Discussion 8

DSC 20, Fall 2023

Meme of the Week

PYTHON:
ME:
ERROR IN LINE 5
ME SEEING LINE 5 IS PERFECTLY CORRECT, BUT LINE 4 HAD A BRACKET MISSING
ME:

Announcement

There will be a substitute for me this week - I will be out of town for work

Discussion participation will be in the form of a gradescope quiz again - make sure to get answers right, your attendance will be contingent on your score

Agenda

- Classes (again)
- Inheritance
- super
- Multiple Inheritance

Classes

Classes aggregate code together into a functional body. A lot of effective python code is written as classes (every time you install a new package, it's basically written as a lot of different classes). For example, recall the pandas dataframe. If you look at the source code for pandas, you'll see that it's a very complicated class definition.

Classes all generally have (at the minimum) a constructor (__init__) function and other object related methods that are used.

In [1]:
```
class phone:
    """
    Class representation of a regular phone.
    """
    pass
```

Constructors

- denoted by __init__ (special function)
- should ingest self as the first parameter (ALWAYS)
- populates instance with attributes

Instances

When an object is created from a class, each individual unit is referred to as an instance. Think of it as "one of x" (for example, an instance of the iPhone class is "1/my iPhone")

Working with self

- Think of classes as a blueprint and everytime you create an instance of one, you've produced a "physical" manifestation.
- We use the self keyword to interact with THIS specific instance of the class.
- Try to think of 'self' as referencing a **specific, singular** instance of the class. Every method that works with a specific instance's values must be passed in self as an argument.

```
In [2]: class phone:
            """
            Class representation of a regular phone.
            """
            def __init__(self, maker, version, owner):
                self.maker = maker
                self.version = version
                self.owner = owner

                self.apps = []
                self.free_memory = 800
```

```
In [3]: iphone = phone('Apple', 'XR', 'Nikki')
        pixel = phone('Google', 'P3', 'Sailesh')
        print(iphone.maker)
        print(iphone.owner)
        print(iphone.apps)
        print()
        print(pixel.maker)
        print(pixel.owner)
        print(pixel.apps)
```

```
Apple
Nikki
[]

Google
Sailesh
[]
```

Instance vs Class variables

Instance variables are attached to instances of a class by keyword self (usually done in the constructor). Class variables are variables attached to the class itself.

```
In [4]: class phone:
            """
            Class representation of a regular phone.
            """
            id_num = 1 # class variable
            o_type = 'phone' # class variable
            def __init__(self, maker, version, owner):
                self.maker = maker
                self.version = version
                self.owner = owner
                self.id = phone.id_num
                phone.id_num+=1

                self.apps = []
                self.free_memory = 800
```

```
In [5]: iphone = phone('Apple', 'XR', 'Nikki')
        pixel = phone('Google', 'P3', 'Sailesh')
        print(iphone.id)
        print(iphone.o_type)
        print()
        print(pixel.id)
        print(pixel.o_type)
```

```
1
phone

2
phone
```

(Class) Methods

```
In [6]: class phone:
            """
            Class representation of a regular phone.
            """
            id_num = 1
            def __init__(self, maker, version, owner):
                self.maker = maker
                self.version = version
                self.owner = owner
                self.id = phone.id_num
                phone.id_num+=1

                self.apps = []
                self.free_memory = 800

            def install_app(self, app, memory):
                if self.free_memory - memory >= 0:
                    self.apps.append(app)
                    self.free_memory -= memory
                    return True
                return False
```

```
In [7]: iphone = phone('Apple', 'XR', 'Nikki')
        print(iphone.install_app('duolingo', 600))
        print(iphone.install_app('genshin impact', 1000))
        print(iphone.apps)
```

```
True
False
['duolingo']
```

Inheritance

- classes can have a "parent-child" relationship
- child class(es) "inherit" methods from their parent class
- use "is-a" paradigm to distinguish; given a parent class phone and a child class pearphone:
    - pearphones are phones (child is a parent)
    - but not all phones are pearphones (parents are not children)

```
In [8]: class pearphone(phone):
            pass
            # since nothing is changed in my pearphone class,
            # it is currently just another name for phone class
```

```
In [9]: pear = pearphone('pear', '9', 'Tim Raw')
        print(isinstance(pear, pearphone))
        print(isinstance(pear, phone))
        print(isinstance(iphone, pearphone))
        print(isinstance(iphone, phone))
```

```
True
True
False
True
```

Inheritance (cont.)

- Though not required, child classes can **overwrite** methods of their parent class
- if certain things are to be retained, super() is a useful function to use

note: super refers to the direct parent of a class

In [10]:
```
class pearphone(phone):
    def __init__(self, maker, version, owner, age):
        self.maker = maker
        self.version = version
        self.age = age
        self.id = phone.id_num
        phone.id_num+=1
        # notice how I can still use ID from phone!

        self.apps = []
        self.free_memory = 800
```

```
In [11]: pear = pearphone('pear', '9', 'Tim Raw', 2)
         print(f'age of pear: {pear.age}')
         print(pear.install_app('duolingo', 600))
         print(pear.apps)
         print(f'pear id number: {pear.id}')
```

```
age of pear: 2
True
['duolingo']
pear id number: 3
```

super()

In pearphone, even though we want a different constructor, you can see that we actually reused a lot of the same code as the constructor for its parent class, phone. To avoid this, we can directly access the parent method using super(), and then add parameters we need.

explore super() more on your own :)

```
In [12]: class pearphone(phone):
             def __init__(self, maker, version, owner, age):
                 # self is implicitly passed with super()
                 super().__init__(maker, version, owner)
                 self.age = age

                 self.apps = []
                 self.free_memory = 800
```

```
In [13]: pear = pearphone('pear', '9', 'Tim Raw', 2)
         print(f'age of pear: {pear.age}')
         print(pear.install_app('duolingo', 600))
         print(pear.apps)
         print(f'pear id number: {pear.id}')
```

```
age of pear: 2
True
['duolingo']
pear id number: 4
```

Overwriting inherited methods

In [14]:
```python
class pearphone(phone):
    def __init__(self, maker, version, owner, age):
        # self is implicitly passed with super()
        super().__init__(maker, version, owner)
        self.age = age

        self.apps = []
        self.free_memory = 800

    def install_app(self, app, memory):
        '''install_app but better (whoo pear phones!)'''
        if self.free_memory - memory//2 >= 0:
            self.apps.append(app)
            self.free_memory -= memory//2
            return True
        return False
```

In [15]:
```python
pear = pearphone('pear', '9', 'Tim Raw', 2)
print(f'pear id number: {pear.id}')
print(pear.install_app('duolingo', 1600))
```

```
pear id number: 5
True
```

Using super...

```
In [16]: class pearphone(phone):
             def __init__(self, maker, version, owner, age):
                 # self is implicitly passed with super()
                 # if you pass inself again, will result in an error
                 super().__init__(maker, version, owner)
                 self.age = age

                 self.apps = []
                 self.free_memory = 800

             def install_app(self, app, memory):
                 '''install_app but cheaper (whoo pear phones!)'''
                 return super().install_app(app,memory//2)
```

```
In [17]: pear = pearphone('pear', '9', 'Tim Raw', 2)
         print(f'pear id number: {pear.id}')
         print(pear.install_app('duolingo', 1600))
```

```
pear id number: 6
True
```

Multiple Inheritance

- a class can inherit from multiple parent classes
- the order you list classes determines the order in which methods are inherited
- can get very messy, good luck!

In [19]:
```
class black:
    def __init__(self, name):
        self.name = name
    def fizz(self, num):
        return num
    def buzz(self):
        return 'something idk'
class berry:
    def __init__(self, name):
        self.name = name
    def fizz(self, num):
        return num*2
    def foo(self, string):
        return string*2

class blackberry(black, berry):
    def __init__(self, name):
        self.name = name
    def buzz(self):
        return 'not idk'
```

```
In [27]: phone = blackberry('bankrupt')
         print(f'fizz result is: {phone.fizz(4)}')
         print(f'buzz result is: {phone.buzz()}')
         print(f'foo result is: {phone.foo("gobble")}')
```

```
fizz result is: 4
buzz result is: not idk
foo result is: gobblegobble
```

fizz is inherited from class black because it was first in the order of classes passed for inheritance (even though the same method exists in class berry).

buzz was overwritten and foo was inherited from class berry.

Thanks for coming!

Make sure to complete the quiz, I will not be giving grace periods (there is no ID to forget)