

Introducción a las ciencias de la computación *y programación en Python*

Programación orientada a objetos

Rodrigo Chang

Banco de Guatemala



Rodrigo Chang <rrcp@banguat.gob.gt>
Este material está construido a partir de modificaciones al material provisto por Ana Bell, Eric Grimson y John Guttag para el curso 6.0001 *Introduction to Computer Science and Programming in Python*, otoño 2016, Massachusetts Institute of Technology: MIT OpenCourseWare, <https://ocw.mit.edu>. Licencia: Creative Commons BY-NC-SA.

Abstract

“Commenting your code is like cleaning your bathroom — you never want to do it, but it really does create a more pleasant experience for you and your guests.”

— Ryan Campbell

Veremos conceptos básicos y ejemplos del paradigma de programación orientada a objetos.

Objetos

- Como vimos antes, Python representa información de diferentes formas:

```
1234      3.14159      "Hello"  
[2, 4, 6, 8, 10]  
{ "GT": "Guatemala", "HN": "Honduras" }
```

- Cada uno es un **objeto**, el cual tiene:
 - Un **tipo**.
 - Una **representación interna** (primitiva o compuesta).
 - Un conjunto de procedimientos de **interacción** con el objeto.
- Un objeto es una **instancia** de un tipo:
 - 1234 es una instancia de un entero : **int**.
 - "hello" es una instancia de un *string*: **str**.

Programación orientada a objetos

- En Python, ¡**TODO ES UN OBJETO!**
- Podemos **crear nuevos objetos** de algún tipo.
- Podemos **manipular objetos**.
- Podemos **destruir objetos**.
 - Utilizando **del** o “dejándolos” a un lado¹.

¹Python utiliza un sistema de recolección de memoria para objetos destruidos o inaccesibles, llamado “recolección de basura”.

Ya en serio: ¿qué son los objetos?

- Representan una **abstracción de datos** que captura:
- Una **representación interna**:
 - A través de *atributos*.
- Una **interfaz** para interactuar con el objeto:
 - A través de *métodos* (alias procedimientos/funciones)
 - Que definen el comportamiento, pero **ocultan detalles de implementación**.

Ejemplo: [1,2,3,4] es de tipo lista

- how are lists **represented internally**? linked list of cells



*follow pointer to
the next index*

- how to **manipulate** lists?
 - `L[i]`, `L[i:j]`, `+`
 - `len()`, `min()`, `max()`, `del(L[i])`
 - `L.append()`, `L.extend()`, `L.count()`, `L.index()`,
`L.insert()`, `L.pop()`, `L.remove()`, `L.reverse()`, `L.sort()`
- internal representation should be private
- correct behavior may be compromised if you manipulate internal representation directly

Ventajas de la OOP

- Permiten **empaquetar** datos y procedimientos para trabajar sobre estos con interfaces bien definidas.
- Desarrollo al estilo “**divide y conquistarás**”
 - Implementación y pruebas sobre cada clase separada.
 - Incrementan la modularidad y reducen la complejidad.
- Permiten **reutilizar** el código fácilmente.
 - Muchos módulos definen nuevas clases.
 - Cada clase tiene un ambiente separado (no hay problemas de nombres de funciones).
 - La **herencia** permite redefinir o extender el comportamiento de una clase padre.

Las clases implementan tipos propios

- Existe una diferencia entre **crear una clase** e **instanciar una clase**.
- **Crear** una clase involucra
 - Definir el nombre de la clase.
 - Definir los atributos.
 - Por ejemplo: *alguien definió la clase `list`, sus atributos y métodos.*
- **Utilizar** la clase involucra:
 - Crear una nueva **instancia** del objeto.
 - Realizar operaciones con la instancia.
 - Por ejemplo: `L = [1, 2]` y `len(L)`.

Definición de una clase

- Para definir un nuevo tipo, utilizamos la palabra `class`:

```
class Coordinate(object):  
    #define attributes here
```

- Similar a `def`, `indentamos` para indicar qué elementos pertenecen a la clase.
- `object` se refiere a la `clase padre` de `Coordinate`.
 - `Coordinate` es una subclase de `object`.
 - `object` es una superclase de `Coordinate`.

¿Qué son atributos?

- Datos y funciones que **pertenecen** a la clase.
- **Atributos de datos**
 - Objetos de datos que componen la clase.
 - Por ejemplo: en `Coordinate` un atributo es la coordenada `x`.
- **Métodos**
 - Funciones que solamente funcionan con esta clase.
 - Permiten **interactuar** con el objeto.
 - En `Coordinate` podríamos definir un método `distance` que nos devuelva la distancia hacia otro objeto `Coordinate`.
 - Notar que no necesariamente habría significado de `distance` entre dos objetos `list`.

El método constructor

- Primero debemos definir **cómo crear** una instancia de la clase.
- A este método le llamamos **constructor**. En Python, definimos la función `__init__(self, ...)`.

```
class Coordinate(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

Instanciando una clase

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x, origin.x)
```

- Los atributos de datos de una instancia se convierten en **variables de la instancia**.
- Notar que:
 - No proveemos argumento `self` \Rightarrow automático.
 - Utilización del `.` para acceder a los atributos/métodos.
 - Al llamar a `Coordinate`, invocamos la función `__init__`.

¿Qué es un método?

- Es también un atributo, pero **solo funciona con esta clase**.
- Python siempre pasa el objeto como primer argumento:
 - Utilizamos `self` como el primer argumento en cualquier método.
- **El operador `.`** permite acceder a cualquier atributo.
 - Llamar a un método del objeto.
 - Acceder a los datos almacenados en el objeto.

Nuestro primer método

```
class Coordinate(object):
    """ A coordinate made up of an x and y value """
    def __init__(self, x, y):
        """ Sets the x and y values """
        self.x = x
        self.y = y
    def distance(self, other):
        """ Returns the euclidean distance between two
        points """
        x_diff_sq = (self.x - other.x)**2
        y_diff_sq = (self.y - other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

- Aparte de `self` y la notación `.`, los métodos **se comportan igual que las funciones**.
 - Toman parámetros.
 - Realizan cálculos, procedimientos.
 - Devuelven valores.

¿Qué asume el método `distance`?

Cómo utilizar un método

- Forma convencional: lo invocamos sobre un objeto.

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

- Notar que se omite `self`, pues está implicado por el objeto `c`.

- Equivalente a:

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.
      distance(c, zero))
```

- Se operan los dos objetos entre sí invocando el método **desde la clase** y no desde una instancia.

```
c = Coordinate(3,4)
print(c)
# Imprime: <__main__.Coordinate object at 0x7fa918510488>
```

- Representación **poco informativa** por defecto.
- Podemos definir un método `__str__` para una clase.
- Cuando utilizamos `print` sobre un objeto, Python llama a su método `__str__`.
- Podemos escoger qué mostrar, supongamos que queremos:

```
print(c)
# Queremos: <3, 4>
```


Definición propia para `print`

```
class Coordinate(object):
    """ A coordinate made up of an x and y value """
    def __init__(self, x, y):
        """ Sets the x and y values """
        self.x = x
        self.y = y
    def distance(self, other):
        """ Returns the euclidean distance """
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        """ Returns a string representation of self """
        return "<"+str(self.x)+", "+str(self.y)+">"
```

¿Qué tipo es devuelto por `__str__`?

Acerca de los tipos y las clases

- can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
```

```
>>> print(c)
```

```
<3,4>
```

```
>>> print(type(c))
```

```
<class __main__.Coordinate>
```

return of the `__str__` method
the type of object `c` is a class `Coordinate`

- this makes sense since

```
>>> print(Coordinate)
```

```
<class __main__.Coordinate>
```

```
>>> print(type(Coordinate))
```

```
<type 'type'>
```

a `Coordinate` is a class
a `Coordinate` class is a type of object

- use `isinstance()` to check if an object is a `Coordinate`

```
>>> print(isinstance(c, Coordinate))
```

```
True
```

Operadores especiales

- `+`, `-`, `==`, `<`, `>`, `len()`, `print`, and many others

<https://docs.python.org/3/reference/datamodel.html#basic-customization>

- like `print`, can override these to work with your class
- define them with double underscores before/after

<code>__add__(self, other)</code>	→	<code>self + other</code>
<code>__sub__(self, other)</code>	→	<code>self - other</code>
<code>__eq__(self, other)</code>	→	<code>self == other</code>
<code>__lt__(self, other)</code>	→	<code>self < other</code>
<code>__len__(self)</code>	→	<code>len(self)</code>
<code>__str__(self)</code>	→	<code>print self</code>

... and others

Ejemplo: fracciones

- Crearemos un **nuevo tipo** para representar fracciones.
- Su **representación interna** serán dos enteros:
 - Numerador.
 - Denominador.
- Definiremos una interfaz (métodos) que permitirán:
 - Sumar, restar.
 - Imprimir la fracción, convertir a flotante.
 - Obtener el recíproco.

Veamos el código

.

El poder de OOP

- Podemos **empaquetar objetos** que comparten:
 - Atributos comunes.
 - Procedimientos que operan sobre esos atributos.
- Utilizamos **abstracción** para diferenciar cómo implementar un objeto y cómo utilizarlo.
- Podemos construir **capas** de abstracción de objetos que **hereden** comportamiento de otras clases de objetos.
- Podemos crear **nuestras propias clases de objetos** utilizando las clases base de Python.