Scripting Languages

Laszlo SZATHMARY
University of Debrecen
Faculty of Informatics

Lab #7

- classes, objects

(last update: 2025-08-27 [yyyy-mm-dd])

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

NameOfClass

every class is a subclass of the ”**object**” class (no need to indicate that in Python 3)

```
3   class EmptyClass:
4       pass
5
6
7   class MyClass:
8       def hello(self):
9           return "hello world"
10
11
12  def main():
13      obj = MyClass()
14      print(obj.hello())
```

instance method
the first parameter must be ”self”, but we don’t write it when calling the method

instantiation
(creating an object)

classes (instance variable, instance method)

```
12  class Hello:
13      """
14      A class for greeting the user.
15      """
16      def create_name(self, name):
17          self.name = name
18
19      def display_name(self):
20          return self.name
21
22      def greet(self):
23          print("Hello {0}!".format(self.name))
24
25
26  def main():
27      h = Hello()
28      h.create_name('Alice')
29      print(h.display_name())
30      h.greet()
```

docstring

instance method

instance variable

```
Alice
Hello Alice!
```

```
print(h.name)
```

everything is public

self

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)

```
 3  class Greetings:
 4
 5      def __init__(self, name):
 6          self.name = name
 7
 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)

```
 3  class Bag:
 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
15      def __str__(self):
16          return str(self.data)
17
18
19  def main():
20      b = Bag()
21      b.add(5)
22      print(b)
23      b.add(3)
24      print(b)
25      b.add_twice(9)
26      print(b)
```

container class
(its instances store data)

special method
(produces a readable representation of the object)

see also: Java's `toString()`

Try it without the special method too!

classes (record)

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      print(john.dept)
```

Another method: use a dictionary

```
john = {}
john['name'] = "John Doe"
…
```

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

```
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    def get_width(self):
        return self._width

    def set_width(self, new_width):
        self._width = new_width

    def get_height(self):
        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())
```

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)”

special methods

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

- `__init__`
- `__str__`

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

class variable
(it was defined in the class,
but *outside* of the class'
methods)

```
 8  class MyClass:
 9      i = 12345
10
11      def hello(self):
12          print("hello")
13
14
15  def main():
16      print(MyClass.i)
17
18      mc = MyClass()
19      mc.hello()
20      print(mc.i)
```

how to get its value

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.)

```
 3  class Balloon:
 4      unique_colors = set()
 5
 6      def __init__(self, color):
 7          self.color = color
 8          Balloon.unique_colors.add(color)
 9
10      @staticmethod
11      def unique_color_count():
12          return len(Balloon.unique_colors)
13
14
15  def main():
16      a = Balloon("red")
17      b = Balloon("green")
18      c = Balloon("green")
19      d = Balloon("white")
20      print(Balloon.unique_color_count())    # 3
```

class variable

decorator

class method

Notice that the function has NO extra parameter!

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

class methods (2nd way)

```
3  class Balloon:
4      unique_colors = set()
5
6      def __init__(self, color):
7          self.color = color
8          Balloon.unique_colors.add(color)
9
10     @classmethod
11     def unique_color_count(cls):
12         return len(Balloon.unique_colors)
```

class variable

decorator

class method

The ”cls” parameter represents the class itself.
We don’t write it either when calling the function.

Notice that the function HAS an extra parameter (`cls`)!

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 diamond problem). It was also removed from Java…

Enum

Enumeration type.

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

class variables

Enum (cont.)

Enumeration type.

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

Exercises

homework

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