





Scripting Languages

Laszlo SZATHMARY University of Debrecen Faculty of Informatics

Lab #7

classes, objects

(last update: 2024-01-19 [yyyy-mm-dd])



2023-2024, 2nd semester

OO programming in Python



In Python you can program in a procedural, or in an OO way. You can choose which one to use: either this or that, or even both.

We have already used Python classes, e.g. str (string class).

```
name = "john"
print(name.capitalize())
```

Now let's see how to define own classes, and how to instantiate objects from these classes.

OO programming in Python (cont.)



All standard OO features can be found in Python. For instance:

- multiple inheritance
- a subclass can override any method of its superclass

It's a dynamic language, thus classes are created during runtime, and once they are created, they can be modified!

All instance variables and instance methods are *public*.

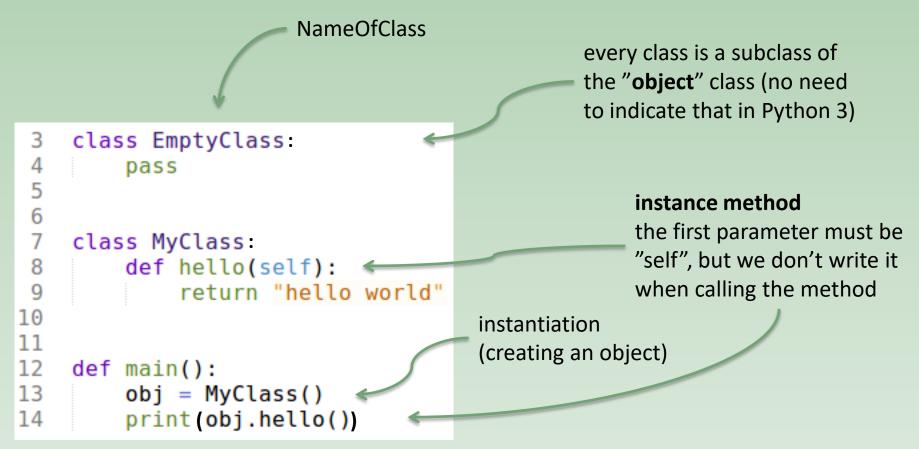
All instance methods are virtual.

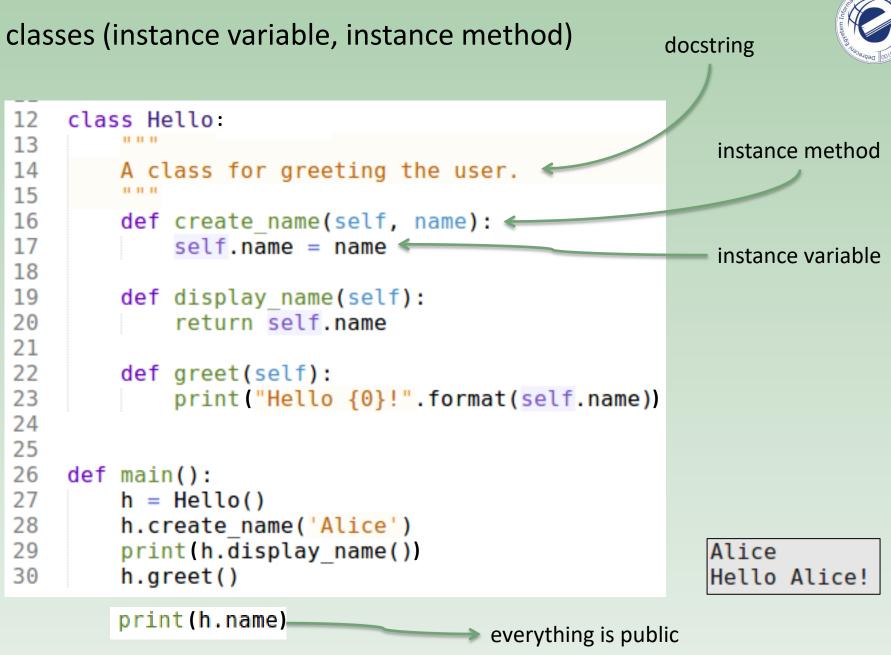
Most built-in operators can be overloaded (redefined) and then they can be used with the objects.

Passing an object as a parameter is cheap, since their addresses are passed (as a reference). Consequence: if we modify an object that we got via parameter passing, then the caller will also see the changes.

classes









The first parameter of every instance method must be "self". This is equivalent to Java's "this", i.e. it's a reference that points to the current object. By convention it's called "self". Don't change its name!

Every (non-static) function's first parameter is "self", but don't indicate this when you call the function!

Python's dynamic nature allows us to to create an instance variable in any function, and then this variable exists from that point on.

classes (init)



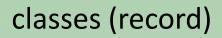
```
class Greetings
 3
 4
 5
        def init (self, name):
6
7
            self.name = name
8
        def say hi(self):
9
            print("Hi {0}!".format(self.name))
10
11
12
    def main():
13
        g = Greetings("Alice")
14
        g.say hi()
```

The constructor automatically calls the __init__() method. Technically, __init__() is not the constructor, but it's very close to it. It will initialize the object.

classes (calling an instance method)



container class class Bag: 3 (its instances store data) 4 5 def init (self): 6 self.data = [] 7 8 def add(self, value): 9 self.data.append(value) 10 11 def add twice(self, value): 12 self.add(value) 13 self.add(value) 14 special method 15 def str (self): (produces a readable 16 return str(self.data) 17 representation of the object) 18 19 def main(): see also: Java's toString() 20 b = Bag()21 b.add(5)22 print(b) *Try it without the special method too!* 23 b.add(3) 24 print(b) 25 b.add twice(9) print(b) 26





Sometimes it'd be nice to have a **record** type, similar to C's struct. It can be done:

3	class Employee:
4	pass
5	
6	def main():
7	john = Employee()
8	john.name = "John Doe"
9	john.dept = "IT"
10	john.salary = 1000
11	
12	<pre>print(john.dept)</pre>

```
Another method: use a dictionary
john = { }
john['name'] = "John Doe"
```

• • •

9

private variable and methods



Private variables/methods that are not accessible from outside just inside the object: they don't exist in Python. Everything is public.

However, there is a convention (again): if the name of a variable/method starts with _ (underscore), then it must be treated as if it were non-public. Example: _spam.

accessors (getters / setters)

Not needed, everything is public.

Once Guido was asked why there are no private variables/methods. Guido's answer: "We are all adults." :)

accessors (getters / setters)

Java style

def init (self, width, height):

self. width = width

return self. width

def get width(self):

def get height(self):

self. height = height

def set width(self, new width):

self. width = new width

class Rectangle:



Python style

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
```

```
def main():
    rect = Rectangle(50, 10)
    rect.width = 60
    print(rect.area())
```

Exercise: extend this source

print(rect) # should produce this output:
-> "Rectangle(60, 10)"

```
return self._height

def set_height(self, new_height):
    self._height = new_height

def area(self):
    return self._width * self._height
```

```
def main():
    rect = Rectangle(50, 10)
    rect.set_width(60)
    print(rect.area())
```

special methods



Their names start and end with ____ (double underscore, "dunder"). We have already seen some:

- ____init____
- •__________

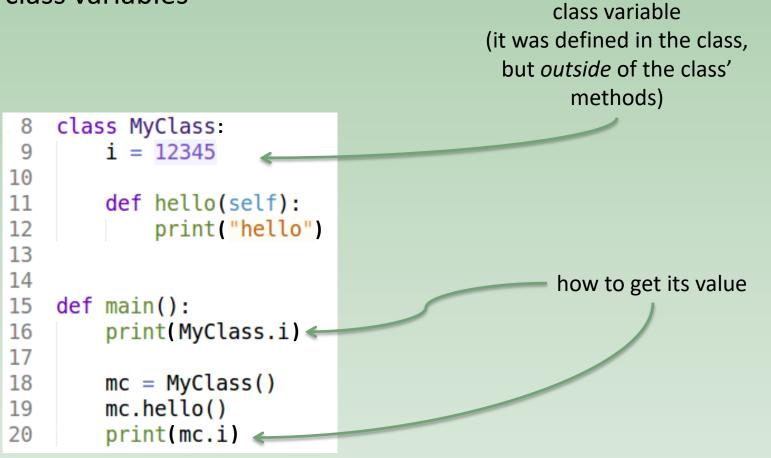
There are several other special methods, see https://rszalski.github.io/magicmethods/.

destructor

Doesn't exist. The garbage collector will delete the object. However, we don't know exactly when this happens.

class variables





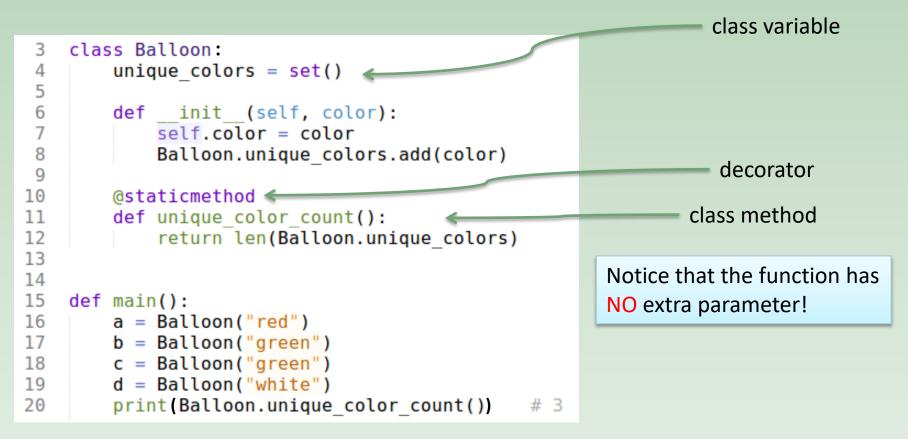
Exercise:

Write a class that counts how many times it was instantiated (how many objects were created from it).

class methods (1st way)



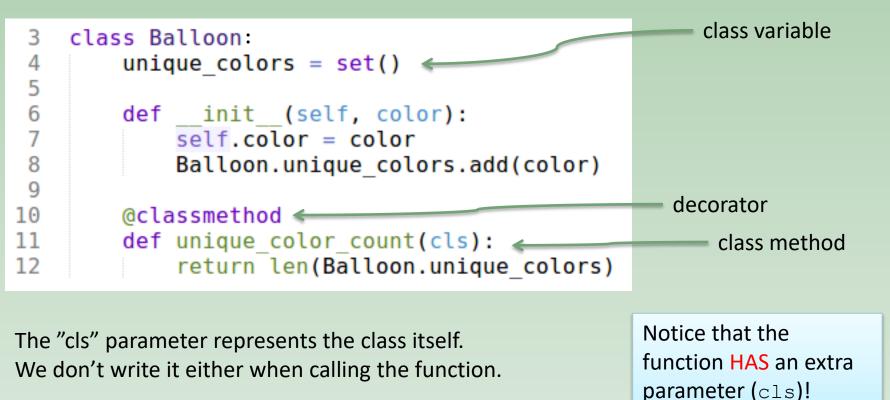
Write a Balloon class, that represents colored balloons. Keep track of the *number of the different colors* of the balloons too. (For instance, if we have 2 red, 1 white, and 5 green balloons, then the number of different colors is three.)



This static function could also be outside the class. We put it in the class because logically it belongs there.

class methods (2nd way)





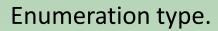
Use this 2nd way when you want to refer to the current class in the function. It can be necessary upon inheritance.

inheritance, multiple inheritance



Python supports multiple inheritance. However, it's better to avoid it (see <u>diamond problem</u>). It was also removed from Java...

Enum





```
from enum import Enum
3
 4
5
   class Direction(Enum):
       UP = 1
6
       RIGHT = 2
7
                                     class variables
8
       DOWN = 3
9
       LEFT = 4
10
11
   def main():
12
       print(Direction.UP)
                              # Direction.UP
       print(type(Direction.UP)) # <enum 'Direction'>
13
14
       print(Direction.UP.name) # "UP" (str)
       print(Direction.UP.value)
15
                                  # 1 (int)
```

Enum (cont.)



Enumeration type.

```
from enum import Enum, auto
 3
 4
   class Direction(Enum):
 5
       UP = auto()
 6
       RIGHT = auto()
 7
       DOWN = auto()
 8
        LEFT = auto()
 9
10
11
   def main():
12
        print(Direction.UP)
                               # Direction.UP
        print(type(Direction.UP))
13
                                   # <enum 'Direction'>
        print(Direction.UP.name)
14
                                    # "UP" (str)
        print(Direction.UP.value)
15
                                    # 1 (int)
```





Exercises

- 1. [20141125a] classes (stack)
- 2. [20130325a] classes (queue with two stacks)