



# CMSC 105 Elementary Programming

Acknowledgement: These slides are adapted from slides provided with "Introduction to Programming Using Python, Liang (Pearson 2013)" and slides shared by Dr. Jory Denny

# Outline

---

Classes & Objects

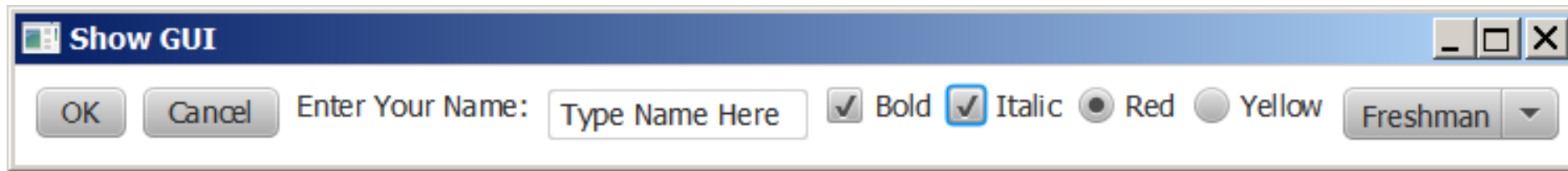
---

Practice Exercises

---

# Motivations

- Suppose you want to develop a graphical user interface as shown below. How do you program it?



- Facebook?
- Pixar animations?
- Amazon

# Class and Objects



University is an example of class.



Department of Art and Art History

Department of Biology

Department of Chemistry

...

Department of Theatre and Dance

Example of objects belonging to a class.

Lectures

Assignments

Labs or reports

Exams

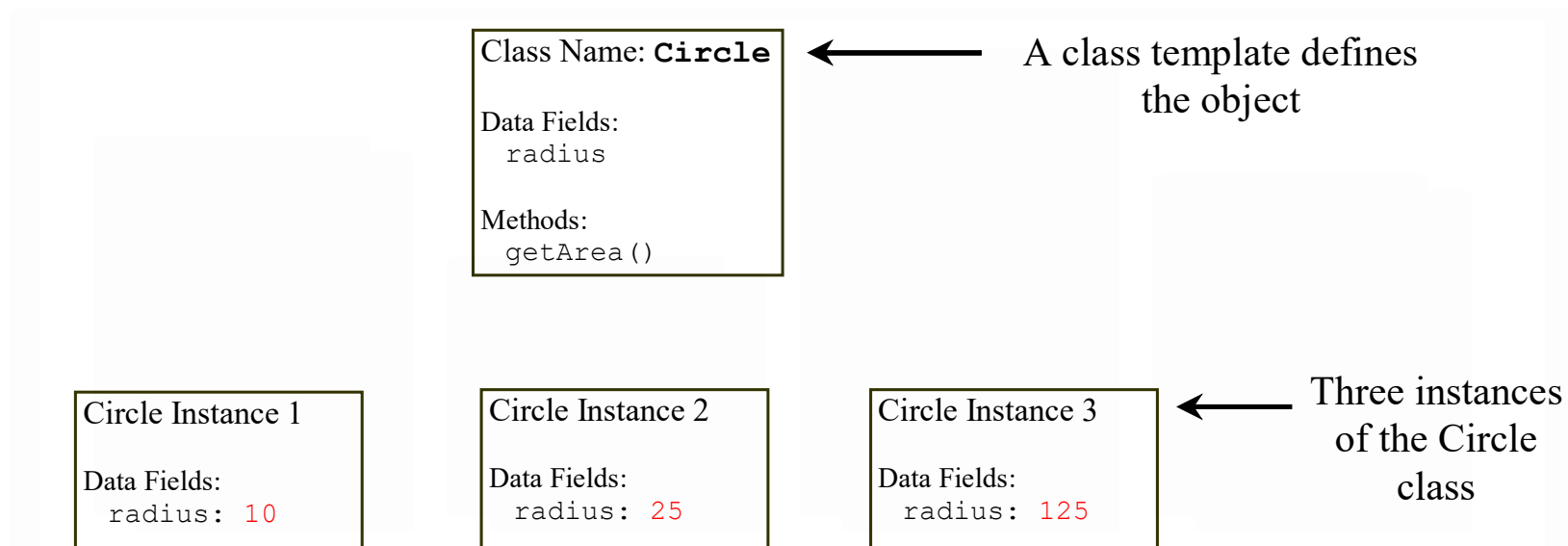
Example of attributes and methods that are shared in common by the objects of a class.

# Object-oriented Programming Concepts

- **Object-oriented programming (OOP)** involves programming using objects
- An **object** represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behaviors.
  - The **state** of an object consists of a set of **data fields** (also known as **properties**) with their current values.
  - The **behavior** of an object is defined by a set of methods.

# Objects

- An **object** has both a state and behavior. The state defines the object, and the behavior defines what the object does.
  - An object **class** defines its possible states and its behaviors
  - An object **instance** is a variable of the object type, i.e., it is a specific “value” or state



# Classes

- **Classes** are constructs that define objects of the same type
- A Python class uses variables to store data fields and defines methods to perform actions. Additionally, a class provides a special type method, known as *initializer*, which is invoked to create a new object. An initializer can perform any action, but an initializer is designed to perform initializing actions, such as creating the data fields of objects.

# Example Class

What else do you notice?

```
import math
class Circle:
    def __init__(self, radius = 1): # Construct a circle
        self.radius = radius        # Define data fields

    def getPerimeter(self):          # Methods operate on data
        return 2*self.radius*math.pi

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

    def setRadius(self, radius):
        self.radius = radius
```

Note, \_\_ is two underscores.

# Constructing Objects

- Once a class is defined, you can create objects from the class by using the following syntax, called a **constructor**:

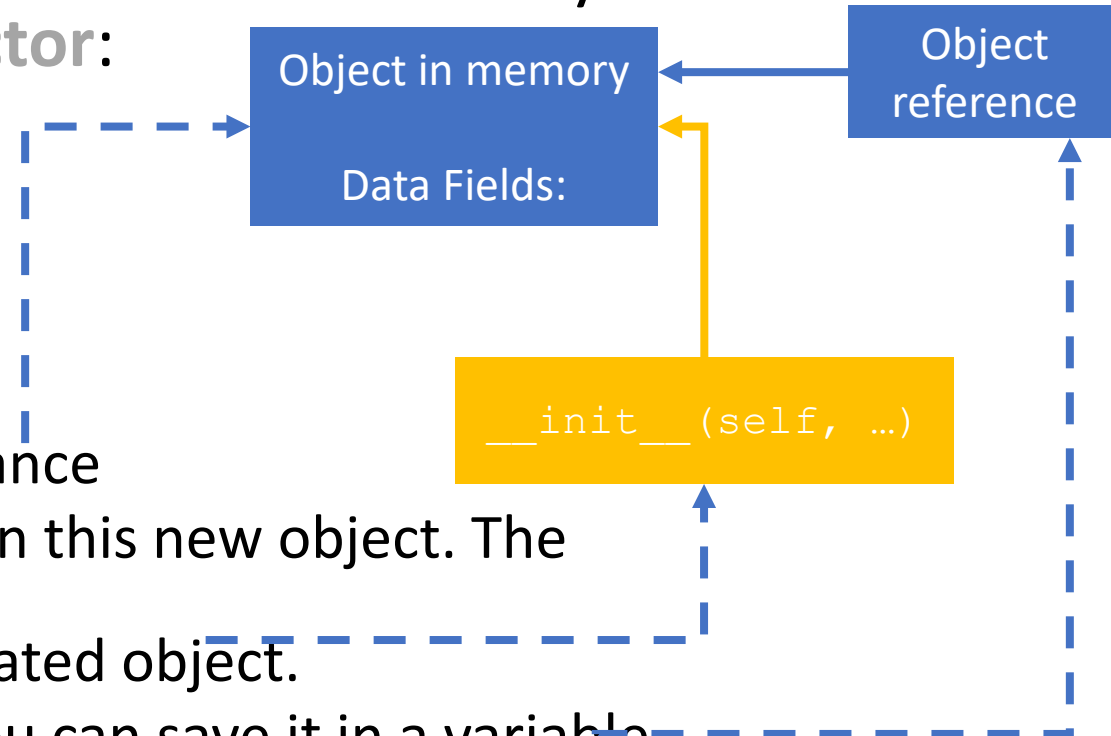
```
className (arguments)
```

- Example:

```
Circle(50)
```

- What happens?

- A new object is created in memory for this instance
- The special method `__init__()` is invoked on this new object. The **self** parameter is automatically set to the newly created object.
- A reference to the object is returned, so that you can save it in a variable.



# Instance Methods

- **Methods** are functions defined inside a class. They are **invoked** by objects to perform actions on the objects.
- For this reason, the methods are also called *instance methods* in Python. You probably noticed that all the methods including the constructor have the first parameter **self**, which refers to the object that invokes the method. You can use any name for this parameter. But by convention, self is used.
- Example:

```
c1 = Circle(50)
c2 = Circle(30)
a1 = c1.getArea() # Here c1 is the self argument
a2 = c2.getArea() # Here c2 is the self argument
```

# Accessing Objects

- After an object is created, you can access its data fields and invoke its methods using the dot operator (.), also known as the *object member access operator*.

- Example:

```
c = Circle(50)
a = c.getArea()
p = c.getPerimeter()
```

# Tracing

```
1. myCircle = Circle(5.0)
2. yourCircle = Circle()
3. yourCircle.setRadius(100)
```


Memory



# Tracing

```
1. myCircle = Circle(5)
2. yourCircle = Circle()
3. yourCircle.setRadius(100)
```

Declare myCircle



Memory

myCircle:  
None

# Tracing

```
1. myCircle = Circle(5)
2. yourCircle = Circle()
3. yourCircle.setRadius(100)
```

Create a circle

## Memory

myCircle:  
None

0xA

### Circle

radius	5
--------	---

# Tracing

```
1. myCircle = Circle(5)
2. yourCircle = Circle()
3. yourCircle.setRadius(100)
```

Assign memory  
location to  
reference variable

## Memory

myCircle:  
0xA (reference)

0xA

### Circle

radius	5
--------	---

# Tracing

```
1. myCircle = Circle(5)
2. yourCircle = Circle()
3. yourCircle.setRadius(100)
```

Declare yourCircle

## Memory

myCircle:  
0xA (reference)

yourCircle:  
None

0xA

**Circle**

radius

5

# Tracing

```
1. myCircle = Circle(5)
2. yourCircle = Circle()
3. yourCircle.setRadius(100)
```

Create a circle

## Memory

myCircle:  
0xA (reference)

yourCircle:  
None

0xA

Circle	
radius	5

0xB

Circle	
radius	1

# Tracing

```
1. myCircle = Circle(5)
2. yourCircle = Circle()
3. yourCircle.setRadius(100)
```

Assign memory  
location to  
reference variable

## Memory

myCircle:  
0xA (reference)

0xA

Circle	
radius	5

yourCircle:  
0xB

0xB

Circle	
radius	1

# Tracing

```
1. myCircle = Circle(5)
2. yourCircle = Circle()
3. yourCircle.setRadius(100)
```

Change radius in  
your circle

## Memory

myCircle

0xA (reference)

0xA

**Circle**

radius

5

yourCircle

0xB

0xB

**Circle**

radius

100

# Why self?

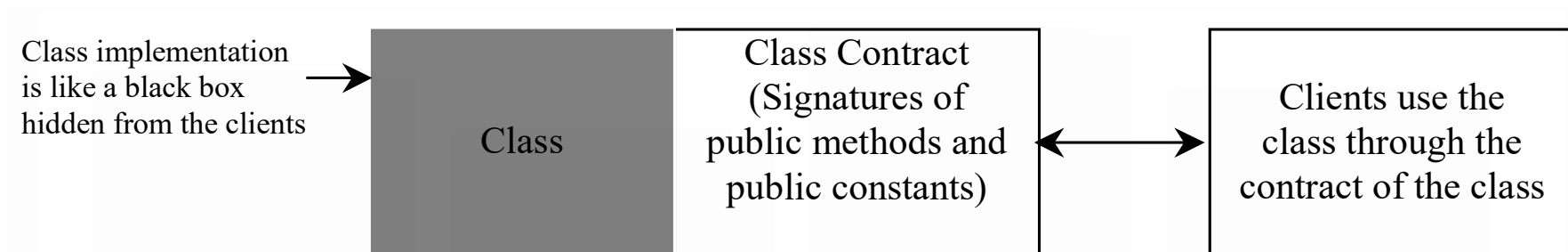
- Note that the first parameter is special. It is used in the implementation of the method, but not used when the method is called.
- So, what is this parameter **self** for? Why does Python need it?
- **self** is a parameter that represents an object
  - Using **self**, you can access instance variables in an object, which storing data fields
  - Each object is an instance of a class and instance variables are tied to specific objects. Thus, each object has its own unique instance variables.
  - You can use the syntax **self.x** to access the instance variable x for the object **self** inside of a method definition.

# Object-Oriented Programming

- **Object-oriented Programming** – design principle for large programs
  - **Abstraction** – Modeling objects
  - **Composition** – Modeling object associations (HAS-A relationship)
  - **Encapsulation** – combining data and operations (methods); data hiding from misuse (private vs public)
  - **Inheritance** – Types and sub-types (IS-A relationship)
  - **Polymorphism** – Abstract types that can act as other types (for algorithm design)

# Abstraction and Encapsulation

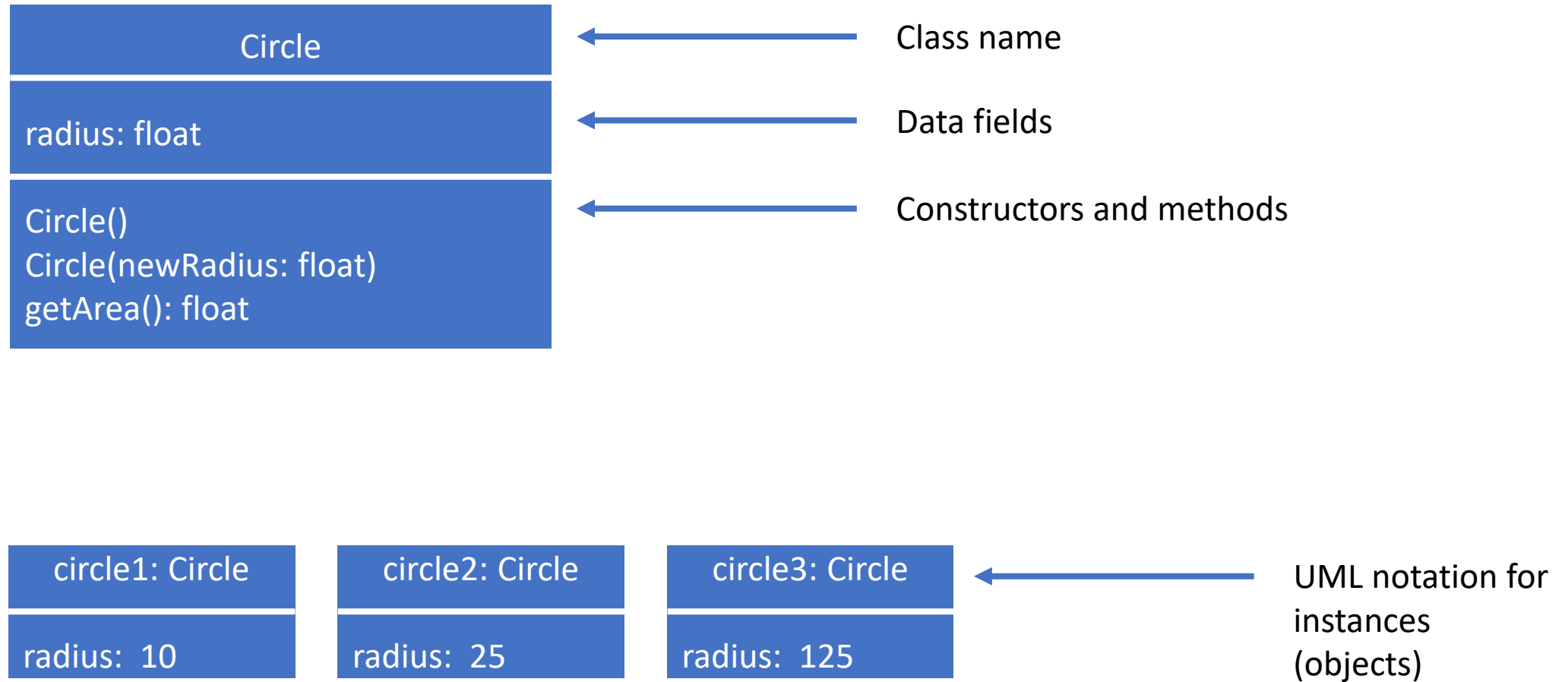
- **Abstraction** means to separate class implementation from the use of the class.
  - A description of the class lets the user know how the class can be used (class **contract**)
  - Thus, the user of the class does not need to know how the class is implemented
  - The detail of implementation is **encapsulated** and hidden from the user.



# UML Class Diagram

- An aside: in design, we often document a class in a special diagram called UML, or Unified Modeling Language.
- In this, we describe classes, their data, methods, and the relationships to other objects.

# UML Class Diagram for Abstraction



# Example UML Diagram

## Defining a TV Object

TV	
channel: int volumeLevel: int on: bool	The current channel (1 to 120) of this TV. The current volume level (1 to 7) of this TV. Indicates whether this TV is on/off.
TV() turnOn(): None turnOff(): None getChannel(): int setChannel(channel: int): None getVolume(): int setVolume(volumeLevel: int): None channelUp(): None channelDown(): None volumeUp(): None volumeDown(): None	Constructs a default TV object. Turns on this TV. Turns off this TV. Returns the channel for this TV. Sets a new channel for this TV. Gets the volume level for this TV. Sets a new volume level for this TV. Increases the channel number by 1. Decreases the channel number by 1. Increases the volume level by 1. Decreases the volume level by 1.

# Data Field Encapsulation

- Important to protect data from misuse, i.e., prevent direct modifications of data fields, don't let the client directly access data fields.
- Important to make class easy to maintain
- **Data field encapsulation** is accomplished by defining **private** data fields. In Python, the private data fields are defined with two leading underscores. You can also define a private method named with two leading underscores

# Data Field Encapsulation

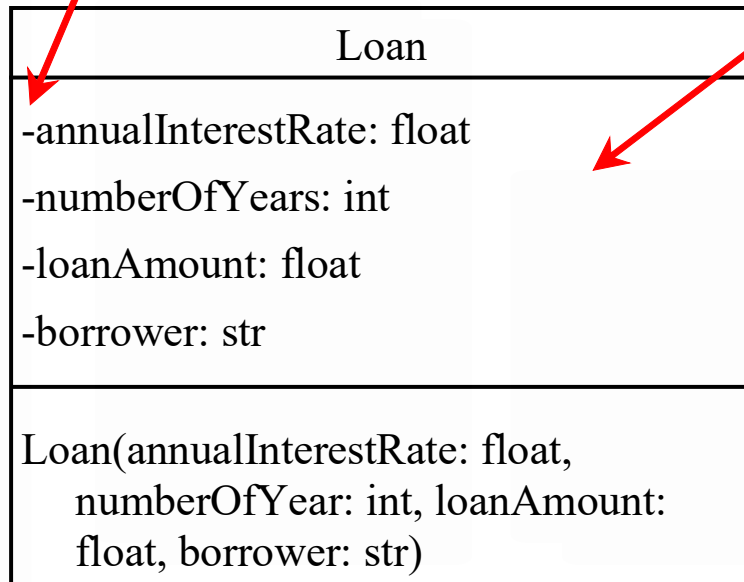
- Sometimes, accessing this variable will give an AttributeError:

```
c = Circle(5)
print(c.__radius) # AttributeError
                  # Note if radius was public
                  # (no __ inside the class)
                  # this would work
```

- Again, *most of the time*, data should be kept private to prevent misuse

# UML Class diagram for encapsulation

The – sign denotes a private data field.



The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The annual interest rate of the loan (default: 2.5).

The number of years for the loan (default: 1)

The loan amount (default: 1000).

The borrower of this loan.

Constructs a Loan object with the specified annual interest rate, number of years, loan amount, and borrower.

# Accessors/Modifiers

- Methods which read/use the data without modifying it are commonly referred to as **accessors**
  - Methods that alter the data of an object are referred to as **modifiers**
  - A common accessor/modifier pair is a **getter/setter** for a specific data member
    - The getter method simply returns the data value
    - The setter method simply sets a new value to the data
- What types are the following methods in the circle class?
    - `getRadius()`
    - `setRadius()`
    - `getArea()`
    - `getPerimeter()`

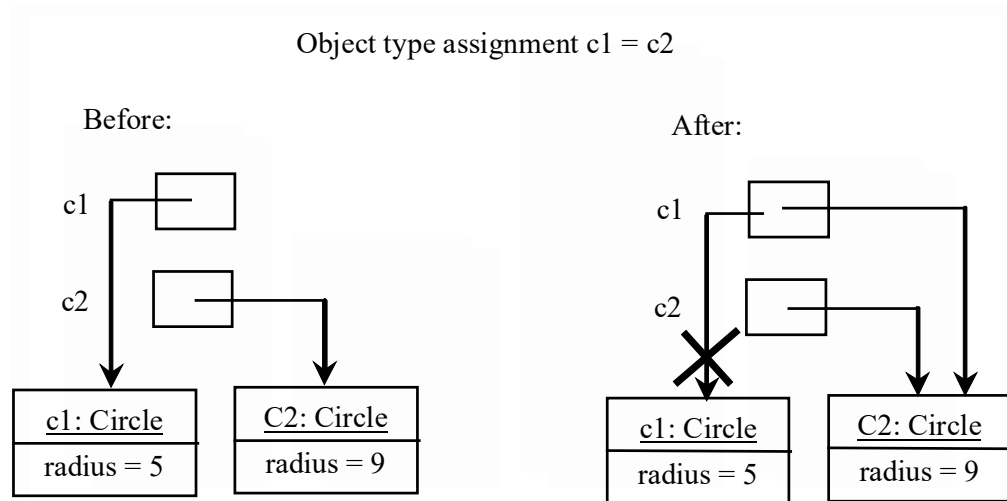
# Immutability

- If the contents of an object cannot be changed once the object is created, the object is **immutable**.
  - If you delete the set method in the Circle class, the class would be immutable because radius is private and cannot be changed without a set method.
- The objects for integers/float/string are immutable in python. This is why they act like primitive types.

# Scope

- Variables private to a class should only be accessed within that class.
- Recall – **scope** is the lifetime of a variable. It dictates where you as the programmer may refer to the identifier (name) in code
  - Rule – The scope of class member variables is the entire class (including inside of any method). They can be declared anywhere inside a class.
  - Rule – The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.

# References passed to functions/copy



- When passing objects into functions, they are passed-by-object-reference. This means that the object that is passed to the function is modified directly.
- During assignment of variables, the reference is being copied!

# Static and class variables

- You can also have variables shared among all instances, these are called class or static variables
- Declare them at the top of the class:

```
class Circle:
    numInstances = 0
    def __init__(self, radius=1):
        self.__radius = radius
        Circle.numInstances += 1
```

# Static and class functions

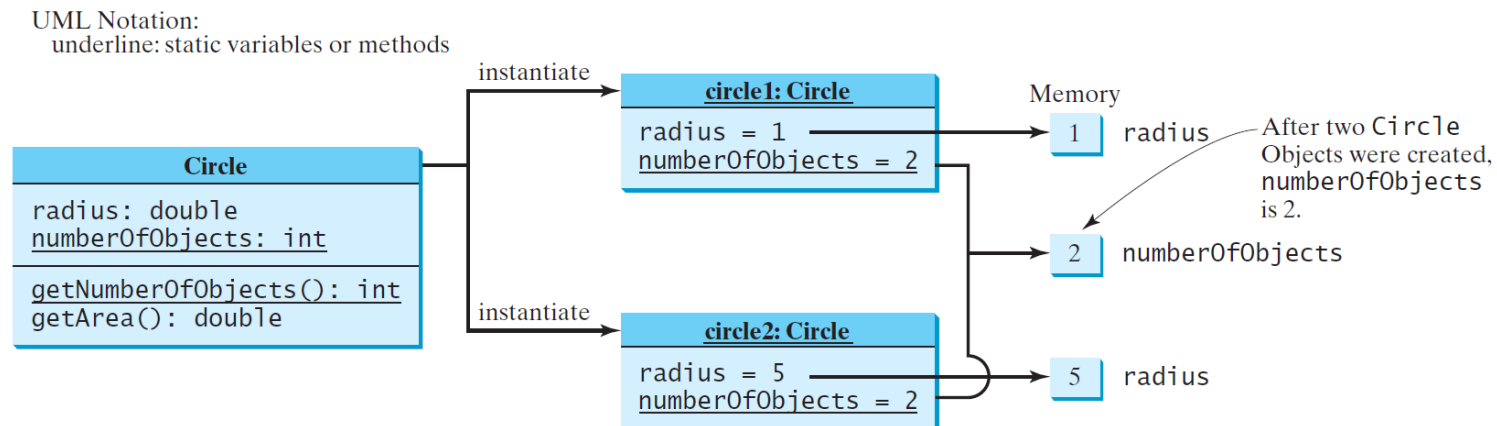
- Class functions can operate on the class or static variables
- First parameter will be `cls` (for class) and variables can be accessed from it.  
Demarcated with `@classmethod`
- Example (inside of a class):

```
@classmethod
def getNumInstances(cls):
    return cls.numInstances
```
- Static functions can only read class or static variables
- Takes no special parameters.  
Demarcated with `@staticmethod`
- Serves as just a utility function
- Example (inside of a class):

```
@staticmethod
def pi(places):
    return round(math.pi,
places)
```

# Instance vs Static

- Instance – a, or relating to a, specific object's value
  - Instance variables belong to a specific instance.
  - Instance methods are invoked by an instance of the class.
- Static – not a, or relating to a, specific object's value (related to the type).
  - Static variables are shared by all the instances of the class.
  - Static methods are not tied to a specific object.



# Exercise

- Make and test a class rectangle, defined by width and height (define `__init__`).
  - Have methods to compute its area and perimeter
- Create a class bank with the following properties:
  - `account_number`
  - `name`
  - `balance`

Given the properties above, write methods to:

- `__init__(self, number, name, balance)`
- `show_balance(self)`
- `withdraw(self, amount)` (update the balance after amount withdrawal).
- `Deposit(self, amount)` (update the balance after amount deposit)



Thank you!  
Questions?