Programación orientada a objetos (OOP)

Lic. Rodrigo Lugones

rlugones@df.uba.ar

¿Qué es un *struct*?

Un *struct* es una colección de variables. Son accesibles desde un único puntero. Internamente están contiguos en memoria.

Por ejemplo:

```
struct product {
  int weight;
  double price;
  string name;
} ;

product apple;
product banana, melon;

apple.price  = 10;
```

Programación orientada a objetos - OOP

en Python

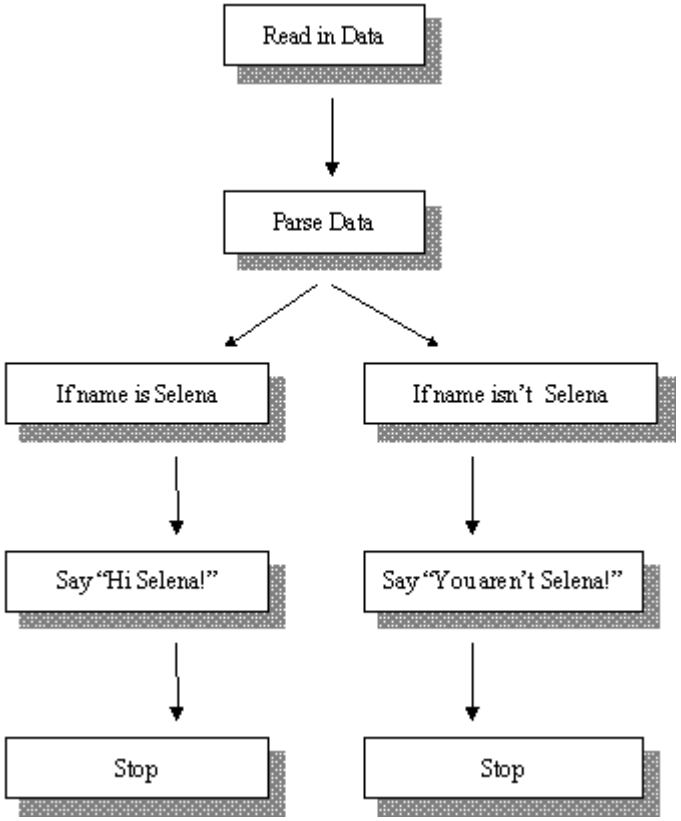
Los **objetos** son abstracciones de Python para referirse a los datos. Todos los datos en un programa de Python son representados por objetos o por relaciones entre objetos.

*https://docs.python.org/2/reference/datamodel.html#

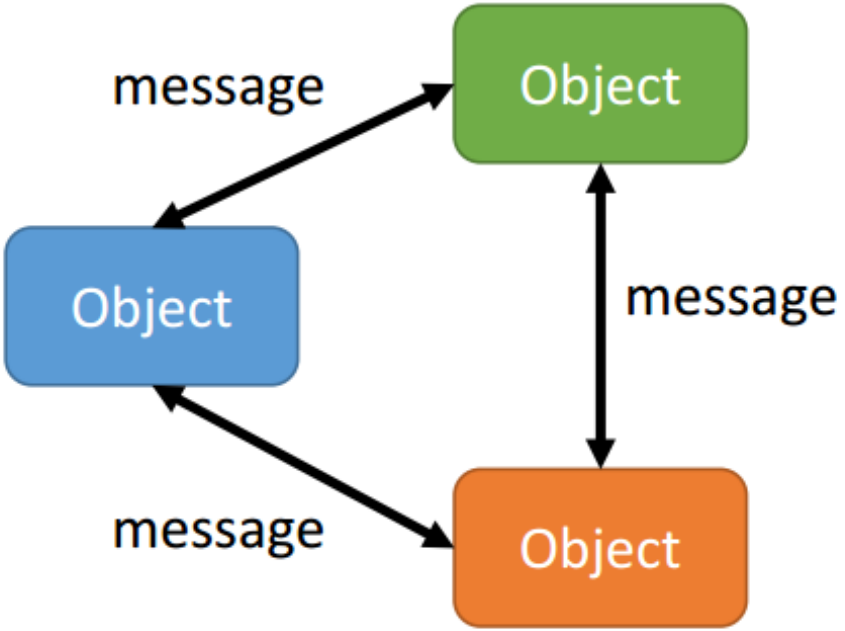
WTPC
Workshop en técnicas de programación científica

Workflow procedural

1) input
2) procesamiento
3) output



Workflow en objetos



Clase

Un constructor de objetos. Es el plano (*blueprint*) para construir objetos.

Estado

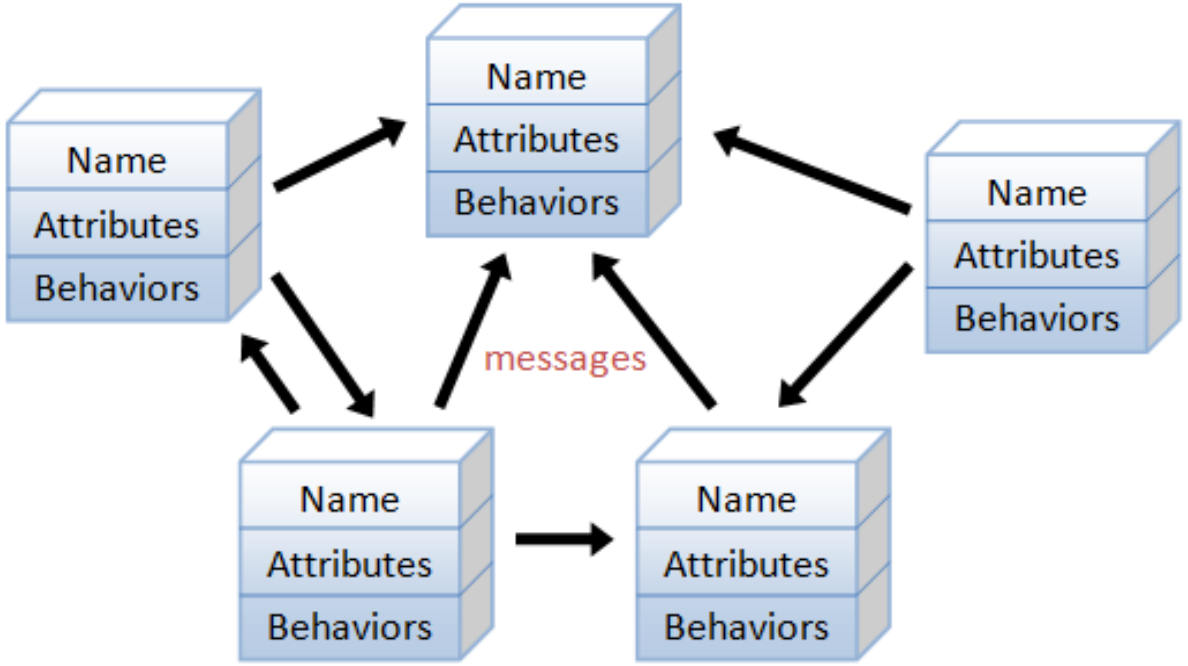
Todas las propiedades de un objeto

Comportamiento

Cómo un objeto reacciona frente a una interacción. Esto se logra llamando a ciertos métodos. En OOP es la manera en la que responde a ciertos mensajes.

Identidad

Distintos objetos pueden tener idénticos estados y el mismo comportamiento, pero cada uno tendrá su identidad.


Name
Attributes
Behaviors
Name
Attributes
Behaviors
Name
Attributes
Behaviors
messages
Name
Attributes
Behaviors
Name
Attributes
Behaviors

Programación orientada a Objetos

Composición

Encapsulamiento

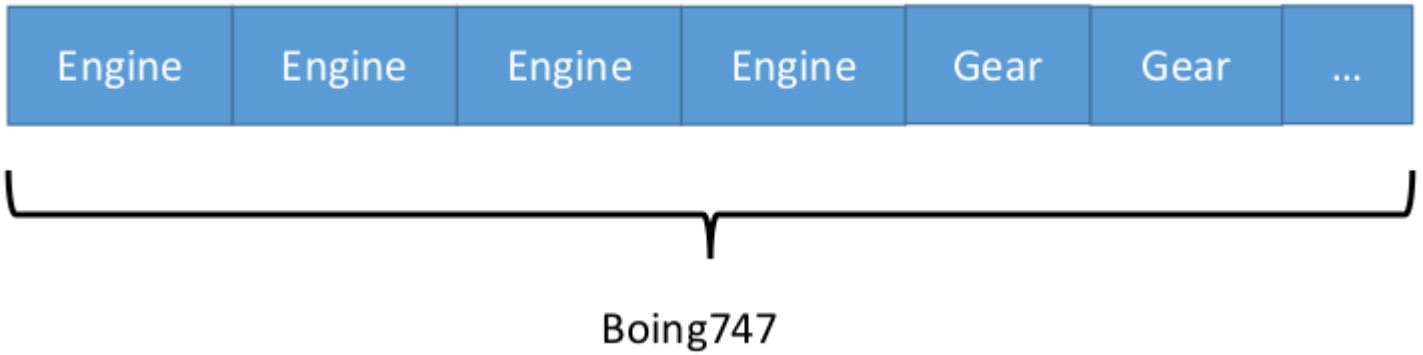
Herencia

Polimorfismo

Composición

- La forma natural de crear objetos es construyéndolos a partir de objetos ya existentes.

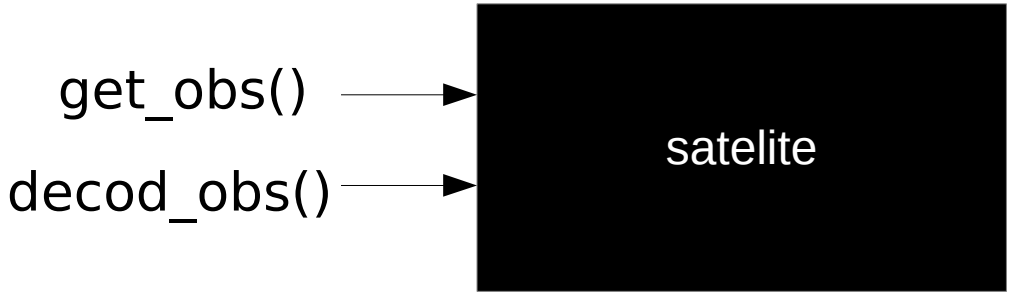
- De esta manera, un sistema complejo se compone te subsistemas más simples.



Encapsulamiento

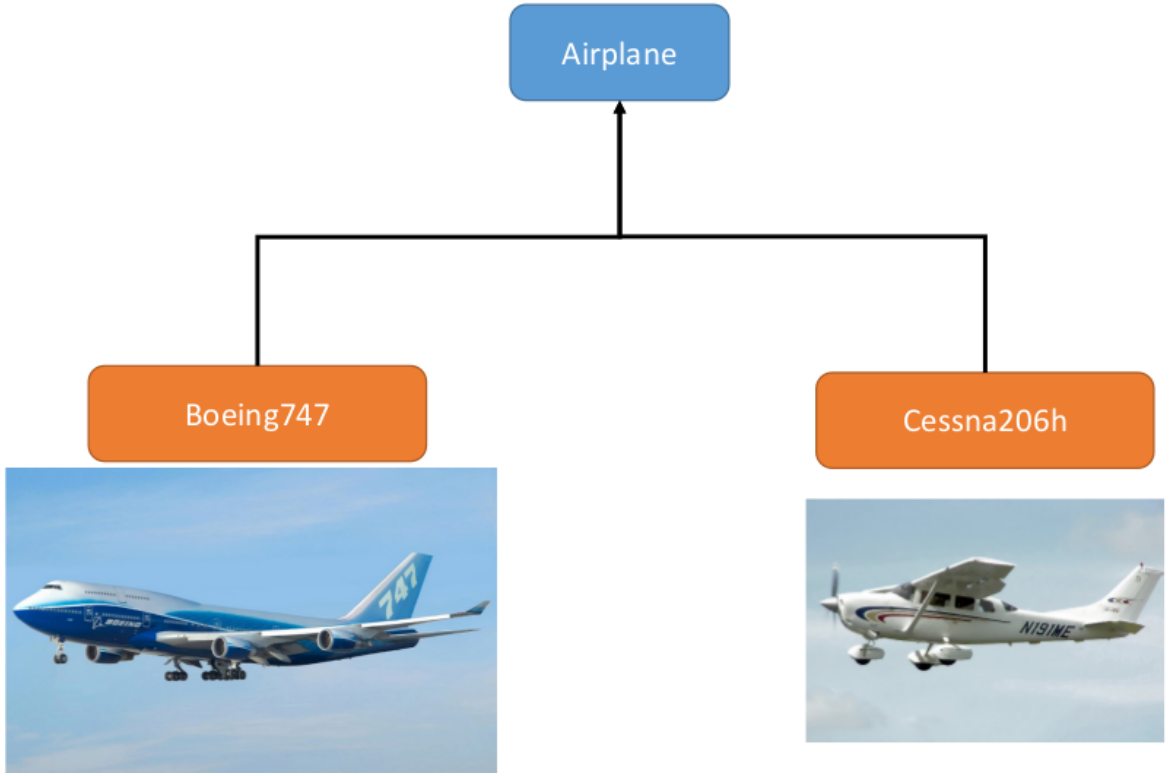
- Uno no siempre quiere que el usuario tenga acceso a todos los métodos de una clase.

-Se denomina encapsulamiento al ocultamiento del estado, es decir, de los datos miembro de un objeto de manera que sólo se pueda cambiar mediante las operaciones definidas para ese objeto.



Herencia

- Crear nuevas clases partiendo de una clase preexistente (ya comprobadas y verificadas) evitando con ello el rediseño, la modificación y verificación de la parte ya implementada.
- La herencia facilita la creación de objetos a partir de otros ya existentes e implica que una subclase obtiene todo el comportamiento (métodos) y eventualmente los atributos (variables) de su superclase.



Polimorfismo

- Propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos.
- El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.

```
def decod_obs (date, satelites):
  for satelite in satelites:
      satelite.decod_obs(date)

def get_obs(date, satelites):
  for satelite in satelites:
      satelite.get_obs(date)
```

```
satelites = []
satelites.append(satellite1)
satelites.append(satellite2)
satelites.append(satellite3)

decod_obs (datetime(2016,3,8), satelites)
```

WTPC Workshop en técnicas de programación científica

Algunos ejemplos

Ejemplos: Clases

```
class TVseries(object):
    """
    Define un clase de serie de TV
    """
    def __init__(self, nombre, episodios):                    # constructor
        self.nombre = nombre                                      #  atributos, variables de clase
        self.eps_por_temporada = episodios

    def esdato(self):                                          # método
        text = '{} tiene {} episodios por temporada.'
        return text.format(self.nombre, self.eps_por_temporada)
```

Ejemplos: Clases

```
class TVseries(object):
    """
    Define un clase de serie de TV
    """
    def __init__(self, nombre, episodios):                    # constructor
        self.nombre = nombre                                      #  atributos, variables de clase
        self.eps_por_temporada = episodios

    def esdato(self):                                          # método
        text = '{} tiene {} episodios por temporada.'
        return text.format(self.nombre, self.eps_por_temporada)
```

```
>>> got = TVseries('Game of Thrones', 10)
>>> bbt = TVseries('Big Bang Theory', 24)

>>> print bbt.nombre
Big Bang Theory

>>> print got.nombre
Game of Thrones

>>> print bbt.estado()
Big Bang Theory tiene 24 episodios por temporada.

>>> print got.estado()
Game of Thrones tiene 10 episodios por temporada.
```

Ejemplos: Métodos

```
class TVseries(object):
    """
    Define un clase de serie de TV
    """
    def __init__(self, nombre, episodios):                    # constructor
        self.nombre = nombre                                       #  atributos, variables de clase
        self.eps_por_temporada = episodios
        self.num_watched = 0

    def seen(self, num = 1):
      self.num_watched += num

    def esdato(self):                                          # método
        text = '{} tiene {} episodios por temporada. Miré {} de ellos.'
        return text.format(self.nombre, self.eps_por_temporada, self.num_watched)
```

```
>>> got = TVseries('Game of Thrones', 10)
>>> bbt = TVseries('Big Bang Theory', 24)

>>> print bbt.nombre
Big Bang Theory

>>> got.vistos(4)
>>> print got.estado()
Game of Thrones tiene 10 episodios por temporada. Miré 4 de ellos.

>>> print bbt.estado()
Big Bang Theory tiene 24 episodios por temporada. Miré 0 de ellos.
```

Ejemplos: Built-in methods

```
class TVseries(object):
  """
  Define a tv serie class
  """
  def __init__(self, name, eps):
    self.name = name
    self.eps_per_s = eps

 def seen(self,  num=1):
    self.num_watched  += num

  def __str__(self):
    text = '{} has {} episodes per season. I saw {} of them.'
     return text.format(self.name, self.eps_per_s , self.num_watched )
```

```
>>> got = TVseries('Game of Thrones', 10)
>>> bbt = TVseries('Big Bang Theory', 24)

>>> got.seen(4)
>>> print got

Game of Thrones has 10 episodes per season. I saw 0 of them.
```

Ejemplos: Herencia

```
class Perro(object):

    def __init__(self, nombre, perseguir_gatos):
        self.nombre = nombre
        self.especie = 'Perro'
        self.perseguir_gatos = perseguir_gatos

    def getNombre(self):
        return self.nombre

    def getEspecie(self):
        return self.especie

    def PerseguirGatos(self):
        return self.perseguir_gatos
```

```
class Gato(object):

    def __init__(self, nombre, dormir):
        self.nombre = nombre
        self.especie = 'Perro'
        self.dormir = dormir

    def getNombre(self):
        return self.nombre

    def getEspecie(self):
        return self.especie

    def Dormir(self):
        return self.dormir
```

¡Son casi iguales!

Ejemplos: Herencia

```
class Mascota(object):

    def __init__(self, nombre, especie):
        self.nombre = nombre
        self.especie = especie

    def getNombre(self):
        return self.nombre

    def getEspecie(self):
        return self.especie
```

```
class Perro(Mascota):

    def __init__(self, nombre, perseguir_gatos):
        Mascota.__init__(self, nombre, 'Perro')
        self.perseguir_gatos = perseguir_gatos

    def PerseguirGatos(self):
        return self.perseguir_gatos
```

```
class Gato(Mascota):

    def __init__(self, nombre, perseguir_gatos):
        Mascota.__init__(self, nombre, 'Gato')
        self.dormir = dormir

    def Dormir(self):
        return self.dormir
```

Ejemplos: Encapsulación

```
class Test(object):
    def __metodo_privado(self):
        pass
    def metodo_publico(self):
        pass
```

```
>>> mi_test = Test()

>>> mi_test.__metodo_privado()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-b2d869da9175> in <module>()
----> 1 t.__metodo_privado()

AttributeError: 'Test' object has no attribute '__metodo_privado'
```

Bonus Track: Paquetes

```
sound/
        __init__.py
        formats/
                __init__.py
                wavread.py
                wavwrite.py
                aiffread.py
                aiffwrite.py
                ...
effects/
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
filters/
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
```

```
import sound.effects as se

from sound.effects import echo

from sound.effects.echo import echofilter
```

Bonus Track: Google Python Style Guide

https://goo.gl/kxXVwK https://www.pylint.org/

Naming

link ▽ `module_name`, `package_name`, `ClassName`, `method_name`, `ExceptionName`, `function_name`, `GLOBAL_CONSTANT_NAME`, `global_var_name`, `instance_var_name`, `function_parameter_name`, `local_var_name`.

Names to Avoid

- single character names except for counters or iterators
- dashes (-) in any package/module name
- `__double_leading_and_trailing_underscore__` names (reserved by Python)

Naming Convention

- "Internal" means internal to a module or protected or private within a class.
- Prepending a single underscore (_) has some support for protecting module variables and functions (not included with `import * from`). Prepending a double underscore (__) to an instance variable or method effectively serves to make the variable or method private to its class (using name mangling).
- Place related classes and top-level functions together in a module. Unlike Java, there is no need to limit yourself to one class per module.
- Use CapWords for class names, but lower_with_under.py for module names. Although there are many existing modules named CapWords.py, this is now discouraged because it's confusing when the module happens to be named after a class. ("wait -- did I write `import StringIO` or `from StringIO import StringIO`?")

Guidelines derived from Guido's Recommendations

Type	Public	Internal
Packages	`lower_with_under`	
Modules	`lower_with_under`	`_lower_with_under`
Classes	`CapWords`	`_CapWords`
Exceptions	`CapWords`	
Functions	`lower_with_under()`	`_lower_with_under()`
Global/Class Constants	`CAPS_WITH_UNDER`	`_CAPS_WITH_UNDER`
Global/Class Variables	`lower_with_under`	`_lower_with_under`
Instance Variables	`lower_with_under`	`_lower_with_under` (protected) or `__lower_with_under` (private)
Method Names	`lower_with_under()`	`_lower_with_under()` (protected) or `__lower_with_under()` (private)
Function/Method Parameters	`lower_with_under`	
Local Variables	`lower_with_under`	

Main

▶ Even a file meant to be used as a script should be importable and a mere import should not have the side effect of executing the script's main functionality. The main functionality should be in a main() function.

Bonus Track: Copiando comportamiento

```
class Test(object):
        def __init__(self):
                self.val = 5 # immutable
                self.list = [5,6,7] # mutable
```

```
>>> a = Test()

>>> b = a
>>> c = copy(a)
>>> d = deepcopy(a)

>>> a.val, b.val, c.val, d.val
(5, 5, 5, 5)

>>> a.val = 7
>>> a.val, b.val, c.val, d.val
(7, 7, 5, 5)

>>> a.list, b.list, c.list, d.list
([5, 6, 7], [5, 6, 7], [5, 6, 7], [5, 6, 7])

>>> a.list.append(999)
>>> a.list, b.list, c.list, d.list
([5, 6, 7, 999], [5, 6, 7, 999], [5, 6, 7, 999], [5, 6, 7])

>>> a.list = 'Hello'
>>> a.list, b.list, c.list, d.list
('Hello', 'Hello', [5, 6, 7, 999], [5, 6, 7])
```