COP 3035

Intro Programming in Python

Summer 2024

Lecture 17 – part 1

Exam 3 – 07/12/24
Lab 8 - 07/15/24
Homework 6 – 07/19/24

Lecture 17 – part 2

Review

Review

Functions

Default arguments

Positional arguments *args

Keyword arguments **kwargs

Scope

Python Scope with the LEGB Rule

The **LEGB rule** is a well-established guideline in the Python community to comprehend the **order** in which Python searches for variable names. The acronym stands for:

L: Local

- Names assigned within a function (def or lambda).
- Not declared as global within that function.

E: Enclosing function locals

- Names in the local scope of any and all enclosing functions (def or lambda), from innermost to outermost.

G: Global (module)

- Names assigned at the top-level of a module.
- Or declared as global within a def in the file.

B: Built-in (Python)

- Names preassigned in Python like open, range, SyntaxError, etc.

In simple terms:

- By default, name assignments will create or change local names.
- Name references search through, at most, four scopes. These scopes, in order, are:
  - local
  - enclosing functions
  - global
  - built-in
- Names declared in **global** and **nonlocal** statements map the assigned names to the enclosing module and function scopes.

Lecture 17 – part 3

Object Oriented Programming

Object Oriented Programming (OOP)

- OOP is a programming paradigm that uses **"objects"** and their interactions to design applications and computer programs.
- It facilitates more flexible and manageable code, making it easier to modify, extend, and maintain software.

Benefits of OOP:

- **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects.
- **Reusability:** Objects can be reused across programs.

Classes and Objects

Classes:

- Think of classes as **blueprints** for creating objects (a particular data structure).
- They define a type in terms of its data and the operations that can be performed on it.

```
class Car:
    pass


class Dog:
    def __init__(self,breed):
        self.breed = breed
```

Objects:

- Objects are **instances** of classes.
- They embody both data (attributes) and ways to manipulate that data (methods).

```
my_car = Car()

sam = Dog(breed='Lab')
frank = Dog(breed='Huskie')
```

Constructor Method `__init__()`

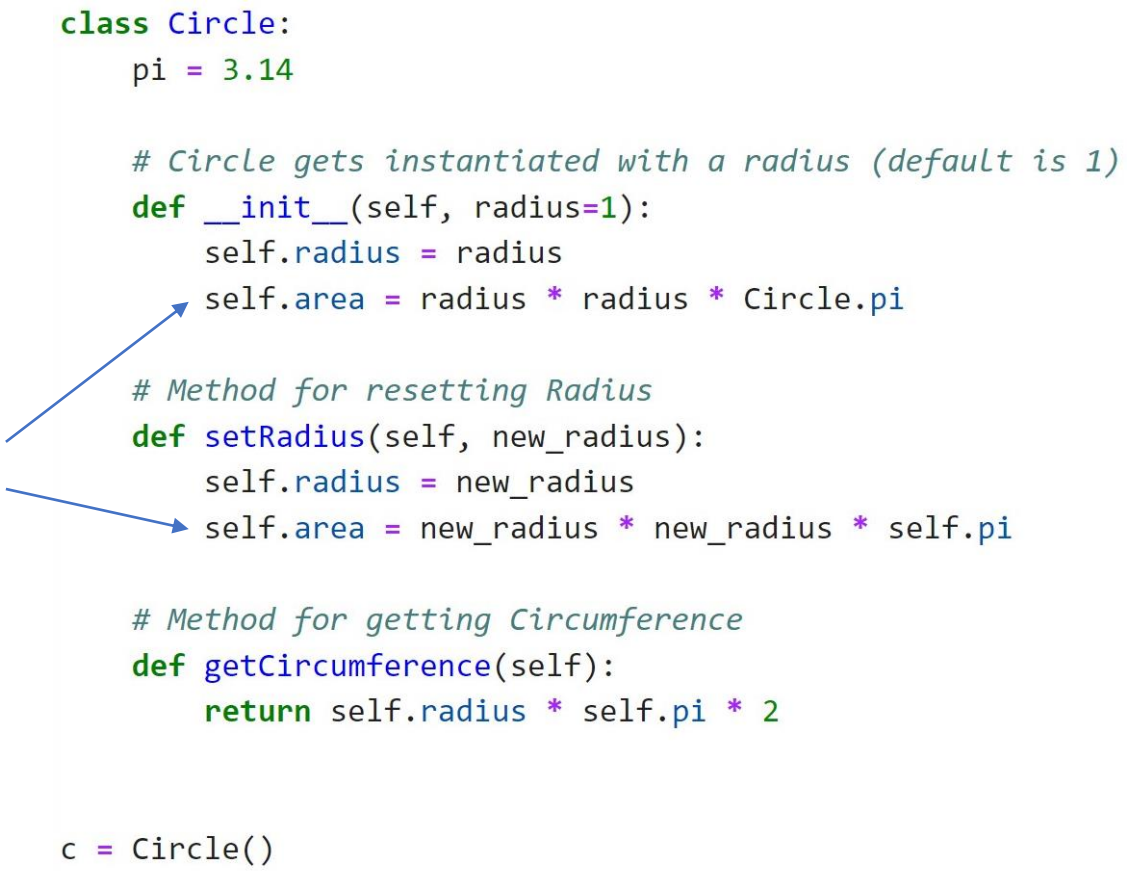
- The `__init__()` method in Python is a special method used for initializing newly created objects.
- It's called automatically when a new instance of a class is created.
- It can take arguments to initialize the object's attributes

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

my_car = Car("Toyota", "Corolla")
print(f"My car is a {my_car.make} {my_car.model}.")
```

Instance Methods

- **Definition:** Instance methods are **functions** defined inside a class that operate on an instance of the class. They implicitly take the instance itself as the first argument, conventionally named **self**.
- **Purpose:** Used to access and modify the state of a specific object of the class.
- **Example:** In the Circle class, both **setRadius** and **getCircumference** are instance methods..



```python
class Circle:
    pi = 3.14

    # Circle gets instantiated with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * Circle.pi

    # Method for resetting Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi

    # Method for getting Circumference
    def getCircumference(self):
        return self.radius * self.pi * 2


c = Circle()
```

Inheritance

- Inheritance is a way to form new classes using classes that have already been defined.
- The newly formed classes are called derived classes, the classes that we derive from are called base classes.
- Important benefits of inheritance are code reuse and reduction of complexity of a program.
- The derived classes (descendants) override or extend the functionality of base classes (ancestors).

Base class (or superclass)

```
class Animal:
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
        print("Animal")

    def eat(self):
        print("Eating")
```

Derived class (or subclass)

```
class Dog(Animal):
    def __init__(self):
        super().__init__(self)
        print("Dog created")

    def whoAmI(self):
        print("Dog")

    def bark(self):
        print("Woof!")
```

override

extend

Polymorphism

- Polymorphism is an OOP principle that allows objects of different classes to be treated as objects of a common superclass.
- In python polymorphism refers to the way in which different object classes can **share the same method** name, and those methods can be called from the same place even though a variety of different objects might be passed in.
- Polymorphism is achieved through methods that have the **same name but possibly act differently** based on which object calls them.

```
class Dog:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Woof!'

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

    def speak(self):
        return self.name+' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak())
print(felix.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Class Variables

- Defined within a class but outside any instance methods
- Shared across all instances of the class
- Accessed using the class name as well as by instance references
- Ideal for storing constants and default values

```
class Circle:
    pi = 3.14

    # Circle gets instantiated with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * Circle.pi

    # Method for resetting Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi

    # Method for getting Circumference
    def getCircumference(self):
        return self.radius * self.pi * 2


c = Circle()
```

Class variable

Instance variable

Composition

Composition:

- A "has-a" relationship where a class is made up of components of another class.
- The composed object cannot exist independently of the owning class.
- Lifecycle dependency: When the owning class is destroyed, its components are also destroyed.

- Example:
  - A Car class containing a instance of a Engine class. If the car ceases to exist, the engine associated with it also cease to exist.

- Use it when you need a strong association between the container object and the contained object(s).

```python
class Engine:
    def start(self):
        print("Engine starting.")

    def stop(self):
        print("Engine stopping.")

# Composition example
class Car:
    def __init__(self):
        self.engine = Engine()  # Car has-a Engine

    def start(self):
        self.engine.start()

    def stop(self):
        self.engine.stop()

myCar = Car()
myCar.start()
```

```
Engine starting.
```

Aggregation

Aggregation:

- A "has-a" relationship that represents ownership between two classes, but with less tightly coupled lifecycles.
- The aggregated object can exist independently of the owning class.
- Lifecycle dependency: When the owning class is destroyed, its aggregated objects can continue to exist.

- Example:
  - A Department class containing instances of a Professor class. Professors can exist without the department.
- Use it when you want to maintain a relationship between objects without enforcing a strong lifecycle dependency.

```python
class Professor:
    def __init__(self, name):
        self.name = name

    def teach(self):
        return "{} is teaching".format(self.name)

class Department:
    # Aggregation
    def __init__(self, name):
        self.name = name
        self.professors = []  # Department "has-a" Professor,
                              # but Professors can exist independently

    def add_professor(self, professor):
        self.professors.append(professor)

    def get_professors(self):
        return [professor.name for professor in self.professors]

# Aggregation example
math_department = Department("Mathematics")
prof_john = Professor("John Doe")
math_department.add_professor(prof_john)

print(math_department.get_professors())
print(prof_john.teach())
```

```
['John Doe']
John Doe is teaching
```

Class Methods and Static Methods

Class Methods:

- Bound to the class rather than its object.
- Can modify the class state that applies across all instances.
- Defined with the **`@classmethod`** decorator.
- Automatically takes the class (**cls**) as the first argument.

Static Methods:

- Behave like regular functions but belong to the class's namespace.
- Do not have access to cls or self unless explicitly passed.
- Defined with the **`@staticmethod`** decorator.
- Useful for utility functions that don't access class or instance state.

```python
class MyClass:
    count = 0

    def __init__(self):
        MyClass.count += 1

    @classmethod
    def get_count(cls):
        return f"There are {cls.count} instances of MyClass."

    @staticmethod
    def utility_function(value):
        return value ** 2

# Using class method
print(MyClass.get_count())
instance = MyClass()
print(MyClass.get_count())

# Using static method
print(MyClass.utility_function(5))  # Output: 25
```

```
There are 0 instances of MyClass.
There are 1 instances of MyClass.
25
```