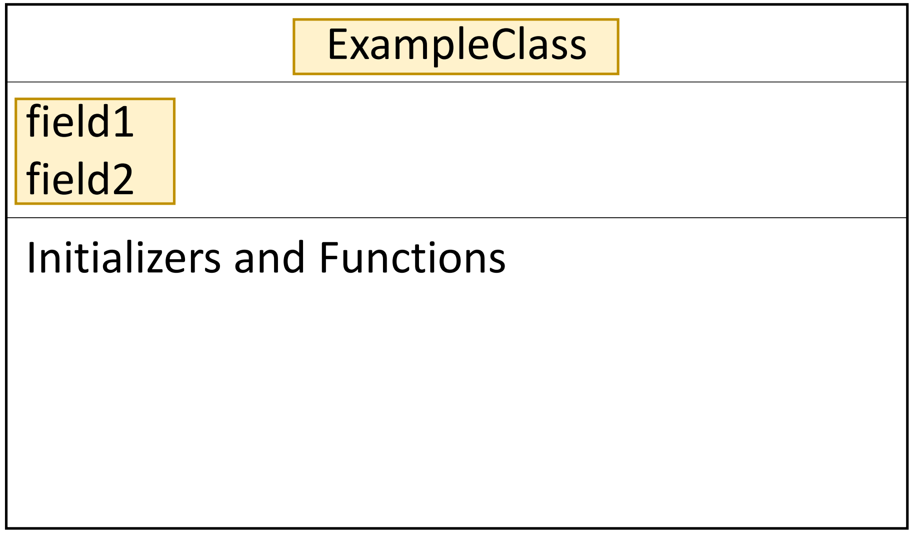
Class Diagrams

- Unified Modeling Language provides a set of standard diagrams for graphically depicting an object oriented system.
- In UML, each class is shown as a box, with three sections:
  - The Class Name
  - Class Attributes (Fields)
  - Class Behaviors (Functions)



Class Diagram (Name and Fields)

```
class ExampleClass :

  def __init__(self, arg1, arg2) :
    self.field1 = arg1
    self.field2 = arg2
    #Other initializer code

  def method1(self) :
    #Function code

  def method2(self, arg) :
    #Function code
```



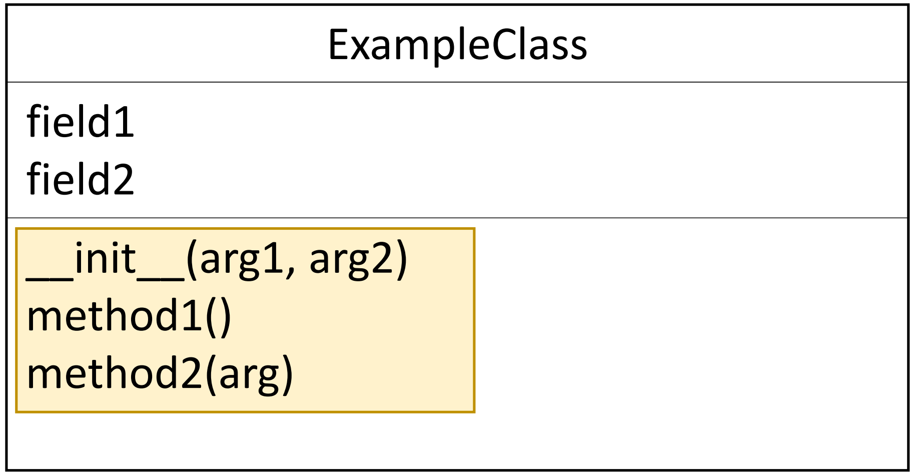
ExampleClass

field1
field2

Initializers and Functions

Class Diagram (Initializers and Functions)

```
class ExampleClass :
    def __init__(self, arg1, arg2) :
        self.field1 = arg1
        self.field2 = arg2
        #Other initializer code

    def method1(self) :
        #Function code

    def method2(self, arg) :
        #Function code
```



ExampleClass

field1
field2

__init__(arg1, arg2)
method1()
method2(arg)