LAB 07

Object-Oriented Programming

Mingxiao Wei
mingxiaowei@berkeley.edu

Mar 7, 2023

LOGISTICS 

- Lab 07 due this Wed 03/08
- Homework 05 due this Thu 03/09

FROM LAST TIME... 👀

- I forgot to put a random question :((
- But there's one question for me: What is ur fav restuarant close to school
  - Literally any restaurant that sells poke bowl 😋 (Sushi Secret, Poke Bar, Simply Bowl, Poke Parlor)
- Feedback from last lab
  - Some of you liked going over the lab questions together :)
  - If you have more feedback, feel free to leave it in the attendance form or use the anonymous feedback form as well!

GENERATORS 

GENERATOR FUNCTIONS

- Have at least one `yield` or `yield from` statement
- When called, return a generator, which is a special type of iterator, without evaluating the function body
- Allow us to define customized iterators

GENERATORS

- Returned by a generator function
- Calling `next()` on a generator - "lazy evaluation"
  - If it's the first time, begin evaluating the body of the generator function until a value is yielded or the function terminates (by `return`, etc.)
  - Otherwise, pick up from where it stopped last time, and keep evaluating until another value is yielded or the function teriminates
- We can define a generator that yield infinite elements without causing an infinite loop
- Each time a generator function is called, a **new** generator object is returned
- If there are no more elements to be generated, calling `next()` on the generator will raise a `StopIteration` error.

GENERATORS - EXAMPLES

```
def countdown(n):
    print("Beginning countdown!")
    while n >= 0:
        yield n
        n -= 1
    print("Blastoff!")
```

```
>>> c1, c2 = countdown(2), countdown(2)
>>> c1 is iter(c1)  # a generator is an iterator
True
>>> c1 is c2
False
>>> next(c1)
Beginning countdown!
2
>>> next(c2)
Beginning countdown!
2
```

YIELD VS YIELD FROM

- Both `yield` and `yield from` defines elements in the generator
- We can only `yield` one element at a time
- We can `yield from` any iterables (sequences, iterators, generators, etc.)

```
for elem in iterable:
    yield elem
# is equivalent to
yield from iterable
```

- With a for loop and `yield`, we can modify the element before yielding them
- With `yield from`, we can only yield the exact elements from the iterable, without modifying them
- Recursive generators

GENERATORS - MORE EXAMPLES

```
>>> def gen_list (lst):
...     yield from lst
...
>>> g = gen_list([1, 2])
>>> next (g)
1
>>> next (g)
2
>>> next (g)
StopIteration
```

```
>>> def gen_list_1 (lst):
...     for elem in lst:
...         yield elem + 1
...
>>> g = gen_list_1([1, 2])
>>> next (g)
2
>>> next (g)
3
>>> next (g)
StopIteration
```

OBJECT-ORIENTED PROGRAMMING

OBJECTS & CLASSES

- Objects (in real life) can be classified into classes
  - E.g., different types of cars all belong to the same car class
    
- All objects of the same class have some shared characteristics
  - E.g., all cars have wheels, colors, can be driven, etc.
- In order to represent and manipulate objects efficiently, we create classes in Python, aka Object-Oriented Programming
  - Objects keep their own state
    - E.g., a car knows its color, number of wheels
  - Objects have their own behaviors and can be manipulated
    - E.g., a car can be driven
- OOP is also systematic way of implementing data abstraction

OBJECT-ORIENTED PROGRAMMING

OOP - a programming paradigm that allows us to treat code as objects, extending the idea of data abstraction.

- class - a template for objects
- instance - a single object created from a class
- attributes
  - instance variable - an attribute specific to an instance
  - class variable - an attribute of an object, shared by all instances of a class
  - method - a bound function that may be called on all instances of a class
  - Use dot notation to access attributes - `Class.attribute` or `instance.attribute`

OBJECT-ORIENTED PROGRAMMING

```
class Car:
    num_wheels = 4  # class variable, shared by all instances

    def __init__(self, color):  # constructor
        self.wheels = Car.num_wheels
        self.color = color

    def drive(self):  # method
        if self.wheels <= Car.num_wheels:
            return self.color + ' car cannot drive!'
        return self.color + ' car goes vroom!'

    def pop_tire(self):  # method
        if self.wheels > 0:
            self.wheels -= 1
```

TERMINOLOGY

- Attributes = class/instance variables + methods
  - Variables = values (numbers, strings, lists, etc.)
  - Methods = functions defined within a class

	class variable	instance variable
Accessing	`Class.var` or `instance.var` *	`instance.var`
Defining	Within the class, `var = ...`	`instance.var = ...`
Meaning	Shared by all instances of the class	Specific to an instance

```
>>> my_car = Car('red') # an instance of the class
>>> my_car.color # instance variable
'red'
>>> Car.num_wheels, my_car.num_wheels  # both are class variable
4, 4
>>> my_car.wheels # instance variable
4
```

* only works if the instance does not have a instance variable of the same name

MORE TERMINOLOGIES

- Constructors
  - builds an instance of the class
  - define a constructor: `def __init__(self, args):`
  - call a constructor: `ClassName(args)`
  - always returns an *instance* of the class without an explicit return statement

```
class Car :
    def __init__(self, color):
        self.wheels = Car.num_wheels
        self.color = color
my_car = Car('red')  # create an instance of the Car class
```

- `self.var = ...`
  - Initialize an **instance** variable `var` for `self` if it doesn't have an instance variable named `var` yet
  - Otherwise update the instance variable `var` for `self` (objects are mutable!)

MORE TERMINOLOGIES

- Methods
  - Functions defined within a class and bound to an instance
  - Think of them as the "verb" of a class
    - a car can *drive* and *pop their tires*

```
>>> my_car = Car('red')
>>> my_car.drive()
'red car goes vroom!'
>>> my_car.wheels
4
>>> my_car.pop_tire()
>>> my_car.wheels
3
```

MORE TERMINOLOGIES

- self
  - The first parameter for all methods (at least in 61a)
  - When a method is called, e.g., `instance.method(arg)`, `instance` is *implicitly* bound to `self`, and `arg` corresponds to the rest of the parameters

```
def drive(self):
    if self.wheels <= Car.num_wheels:
        return self.color + ' car cannot drive!'
    return self.color + ' car goes vroom!'
```

```
>>> my_car = Car('red')
>>> my_car.drive()
'red car goes vroom!'
```

Though the `drive` takes in one argument `self`, we don't have to pass it in because the dot notation implicitly passes in `my_car` as `self` for us

CALLING A METHOD

Two equivalent ways of calling a method on an instance:

- `instance.method(...)`
  - `instance` is *implicitly* passed in as the first argument and bound to `self`
- `Class.method(instance, ...)`
  - Need to *explicitly* pass in `instance`

```
>>> my_car = Car('red')
>>> my_car.drive()
'red car goes vroom!'
>>> Car.drive(my_car)
'red car goes vroom!'
```

Either way, a method must be called with dot notation, not on its own!

NOW IT'S LAB TIME 

- Get started on the lab and raise your hand whenever you need help!
- Get to know your neighbors and collaborate if you'd like!
- Slides: go.cs61a.org/mingxiao-index
- Leave any anonymous feedback here: go.cs61a.org/mingxiao-anon

AND REMEMBER TO GET CHECKED OFF! 

---

go.cs61a.org/mingxiao-att

The secret phrase is ...
(NOT 3 dots! I'll announce it 🙉)