

Classes

David H Smith IV

University of Illinois Urbana-Champaign

Tues, Nov 30 2021

Reminders

Reminders

- Usual mix of homework (see the website)
- Lab is extended to Dec 6th

The Birds Eye View

Classes, Objects, Instances, oh my!

- Ⓐ **Classes:** The actual Python code that provides instructions on how to build a class (`__init__()`), the attributes in the class, and definitions for the class functions.

Classes, Objects, Instances, oh my!

- Ⓐ **Classes:** The actual Python code that provides instructions on how to build a class (`__init__()`), the attributes in the class, and definitions for the class functions.
 - **Class Attribute:** A value in the class that is accessible to all instances of that class.

Classes, Objects, Instances, oh my!

- Ⓐ **Classes:** The actual Python code that provides instructions on how to build a class (`__init__()`), the attributes in the class, and definitions for the class functions.
- **Class Attribute:** A value in the class that is accessible to all instances of that class.
 - **Instance Attribute:** A value that is only accessible to a given instance.

Classes, Objects, Instances, oh my!

- Ⓐ **Classes:** The actual Python code that provides instructions on how to build a class (`__init__()`), the attributes in the class, and definitions for the class functions.
- **Class Attribute:** A value in the class that is accessible to all instances of that class.
 - **Instance Attribute:** A value that is only accessible to a given instance.
 - **Instance Method:** A function that is callable from within a given instance. The words 'method' and 'function' mean the same thing.

Classes, Objects, Instances, oh my!

- A** **Classes:** The actual Python code that provides instructions on how to build a class (`__init__()`), the attributes in the class, and definitions for the class functions.
- **Class Attribute:** A value in the class that is accessible to all instances of that class.
 - **Instance Attribute:** A value that is only accessible to a given instance.
 - **Instance Method:** A function that is callable from within a given instance. The words 'method' and 'function' mean the same thing.
- B** **Instance:** An object that was created using a given class. We can have multiple instances of the same class.

Classes, Objects, Instances, oh my!

- Ⓐ **Classes:** The actual Python code that provides instructions on how to build a class (`__init__()`), the attributes in the class, and definitions for the class functions.
 - **Class Attribute:** A value in the class that is accessible to all instances of that class.
 - **Instance Attribute:** A value that is only accessible to a given instance.
 - **Instance Method:** A function that is callable from within a given instance. The words 'method' and 'function' mean the same thing.
- Ⓑ **Instance:** An object that was created using a given class. We can have multiple instances of the same class.
- Ⓒ **Object:** Just another name for the thing we instantiated.

Making a Class

Making a Class: Poll Question

Which of the following is the correct way of instantiating the class `Foo`?

```
class Foo:
    def __init__(self):
        print("I'm a class!")
```

- ☐ A `foo = Foo()`
- ☐ B `foo = Foo(self)`
- ☐ C `foo = Foo.__init__()`
- ☐ D `foo = __init__()`

Making a Class: Poll Question

What is the result of the following code?

```
class Foo:
    def bar(self, x, y):
        return x + y

foo = Foo()
x = foo.bar(1, 2)
print(x)
```

- ☐ A SyntaxError
- ☐ B NameError
- ☐ C 3
- ☐ D TypeError

Making a Class: `self`

What is the result of the following code?

```
class Foo:

    def __init__(self):
        print("I'm a class!")

    def get_id(self):
        return id(self)

foo = Foo()
print(id(foo) == foo.get_id())
```

- ☐ A True
- ☐ B False
- ☐ C Trick question

self as an automatic first argument

Given this class...

```
class Foo:
    def __init__(self):
        print("class created!")

    def bar(self, x, y):
        return x + y
```

self as an automatic first argument

Given this class...

```
class Foo:
    def __init__(self):
        print("class created!")

    def bar(self, x, y):
        return x + y
```

This...

```
f = Foo()
x = f.bar(5, 6)
```


`self` as an automatic first argument

Given this class...

```
class Foo:
    def __init__(self):
        print("class created!")

    def bar(self, x, y):
        return x + y
```

This...

```
f = Foo()
x = f.bar(5, 6)
```

Is the same as this...

```
f = Foo()
x = Foo.get_id(f, 5, 6)
```

Classes in General

You've seen this before, you just didn't know it... Any thoughts?

Classes in General

You've seen this before, you just didn't know it... Any thoughts?

```
lst = list()  
s = set()  
d1 = dict()
```

Classes in General

You've seen this before, you just didn't know it... Any thoughts?

```
lst = list()  
s = set()  
d1 = dict()
```

These...

- 1 `lst.append(x)`
- 2 `s.add(x)`
- 3 `d1.update(d2)`

Classes in General

You've seen this before, you just didn't know it... Any thoughts?

```
lst = list()  
s = set()  
d1 = dict()
```

These...

- 1 `lst.append(x)`
- 2 `s.add(x)`
- 3 `d1.update(d2)`

Are the same as these

- 1 `list.append(lst, x)`
- 2 `set.add(s, x)`
- 3 `dict.update(d1, d2)`

Classes in General

You've seen this before, you just didn't know it... Any thoughts?

```
lst = list()  
s = set()  
d1 = dict()
```

These...

- 1 `lst.append(x)`
- 2 `s.add(x)`
- 3 `d1.update(d2)`

Are the same as these

- 1 `list.append(lst, x)`
- 2 `set.add(s, x)`
- 3 `dict.update(d1, d2)`

This is more syntactic sugar brought to you by Python. The key takeaway is that `self` is automatically added in front and refers to the object before the dot (`.`).

Making Classes: Key Takeaways

- 1 Classes are a list of instructions for how to instantiate an object just as functions are a list of instructions on how to perform an operation given some data.

Making Classes: Key Takeaways

- 1 Classes are a list of instructions for how to instantiate an object just as functions are a list of instructions on how to perform an operation given some data.
- 2 Classes are abstract descriptions, objects are concrete and actually exist. Much like function definitions vs function calls().

Making Classes: Key Takeaways

- 1 Classes are a list of instructions for how to instantiate an object just as functions are a list of instructions on how to perform an operation given some data.
- 2 Classes are abstract descriptions, objects are concrete and actually exist. Much like function definitions vs function calls().
- 3 `__init__` Is called when you create a function but is never explicitly called.
 - Example: `foo = Foo()`

Making Classes: Key Takeaways

- 1 Classes are a list of instructions for how to instantiate an object just as functions are a list of instructions on how to perform an operation given some data.
- 2 Classes are abstract descriptions, objects are concrete and actually exist. Much like function definitions vs function calls().
- 3 `__init__` Is called when you create a function but is never explicitly called.
 - Example: `foo = Foo()`
- 4 Self is automatically passed in and refers to the object bound to the variable before the dot.
 - Example: `foo.call_function()`

Making Classes: Key Takeaways

- ❶ Classes are a list of instructions for how to instantiate an object just as functions are a list of instructions on how to perform an operation given some data.
- ❷ Classes are abstract descriptions, objects are concrete and actually exist. Much like function definitions vs function calls().
- ❸ `__init__` Is called when you create a function but is never explicitly called.
 - Example: `foo = Foo()`
- ❹ Self is automatically passed in and refers to the object bound to the variable before the dot.
 - Example: `foo.call_function()`
- ❺ `__init__` is not required. If it is not present in a class definition a default one will be provided and used to instantiate the object.

Attributes and Methods

Poll Question:

Which of the following lines can be used to increment the class attribute count of Name instances that have been instantiated?

```
class Name:
    name_count = 0
    def __init__(self, name):
        ??
        self.name = name
```

```
n1 = Name("foo")
n2 = Name("bar")
n3 = Name("baz")
```

- `Name.name_count += 1`
- `self.name_count += 1`
- `name_count += 1`
- `self.name_count = Name.name_count + 1`
- `Name.name_count = self.name_count + 1`

Consider each and then I'll go through and ask true (A) or false (B) for each.

Poll Question:

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name_count += 1
        self.name = name

n1 = Name("foo")
n2 = Name("bar")
n3 = Name("baz")
print(n1.name_count, n2.
      name_count, n3.name_count)
```

What is the output of the program on the right?

- ☐ A 1 1 1
- ☐ B 1 2 3
- ☐ C 3 3 3
- ☐ D NameError

Scoping in Python Sucks (I Hate it Very Much)

This...

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name_count += 1
        self.name = name
```

Is the same as this.

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name_count = self.name_count + 1
        self.name = name
```

Scoping in Python Sucks (I Hate it Very Much)

This...

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name_count += 1
        self.name = name
```

Is the same as this.

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name_count = self.name_count + 1
        self.name = name
```

Looking at the one on the right:

- 1 Python starts by evaluating the expression on the right.

Scoping in Python Sucks (I Hate it Very Much)

This...

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name_count += 1
        self.name = name
```

Is the same as this.

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name_count = self.name_count + 1
        self.name = name
```

Looking at the one on the right:

- 1 Python starts by evaluating the expression on the right.
- 2 `self.name_count + 1`: `self.name_count` isn't an instance attribute so it resolves to the class attribute.

Scoping in Python Sucks (I Hate it Very Much)

This...

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name_count += 1
        self.name = name
```

Is the same as this.

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name_count = self.name_count + 1
        self.name = name
```

Looking at the one on the right:

- 1 Python starts by evaluating the expression on the right.
- 2 `self.name_count + 1`: `self.name_count` isn't an instance attribute so it resolves to the class attribute.
- 3 `self.name_count = 1`: `self.name_count` doesn't exist as an instance level attribute so scope resolution decides to create a new instance attribute, thus leaving the class attribute unaffected.

Scoping in Python Sucks (I Hate it Very Much)

This...

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name_count += 1
        self.name = name
```

Is the same as this.

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name_count = self.name_count + 1
        self.name = name
```

Looking at the one on the right:

- 1 Python starts by evaluating the expression on the right.
- 2 `self.name_count + 1`: `self.name_count` isn't an instance attribute so it resolves to the class attribute.
- 3 `self.name_count = 1`: `self.name_count` doesn't exist as an instance level attribute so scope resolution decides to create a new instance attribute, thus leaving the class attribute unaffected.

Always use the class name to change class attributes. Bad confusing things happen otherwise.

Poll Question:

```
class Name:
    name_count = 0
    def __init__(self, name):
        Name.name_count += 1
        self.name = name

n1 = Name("foo")
n2 = Name("bar")
n3 = Name("baz")
print(??)
```

Poll Question: The Race Class

```
class Racer:

    finished_list = []

    def __init__(self, name, number):
        self.name = name
        self.number = number

    def finished(self):
        Racer.finished_list.append(self)
        print("finished in", len(Racer.finished_list))
```

Poll Question: The Race Class

```
class Racer:

    finished_list = []

    def __init__(self, name, number):
        self.name = name
        self.number = number

    def finished(self):
        Racer.finished_list.append(self)
        print("finished in", len(Racer.finished_list))
```

What is produced by the following code?

```
r1 = Racer("David", 13)
r2 = Racer("Dipti", 142)
print(Racer.finished_list)
r2.finished()
r1.finished()
print([r.name for r in Racer.finished_list])
```

A

```
[]
finished in 2
finished in 1
['David', 'Dipti ']
```

B

AttributeError

C

```
[]
finished in 1
finished in 2
['Dipti ', 'David ']
```

```
[]
finished in 1
finished in 1
[]
```

Poll Question: Ready-to-Go

```
class ReadyToGo:

    ready = 0
    instances = 0

    def __init__(self, name):
        self.name = name
        self.ready = False
        ReadyToGo.instances += 1

    def set_ready(self):
        ReadyToGo.ready += 1
        self.ready = True
```

Poll Question: Ready-to-Go

```
class ReadyToGo:

    ready = 0
    instances = 0

    def __init__(self, name):
        self.name = name
        self.ready = False
        ReadyToGo.instances += 1

    def set_ready(self):
        ReadyToGo.ready += 1
        self.ready = True
```

What is produced by the following code?

```
p1 = ReadyToGo("Alice")
p2 = ReadyToGo("Bob")
p3 = ReadyToGo("Charlie")
players = [p1, p2, p3]

p1.set_ready()
p3.set_ready()

for player in players:
    if not player.ready:
        print(player.name, "is not ready")
```

- ☐ A Bob is not ready
- ☐ B SyntaxError
- ☐ C NameError
- ☐ D AttributeError
- ☐ E Alice is not ready
Bob is not ready
Charlie is not ready

Modifying Attributes after Instantiation

Poll Question:

What is produced by the following code?

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name = name
        Name.name_count += 1

foo = Name("foo")
bar = Name("bar")
foo.name = "Fred"
print(bar.name, foo.name)
```

- ☐ A AttributeError
- ☐ B NameError
- ☐ C bar foo
- ☐ D bar Fred

Poll Question:

What is produced by the following code?

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name = name
        Name.name_count += 1

foo = Name("foo")
bar = Name("bar")
Name.name_count = 1000
print(bar.name_count)
```

- ☐ A AttributeError
- ☐ B NameError
- ☐ C 1000
- ☐ D 2

Poll Question:

What is produced by the following code?

```
class Name:
    name_count = 0
    def __init__(self, name):
        self.name = name
        Name.name_count += 1

foo = Name("foo")
bar = Name("bar")
foo.name_count = 1000
print(bar.name_count)
```

- ☐ A AttributeError
- ☐ B NameError
- ☐ C 1000
- ☐ D 2

Key Takeaways

① How to reference attributes best practices:

- Reference class attributes using the class name:

```
class Foo:
    count = 0
    def __init__(self):
        Foo.count += 1
```

- Reference instance attributes and methods using *self* *when inside the class*.

```
class Foo:
    def __init__(self, name):
        self.name = name
    def print_name(self):
        print(self.name)
```

- Reference instance attributes and methods using instance's variable name *when outside the class*.

```
class Foo:
    def __init__(self, name):
        self.name = name
x = Foo("bar")
print(x.name)
```

② *self* must be the first parameter in every instance methods arguments.