COP 3035

Intro Programming in Python

Summer 2024

Lecture 16 – part 1

Exam 3 – 07/12/24
Lab 8 - 07/15/24

Lecture 16 – part 2

Review

Review

Functions

Lambda Expressions

sort(), sorted()

default values, *args, **kargs

Lambda Expressions

- Lambda expressions allow us to create "anonymous" functions.
- We can quickly make ad-hoc functions without needing to properly define a function using def.
- Lambda's body is a single expression, not a block of statements.

```
def square(num):
    result = num**2
    return result
```

```
lambda num: num ** 2
```

```
def square(num): return num**2
```

```
square = lambda num: num **2
```

.sort(), sorted()

.sort(): In-place sorting:

- This method modifies the list it is called on.
- This method does not return a new list; it returns None.
- List only: This method is specific to lists.

```
listnumbers = [5, 2, 9, 1, 5, 6]

numbers.sort()

print(numbers)
[1, 2, 5, 5, 6, 9]
```

sorted(): Returns a new list:

- This function creates a new sorted list from the iterable passed to it.
- Works on **any** iterable: This function can accept any iterable (e.g., lists, tuples, strings, dictionaries).

```
sorted_numbers = sorted(numbers)

print(sorted_numbers)
[1, 2, 5, 5, 6, 9]
```

Sorting - lambda expressions as custom key

Lambda expressions are often used with these sorting functions to sort based on a custom key.

```
# Example list of tuples
students = [("John", 25), ("Jane", 22), ("Dave", 20)]

# Sort by age (in-place)
students.sort(key = lambda student: student[1])

# Output
print(students)

[('Dave', 20), ('Jane', 22), ('John', 25)]
```

Note:
student: variable that represents each tuple in the students list during the sorting process.

student[1]: is the age.

Note: For descending order use : reverse = True

Sorting with Multiple Keys

```
# Example list of tuples
students = [("John", 25, 3.9), ("Jane", 22, 3.8), ("Dave", 20, 4.0)]

# Sort by age first, then by GPA
students.sort(key=lambda student: (student[1], student[2]))
print(students)

[('Dave', 20, 4.0), ('Jane', 22, 3.8), ('John', 25, 3.9)]

# Sort by age first (descending), then by GPA (ascending)
students.sort(key=lambda student: (student[1], -student[2]), reverse=True)
print(students)

[('John', 25, 3.9), ('Jane', 22, 3.8), ('Dave', 20, 4.0)]
```

Question: can you sort dictionaries ?

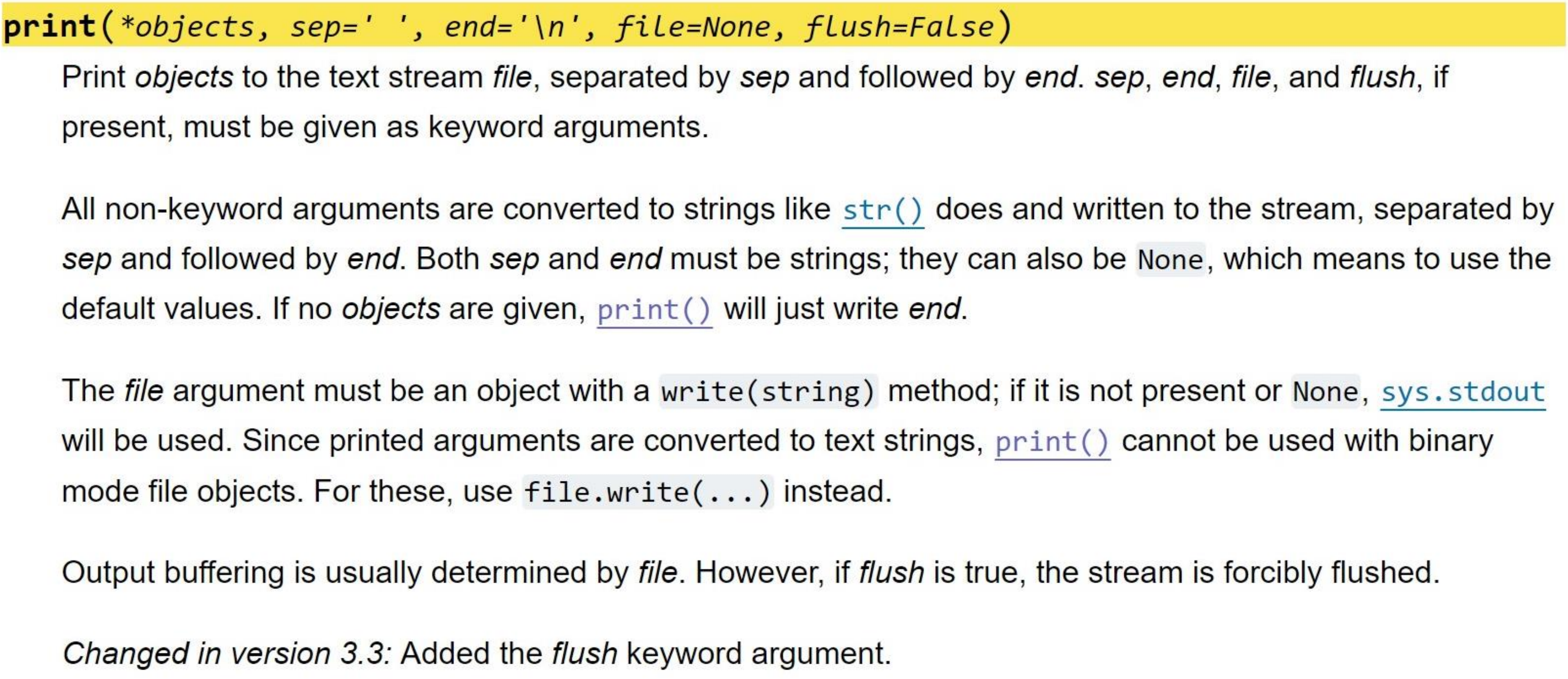
Lecture 16 – part 3

default values, *args, **kwargs

The print function

Same as *args

Default values



```
print(*objects, sep=' ', end='\n', file=None, flush=False)
```

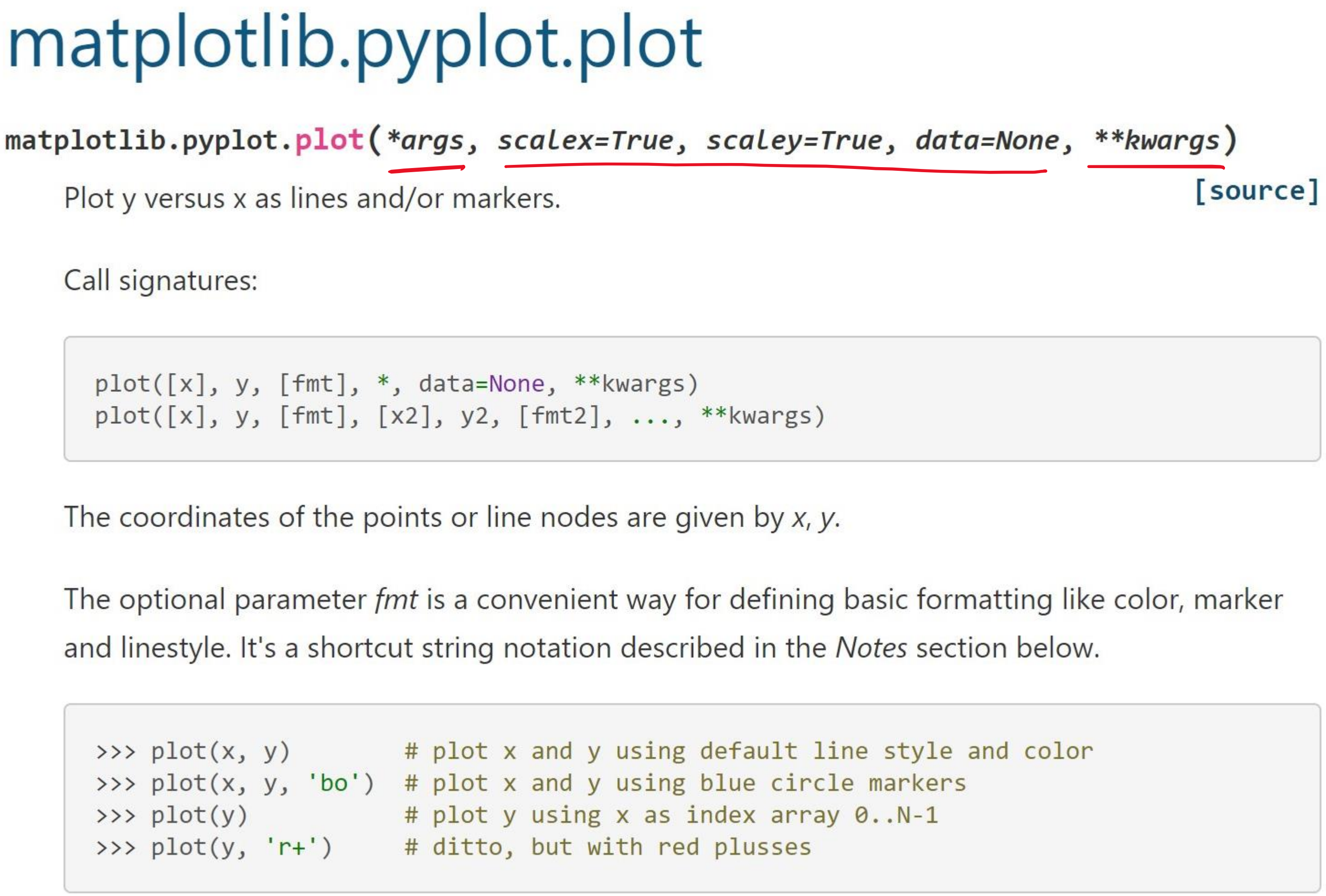
Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file*, and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Output buffering is usually determined by *file*. However, if *flush* is true, the stream is forcibly flushed.

Changed in version 3.3: Added the *flush* keyword argument.



matplotlib.pyplot.plot

matplotlib.pyplot.plot(*args, scalex=True, scaley=True, data=None, **kwargs) [source]

Plot y versus x as lines and/or markers.

Call signatures:

```
plot([x], y, [fmt], *, data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

The coordinates of the points or line nodes are given by *x*, *y*.

The optional parameter *fmt* is a convenient way for defining basic formatting like color, marker and linestyle. It's a shortcut string notation described in the *Notes* section below.

```
>>> plot(x, y)        # plot x and y using default line style and color
>>> plot(x, y, 'bo')  # plot x and y using blue circle markers
>>> plot(y)           # plot y using x as index array 0..N-1
>>> plot(y, 'r+')     # ditto, but with red plusses
```

Default Values in Function Parameters

- Functions can have **default values for parameters**.
- These defaults are used if no argument is passed for that parameter.

Syntax:

```
def function_name(param=default_value):
```

- Default values make function arguments optional.
- If an argument is passed, it overrides the default value

```python
def greet(name, message="Hello"):
    return f"{message}, {name}!"

print(greet("Alice"))             # Output: Hello, Alice!

print(greet("Alice", "Goodbye"))  # Output: Goodbye, Alice!
```

Understanding ***args** in Python

*args allows a function to accept any number of **positional arguments**.

Syntax:

```
def function_name(*args):
```

How it works:

- Inside the function, args is accessible as a tuple.
- Enables flexible function calls without specifying the exact number of arguments.

```
def test_function(*args):

        for i,a in enumerate(args):

                print(f'Index: {i}, Argument: {a}’)

return 0
```

```
a = test_function(1,2,3,4,5)
```

```
Index: 0, Argument: 1
Index: 1, Argument: 2
Index: 2, Argument: 3
Index: 3, Argument: 4
Index: 4, Argument: 5
```

```
a = test_function('salary',200,10)
```

```
Index: 0, Argument: salary
Index: 1, Argument: 200
Index: 2, Argument: 10
```

Understanding ****kwargs** in Python

kwargs allows a function to accept any number of **keyword arguments.

Syntax:

```
def function_name(**kwargs):
```

How it works:

- Inside the function, kwargs is accessible as a dictionary.
- Facilitates receiving named arguments not predefined in function parameters.

```
def person_details(**kwargs):

    for key, value in kwargs.items():

        print(f"{key}: {value}")


person_details(name="John", age=30, city="New York")
```

```
name: John
age: 30
city: New York
```

Lecture 16 – part 4

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 16 – part 5

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')
```

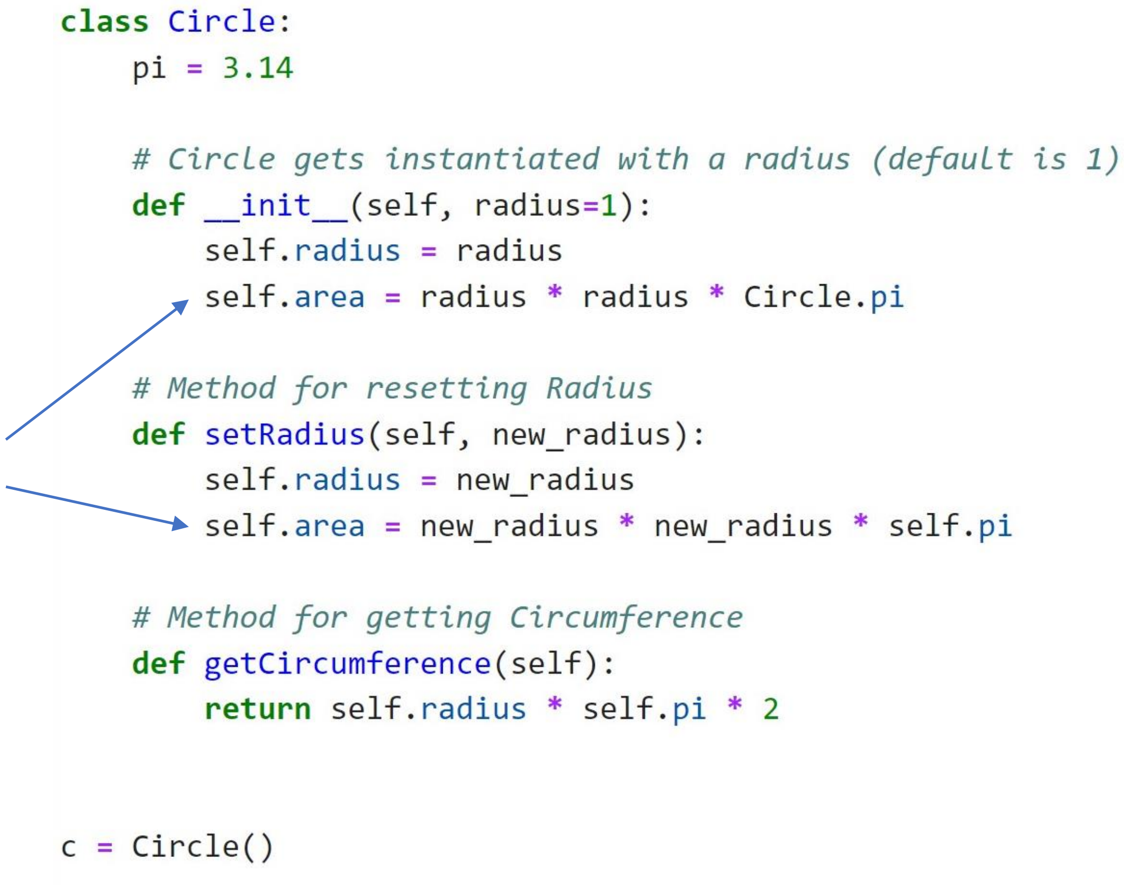
Constructor Method `__init__()`

- 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..

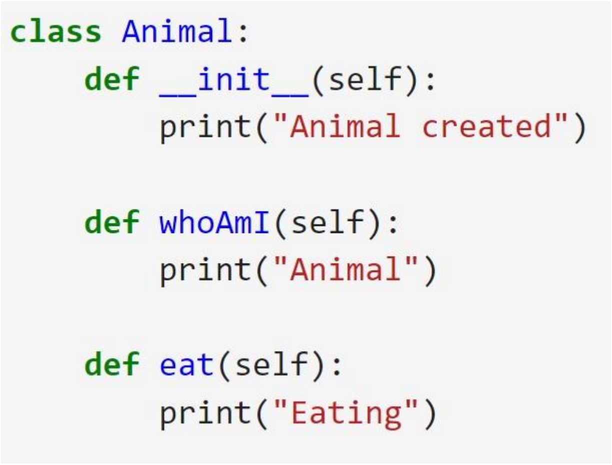


```python
class Circle:
    pi = 3.14

    # 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

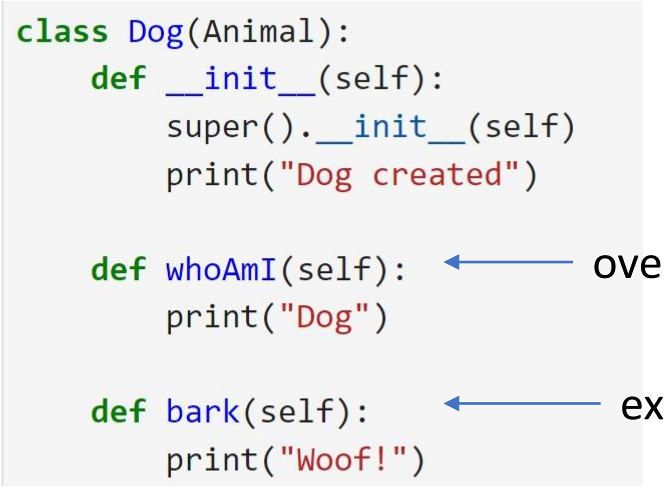
- Inheritance is a way to form new classes using classes that have already been defined.
- The newly formed classes are called derived classes, the classes that we derive from are called base classes.
- Important benefits of inheritance are code reuse and reduction of complexity of a program.
- The derived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



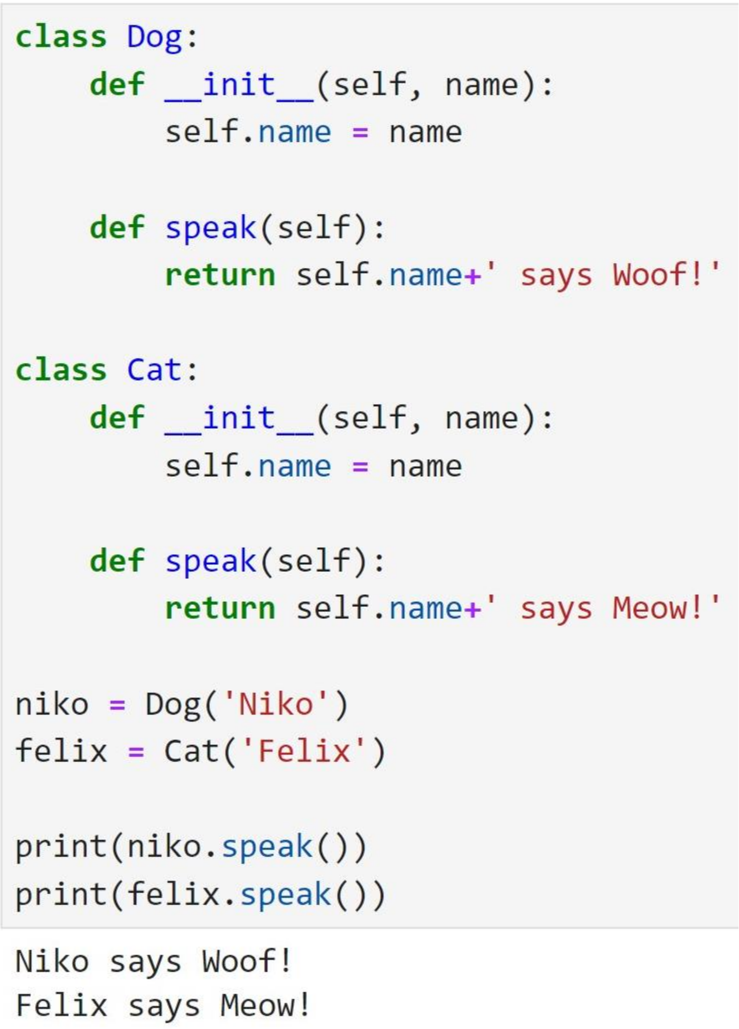
```
class Dog(Animal):
    def __init__(self):
        super().__init__(self)
        print("Dog created")

    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.



```python
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!
```