1. Programación orientada a objetos

Programación II

Grado en Inteligencia Artificial

M. Cabrero

facultade de informática da coruña

UNIVERSIDADE DA CORUÑA

Contenidos

Introducción

Programación *Imperativa*

- Se describe paso a paso el proceso de ejecución de un programa.
- El orden de los pasos es crucial, porque el paso siguiente depende en gran medida del anterior.

```
# Calculate total in the list
total = 0
myList = [1,2,3,4,5]

# Create a for loop to add numbers in the list to the total
for x in myList:
     total += x
print(total)
```

Programación *Funcional*

- Se describe lo que se quiere que consiga el programa en lugar de cómo debe ejecutarse (flujo de control).

```
mylist = [1,2,3,4,5]

# set total to the sum of numbers in mylist
total = sum(mylist)
print(total)
```

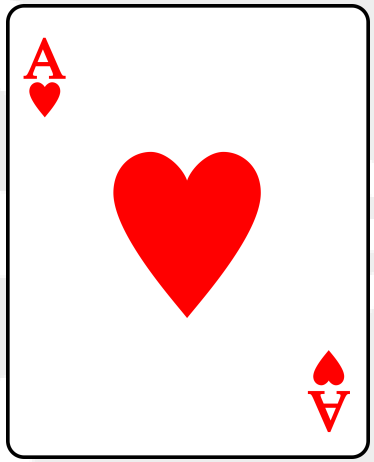
Tipos de Programación Imperativa

- *Procedural o estructurada*:
  - Construcción de conjuntos de instrucciones con nombre (funciones o procedimientos)
  - Paso de datos a través de llamadas a esas funciones. Retorno de resultados
  - Almacenamiento de datos locales no accesibles desde fuera del ámbito de la función
  - Acceso y modificación de variables globales

- *Orientada a objetos*:
  - Combinación de código y datos en unidades cohesionadas que suelen ser muy reutilizables
  - Clase, objeto, abstracción, herencia, encapsulación y polimorfismo.

Ejemplo 1: Juego de Cartas Mayor o Menor

- 8 cartas de la baraja
- Se muestra la primera
- La siguiente, ¿mayor o menor?
  - Acierto: 20 ptos
  - Fallo: -15 ptos
- Mismo valor ocasiona fallo



Representación de datos

- Baraja/Mazo: lista de 52 cartas
- Carta: diccionario con tres pares clave/valor
  - Nombre (rank): As, 2., 3, ..., 10, Jota, Reina, Rey
  - Valor (value): 1..13
  - Palo (suite): Picas ♠, Corazones ♥, Tréboles ♣, Diamantes ♦
  - P.e. Jota de Tréboles:

```
{'nombre': 'Jota', 'palo': 'Tréboles', 'valor': 10}
```

Implementación

- Crear la baraja: lista de cartas
- Crear el mazo: clon de la lista de cartas (*suffle*)
- Recuperar primera carta del mazo
- Guardar componentes (nombre, valor y palo) en 3 variables globales
- Para cada ronda (7 cartas):
  - Pedir al usuario una predicción
  - Recuperar siguiente carta
  - Guardar componentes en otras 3 variables globales
  - Comparar respuesta del usuario con la carta extraída
  - Asignar puntos
- ¿Volver a jugar?

¿Código reutilizable?

- Uso de muchos elementos de programación: variables, sentencias de asignación, if/else, print, bucles while, funciones y variables de tipo listas, cadenas y diccionarios
- Es difícil identificar todas las piezas del código
- Los datos y el código que manipula los datos pueden no estar muy agrupados
- Conclusión: reutilizar para otro programa no es fácil

Solución OO
Primer vistazo a una clase Card

```
class Card:
    def __init__(self, rank, suit, value):
        self.rank = rank
        self.suit = suit
        self.cardName = rank + ' of ' + suit
        self.value = value

    def getName(self):
        return self.cardName

    def getValue(self):
        return self.value

    def getSuit(self):
        return self.suit

    def getRank(self):
        return self.rank
```

Solución OO
Primer vistazo a una clase Baraja I

```
import random
from Card import *

class Deck:
    SUIT_TUPLE = ('Diamonds', 'Clubs', 'Hearts', 'Spades')
    STANDARD_DICT = {'Ace':1, '2':2, '3':3, '4':4, '5':5, '6':6, '7':7,
                     '8': 8, '9':9, '10':10, 'Jack':11, 'Queen':12, 'King':13}

    def __init__(self, rankValueDict=STANDARD_DICT):
        self.startingDeckList = []
        self.playingDeckList = []
        for suit in Deck.SUIT_TUPLE:
            for rank, value in rankValueDict.items():
                oCard = Card(rank, suit, value)
                self.startingDeckList.append(oCard)
        self.shuffle()

    def shuffle(self):
        # Copy the starting deck and save it in the playing deck list
        self.playingDeckList = self.startingDeckList.copy()
        random.shuffle(self.playingDeckList)
```

Solución OO
Primer vistazo a una clase Baraja II

```
import random
from Card import *

class Deck:
    def __init__(self, rankValueDict=STANDARD_DICT):
        ...

    def shuffle(self):
        ...

    def getCard(self):
        if len(self.playingDeckList) == 0:
            raise IndexError('No more cards')
        # Pop one card off the deck and return it
        oCard = self.playingDeckList.pop()
        return oCard

    def returnCardToDeck(self, oCard):
        # Put a card back into the deck
        self.playingDeckList.insert(0, oCard)
```

Ejemplo 2: Cuenta bancaria

- Simularemos el funcionamiento de un banco
- Operaciones típicas de un cliente: crear una cuenta, depositar y retirar fondos y comprobar el saldo
- Datos necesarios: nombre, contraseña y saldo del cliente
- Simplificación: Una cuenta, contraseñas son texto, y cantidades son enteros

Implementación: Única cuenta *sin* funciones

- Inicialización de tres variables globales (nombre, contraseña y saldo)
- Se muestra un menú para elegir la operación
- No hay funciones: todas las acciones a nivel principal

Alerta

Programa demasiado largo: mover código relacionado a funciones y hacer llamadas a esas funciones

Implementación: Única cuenta *con* funciones

- Se crea una función para cada una de las operaciones (crear, comprobar el saldo, depositar y retirar)
- Reorganización: el código principal contiene llamadas a las funciones
- Programa mucho más legible: si tecleamos `d` (depósito) se llama a la función `deposit()`
- Se accede a (obtener o establecer) las variables *globales* que representan la cuenta

Alerta

Principio general de programación: las funciones nunca deben modificar variables globales, sólo deben usar los datos que se le pasan, hacer cálculos basados en esos datos y devolver un resultado(s)

Implementación: Dos cuentas *con* funciones

- Misma versión anterior con soporte para dos cuentas
- Tres variables globales por cuenta bancaria para almacenar los datos
- Sentencias `if...else` en cada función para elegir qué conjunto de variables globales procesar

Alerta

Añadir más sentencias `if...else` y vars. globales para cada *nueva* cuenta

Implementación: Múltiples cuentas usando listas

- Representar los datos usando tres listas *paralelas* globales: `nombres`, `contraseñas`, y `saldos`
- El número de cuenta es el índice de las listas: `0`, `1`, `...`, `n`
- Los datos no están agrupados de forma lógica (por cliente)

Account N.	Name	Password	Balance
0	Joe	soup	100
1	Mary	nuts	3550
2	Bill	frisbee	1000
3	Sue	xxyyzz	750
4	Henry	PW	10000

Alerta

Añadir un nuevo atributo es añadir una nueva lista (global)

Implementación: Lista de diccionarios de cuentas

- Cada cuenta es un diccionario:

```
{'name':<someName>, 'password':<somePassword>, 'balance':<someBalance>}
```

- Todas las cuentas se almacenan en una lista (global)
- El índice de la lista corresponde al número de cuenta
- Los datos se agrupan de forma lógica (por cliente)

Account N.	Name	Password	Balance
0	Joe	soup	100
1	Mary	nuts	3550
2	Bill	frisbee	1000
3	Sue	xxyyzz	750
4	Henry	PW	10000

Alerta

Las funciones siguen accediendo a la lista global de cuentas

Implementación procedural: Problemas

- La función que accede a/cambia datos globales no es fácilmente reutilizable, porque éstos *viven* en un nivel superior.
- Es difícil depurar y mantener programas que manipulan variables globales tanto en el programa principal como dentro de las funciones.
- La función que usa variables globales tiene acceso a demasiados datos: en condiciones normales debería poder operar sobre una pequeña parte.

Solución OO
Primer vistazo a una clase Account

```
class Account:
    def __init__(self, name, balance, password):
        self.name = name
        self.balance = int(balance)
        self.password = password

    def deposit(self, amountToDeposit, password):
            if password != self.password:
            print('Sorry, incorrect password')
            return None

        if amountToDeposit < 0:
            print('You cannot deposit a negative amount')
            return None

        self.balance = self.balance + amountToDeposit
        return self.balance
```

Solución OO
Primer vistazo a una clase Account

```
def withdraw(self, amountToWithdraw, password):
        if password != self.password:
        print('Sorry, incorrect password')
        return None

    if amountToWithdraw < 0:
        print('You cannot withdraw a negative amount')
        return None

    if amountToWithdraw > self.balance:
        print('You cannot withdraw more than you have in your account')
        return None

    self.balance = self.balance - amountToWithdraw
    return self.balance

def getBalance(self, password):
        if password != self.password:
        print('Sorry, incorrect password')
        return None
    return self.balance
```

Modelado de objetos físicos con OO

Modelado software

- La descripción de un objeto hace referencia a sus atributos, muchas veces comunes a varios. P.e. un escritorio tiene color, dimensiones, peso, material, ...
- Algunos objetos tienen atributos propios. P.e. el coche tiene puertas pero una camisa no tiene. La caja puede estar abierta o cerrada, vacía o llena pero esas no son características aplicables a un ladrillo.
- Otros objetos pueden realizar acciones, como avanzar, retroceder, girar a la izquierda o a la derecha.
- Para modelar un objeto del mundo real en código, hay que decidir:
  - los datos que representarán los atributos (*estado*)
  - las operaciones que puede realizar (*comportamiento*)

Ejemplo: modelo procedural de interruptor

```
def turnOn():
        global switchIsOn
        # turn the light on
        switchIsOn = True

def turnOff():
        global switchIsOn
        # turn the light off
        switchIsOn = False

# Main code
switchIsOn = False

# Test code
print(switchIsOn)
turnOn()
print(switchIsOn)
turnOff()
print(switchIsOn)
turnOn()
print(switchIsOn)
```

- *Estado*: on/off (`switchIsOn`)
- *Comportamiento*: acciones `turnOn()`/`turnOff()` que cambian el valor de la variable

Salida por pantalla

```
False
True
False
True
```

Introduciendo conceptos de OO

- Una clase es como una plantilla que define el aspecto que tendrá un objeto cuando se cree a partir de ella.
- Pongamos como ejemplo, un negocio de pastelería:
  - Trabaja a demanda: hacen su famoso Bundt Cake cuando llega el pedido.
  - Tiene un molde que define su aspecto.
  - El pastel es un objeto creado con el molde.
  - Con el molde se crea cualquier número de cakes.
  - Los cakes pueden tener distintos atributos: sabores, glaseado, pepitas de chocolate, ..., pero todos salen del mismo molde.

La clase Interruptor de la luz

Definición

Una clase es el código que define lo que un objeto *recordará* (sus datos o estado) y las cosas que *podrá hacer* (sus funciones o comportamiento).

```
class LightSwitch:
    def __init__(self):
        self.switchIsOn = False

    def turnOn(self):
        # turn the switch on
        self.switchIsOn = True

    def turnOff(self):
        # turn the switch off
        self.switchIsOn = False
```

- Una única variable `self.switchIsOn`
- Dos funciones para comportamiento
- Si se ejecuta no ocurre nada (igual que las funciones)
- Es necesario crear un objeto de la clase

Un objeto/instancia *Interruptor de la luz*

Definición

La instanciación es el proceso de crear un objeto a partir de una clase.

- `oLightSwitch = LightSwitch()`
  - Encuentra la clase `LightSwitch`
  - Crea un objeto `LightSwitch` a partir de esa clase
  - Asigna el objeto resultante a la variable `oLightSwitch`
- Decimos que "un objeto `LightSwitch` es una instancia de la clase `LightSwitch`"

Escribiendo una clase

```
class NombreClase():

    def __init__(self, param1, ..., paramN):
        # cualquier código de inicialización aquí

    # Cualquier número de funciones que acceden a los datos
    # Cada una de la forma:

    def nombreFuncion1(self, param1, ..., paramN):
        # cuerpo de la función

    # ... más funciones
    def nombreFuncion2(self, param1, ..., paramN):
        # cuerpo de la función
```

- Nombre de la clase (camel case e inicial en mayúscula)
- En el cuerpo, cualquier número de funciones (comportamiento del objeto) escritas con *sangrado*
- Existe un parámetro necesario: `self`

Las funciones son métodos

Definición

Un método es una función definida dentro de una clase. Siempre tiene al menos un parámetro, que normalmente se llama `self`.

- `__init__` es el primer método de cada clase que se ejecuta automáticamente al crear un objeto.
- Es el lugar donde inicializar *variables de instancia*

Ámbito de las variables

- Programación procedimental: ámbito global y local
- POO: añade el ámbito de definición de clase (u objeto)

Definición

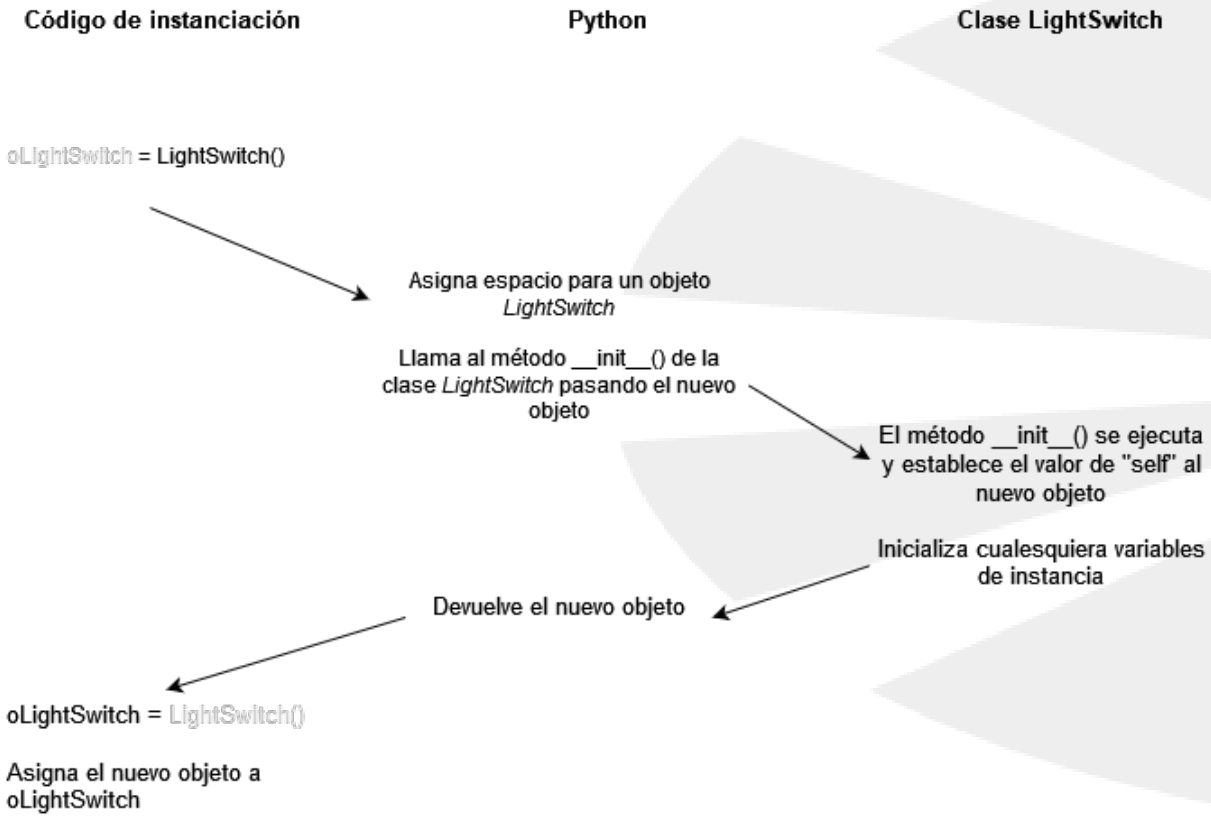
Además de locales, los métodos pueden tener *variables de instancia*, cuyo ámbito es el objeto (y todos sus métodos) y su nombre empieza por self.

```
class MyClass():
    def __init__(self):
        self.count = 0 # crea self.count
                       #  e inicializa a 0

    def increment(self):
        self.count = self.count + 1 # incrementa variable
```

- Cada objeto obtiene (y mantiene) su propio conjunto de variables de instancia (ejemplo `LightSwitch`)

Creando un objeto a partir de una clase

```
<objeto> = <NombreClase>(<argumentos opcionales>)}
```



Llamada a los métodos de un objeto

```
<objeto>.<nombreMetodo>(<cualquier argumento>)
```

```
class LightSwitch:
    def __init__(self):
        self.switchIsOn = False

    def turnOn(self):
        # turn the switch on
        self.switchIsOn = True

    def turnOff(self):
        # turn the switch off
        self.switchIsOn = False

    def show(self):
        # added for testing
        print(self.switchIsOn)
```

```
from LightSwitch import *

oLightSwitch = LightSwitch()

oLightSwitch.show()
oLightSwitch.turnOn()
oLightSwitch.show()
oLightSwitch.turnOff()
oLightSwitch.show()
oLightSwitch.turnOn()
oLightSwitch.show()
```

Salida por pantalla

```
False
True
False
True
```

Creación de múltiples instancias de la misma clase

```
from LightSwitch import *

# Main code
oLightSwitch1 = LightSwitch() # create a LightSwitch object
oLightSwitch2 = LightSwitch() # create another LightSwitch object

# Test code
oLightSwitch1.show()
oLightSwitch2.show()
oLightSwitch1.turnOn() # Turn switch 1 on

# Switch 2 should be off at start, but this makes it clearer
oLightSwitch2.turnOff()
oLightSwitch1.show()
oLightSwitch2.show()
```

Salida por pantalla

```
False
False
True
False
```

- Los cambios realizados en los datos de un objeto no afectan a los de otro objeto.

Usando clases en Python sin saberlo

- Todos los tipos incorporados en Python se implementan como clases.
- Cuando se escribe una nueva clase se define un nuevo tipo de datos

```
>>> miCadena = 'abcde'
>>> print(type(miCadena))
<class 'str'>
>>> miLista = [10, 20, 30, 40]
>>> print(type(miLista))
<class 'lista'>
```

```
>>> oLightSwitch = LightSwitch()
>>> print(type(oLightSwitch))
<class 'LightSwitch'>
```

Construir una clase más complicada

```
class DimmerSwitch:
    def __init__(self):
        self.switchIsOn = False
        self.brightness = 0

    def turnOn(self):
        self.switchIsOn = True

    def turnOff(self):
        self.switchIsOn = False

    def raiseLevel(self):
        if self.brightness < 10:
            self.brightness = self.brightness + 1

    def lowerLevel(self):
        if self.brightness > 0:
            self.brightness = self.brightness - 1

   # Extra method for debugging
    def show(self):
        print('Switch is on?', self.switchIsOn)
        print('Brightness is:', self.brightness)
```

```
from DimmerSwitch import *

oDimmer = DimmerSwitch()

# Turn switch on, and raise level 3 times
oDimmer.turnOn()
oDimmer.raiseLevel()
oDimmer.raiseLevel()
oDimmer.raiseLevel()
oDimmer.show()

# Lower level 2 times, and turn switch off
oDimmer.lowerLevel()
oDimmer.lowerLevel()
oDimmer.turnOff()
oDimmer.show()

# Turn switch on, and raise level 1 time
oDimmer.turnOn()
oDimmer.raiseLevel()
oDimmer.show()
```

Salida por pantalla

```
Switch is on? True
Brightness is: 3
Switch is on? False
Brightness is: 1
Switch is on? True
Brightness is: 2
```

Representar un objeto físico real: TV

- Un televisor requiere más datos que un interruptor para representar su estado, y tiene más comportamientos.
- Estados (variables de instancia):
  - encendido o apagado, silencio, lista de canales disponibles, canal actual, volumen actual y rango disponible de niveles de volumen
- ...y acciones (métodos):
  - encender/apagar, subir/bajar volumen, subir/bajar el canal, silenciar/activar el sonido, obtener información sobre la configuración actual e ir a un canal especificado.

Clase TV

```
class TV:
    def __init__(self):
        self.isOn = False
        self.isMuted = False

        self.chList = [2, 11, 54, 65]
        self.nChannels = len(self.chList)
        self.chIndex = 0
        self.VOLUME_MIN = 0 # constant
        self.VOLUME_MAX = 10 # constant
        self.volume = self.VOLUME_MAX // 2

    def power(self):
        self.isOn = not self.isOn # toggle

    def volumeUp(self):
        if not self.isOn:
            return
        if self.isMuted:
            self.isMuted = False
        if self.volume < self.VOLUME_MAX:
            self.volume = self.volume + 1

    def volumeDown(self):
        if not self.isOn:
            return
        if self.isMuted:
            self.isMuted = False
        if self.volume > self.VOLUME_MIN:
            self.volume = self.volume - 1
```

```
def channelUp(self):
    if not self.isOn:
        return
    self.chIndex = self.chIndex + 1
    if self.chIndex > self.nChannels:
        self.chIndex = 0

def channelDown(self):
    ...

def mute(self):
    if not self.isOn:
        return
    self.isMuted = not self.isMuted

def setChannel(self, newChannel):
    if newChannel in self.chList:
        self.chIndex = self.chList.index(newChannel)

def showInfo(self):
    if self.isOn:
        print(' TV is: On')
        print(' Ch:', self.chList[self.chIndex])
        if self.isMuted:
            print(' Volume is:', self.volume, '(muted)')
        else:
            print(' Volume is:', self.volume)
    else:
        print(' TV is: Off')
```

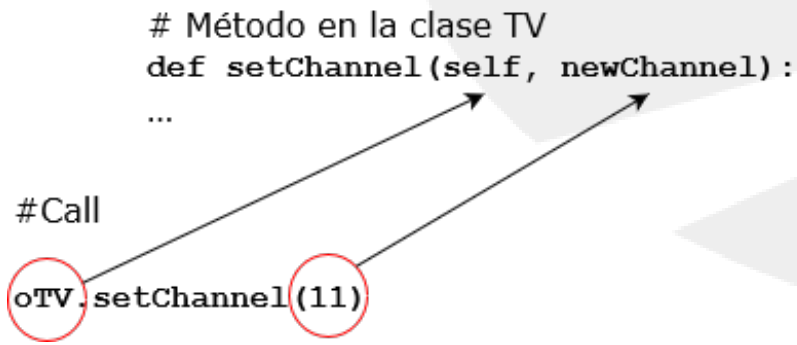
Clase TV

```
from TV import *

oTV = TV() # create the TV object

# Turn on TV on and show the status
oTV.power()
oTV.showInfo()

# Change the channel up twice, raise the volume twice, show status
oTV.channelUp()
oTV.channelUp()
oTV.volumeUp()
oTV.volumeUp()
oTV.showInfo()

# Turn the TV off, show status, turn the TV on, show status
oTV.power()
oTV.showInfo()
oTV.power()
oTV.showInfo()

# Lower the volume, mute the sound, show status
oTV.volumeDown()
oTV.mute()
oTV.showInfo()

# Change the channel to 11, mute the sound, show status
oTV.setChannel(11)
oTV.mute()
oTV.showInfo()
```

```
TV Status:
 TV is: On
 Channel is: 2
 Volume is: 5

TV Status:
 TV is: On
 Channel is: 5
 Volume is: 7

TV Status:
 TV is: Off

TV Status:
 TV is: On
 Channel is: 5
 Volume is: 7

TV Status:
 TV is: On
 Channel is: 5
 Volume is: 6 (muted)

TV Status:
 TV is: On
 Channel is: 11
 Volume is: 6
```

Paso de argumentos a un método

- En cada llamada a una función (o método), el número de argumentos tiene que ser igual al número de parámetros en la sentencia `def`.
  - `def power(self):` espera un valor, sin embargo se invoca como `oTV.power()` ¿Falta el argumento?
  - `def setChannel(self, newChannel):` requiere dos, pero sólo se pasa un argumento `oTV.setChannel(11)`



- El método se ejecuta con los valores de las variables de instancia del objeto pasado como primer argumento.

Instancias múltiples

- Cada método se escribe con `self` como primer parámetro (recibe el objeto utilizado en cada llamada).
- Esto permite que cualquier método dentro de una clase trabaje con diferentes objetos.
- Creando dos objetos con `oTV1 = TV()` y `oTV2 = TV()`, tendremos un conjunto de valores de instancia distintos: cada TV tiene un ajuste de volumen, una lista de canales, un ajuste de canal, etc.
- Haciendo llamadas a los distintos métodos de los diferentes objetos, el estado de cada TV será distinto.

Instancias múltiples

```
from TV import *

# Main code
oTV1 = TV() # create one TV object
oTV2 = TV() # create another TV object

# Turn both TVs on
oTV1.power()
oTV2.power()

# Raise the volume of TV1
oTV1.volumeUp()
oTV1.volumeUp()

# Raise the volume of TV2
oTV2.volumeUp()
oTV2.volumeUp()
oTV2.volumeUp()
oTV2.volumeUp()
oTV2.volumeUp()

# Change TV2's channel, then mute it
oTV2.setChannel(44)
oTV2.mute()

# Now display both TVs
oTV1.showInfo()
oTV2.showInfo()
```

Salida por pantalla

```
Status of TV:
 TV is: On
 Channel is: 2
 Volume is: 7

Status of TV:
 TV is: On
 Channel is: 44
 Volume is: 10 (muted)
```

Parámetros de inicialización

- Al instanciar un objeto existe la posibilidad de pasar argumentos al método `__init__()` para crear instancias diferentes con distintos valores iniciales.
- Por convención el nombre de la variable de instancia será el mismo que el nombre del parámetro.

```
class TV:
    def __init__(self, marca, location):
        self.marca = marca
        self.ubicacion = ubicacion
        ...
    def showInfo(self):
        print()
        print('Status of TV:', self.marca)
        print(' Location:', self.ubicacion)
        if self.isOn:
        ...

oTV1 = TV('Samsung', 'Salón')
oTV2 = TV('Sony', 'Dormitorio')
--
oTV1.showInfo()
oTV2.showInfo()
...
```

Salida por pantalla

```
Status of TV: Sony
 Location: Dormitorio
 TV is: On
 Channel is: 2
 Volume is: 7

Status of TV: Samsung
 Location: Salón
 TV is: On
 Channel is: 44
 Volume is: 10 (muted)
```

Clases en uso habitual

- Modelar un estudiante en un curso.
  - Clase: `Student`
  - Variables de instancia: `name`, `emailAddress`, `currentGrade`, etc.
- Juego con varios jugadores.
  - Clase: `Player`
  - Variables de instancia: `name`, `points`, `health`, `location`, etc.
- Una libreta de direcciones.
  - Clase: `Person`
  - Variables de instancia: `name`, `address`, `phoneNumber`, y `birthday`.
- Cada objeto creado tiene idénticas capacidades (métodos) que pueden funcionar de manera diferente en función de los valores de las variables de instancia.

POO como solución

Lecciones aprendidas:

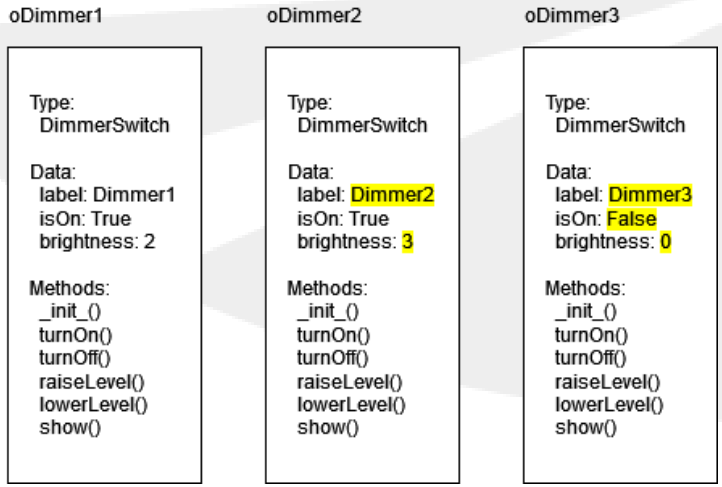
- Una clase bien escrita se puede reutilizar fácilmente, porque no necesitan acceder a datos globales.
- La POO permite depurar el código más fácilmente porque la clase es un marco en el que los datos y el código que actúa sobre los datos existen en una misma agrupación.
- Los objetos sólo tienen acceso a sus *propios* datos (variables de instancia), incluso cuando hay múltiples objetos de la misma clase.

Modelo mental de objetos (alto nivel)

```
class DimmerSwitch:
    def __init__(self, label):
        self.label = label
        ...
    def show(self):
        print(' Label: ', self.label)
        ...
oDimmer1 = DimmerSwitch('Dimmer1')
oDimmer1.turnOn()
oDimmer1.raiseLevel()
oDimmer1.raiseLevel()

oDimmer2 = DimmerSwitch('Dimmer2')
oDimmer2.turnOn()
oDimmer2.raiseLevel()
oDimmer2.raiseLevel()
oDimmer2.raiseLevel()

oDimmer3 = DimmerSwitch('Dimmer3')

oDimmer1.show()
oDimmer2.show()
oDimmer3.show()
```

Salida por pantalla

```
Label: Dimmer1
Switch is on? True
Brightness is: 2

Label: Dimmer2
Switch is on? True
Brightness is: 3

Label: Dimmer3
Switch is on? False
Brightness is: 0
```



Modelo mental de objetos (bajo nivel)

```
# Create three DimmerSwitch objects
oDimmer1 = DimmerSwitch('Dimmer1')
print(type(oDimmer1))
print(oDimmer1)
print()

oDimmer2 = DimmerSwitch('Dimmer2')
print(type(oDimmer2))
print(oDimmer2)
print()

oDimmer3 = DimmerSwitch('Dimmer3')
print(type(oDimmer3))
print(oDimmer3)
print()
```

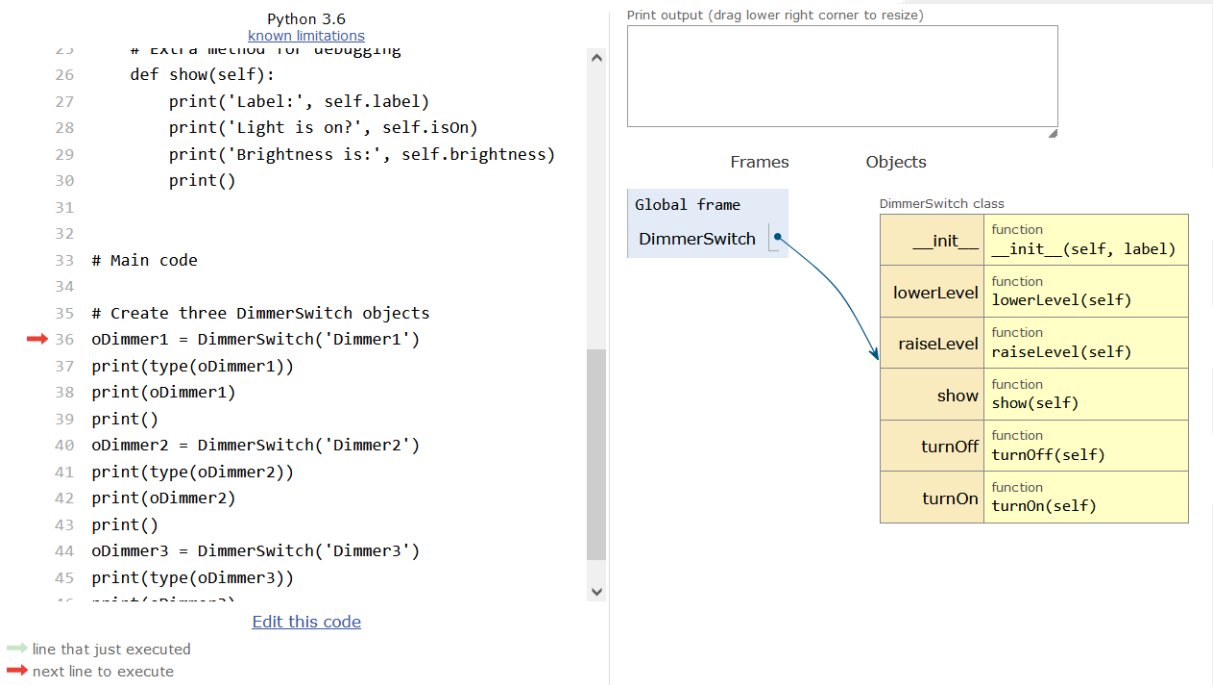
Salida por pantalla

```
<class '__main__.DimmerSwitch'>
<__main__.DimmerSwitch object at 0x7ffe503b32e0>

<class '__main__.DimmerSwitch'>
<__main__.DimmerSwitch object at 0x7ffe503b3970>

<class '__main__.DimmerSwitch'>
<__main__.DimmerSwitch object at 0x7ffe503b39d0>
```

- Tipo de datos
- Ubicación en la memoria

¿Qué pasa por dentro?

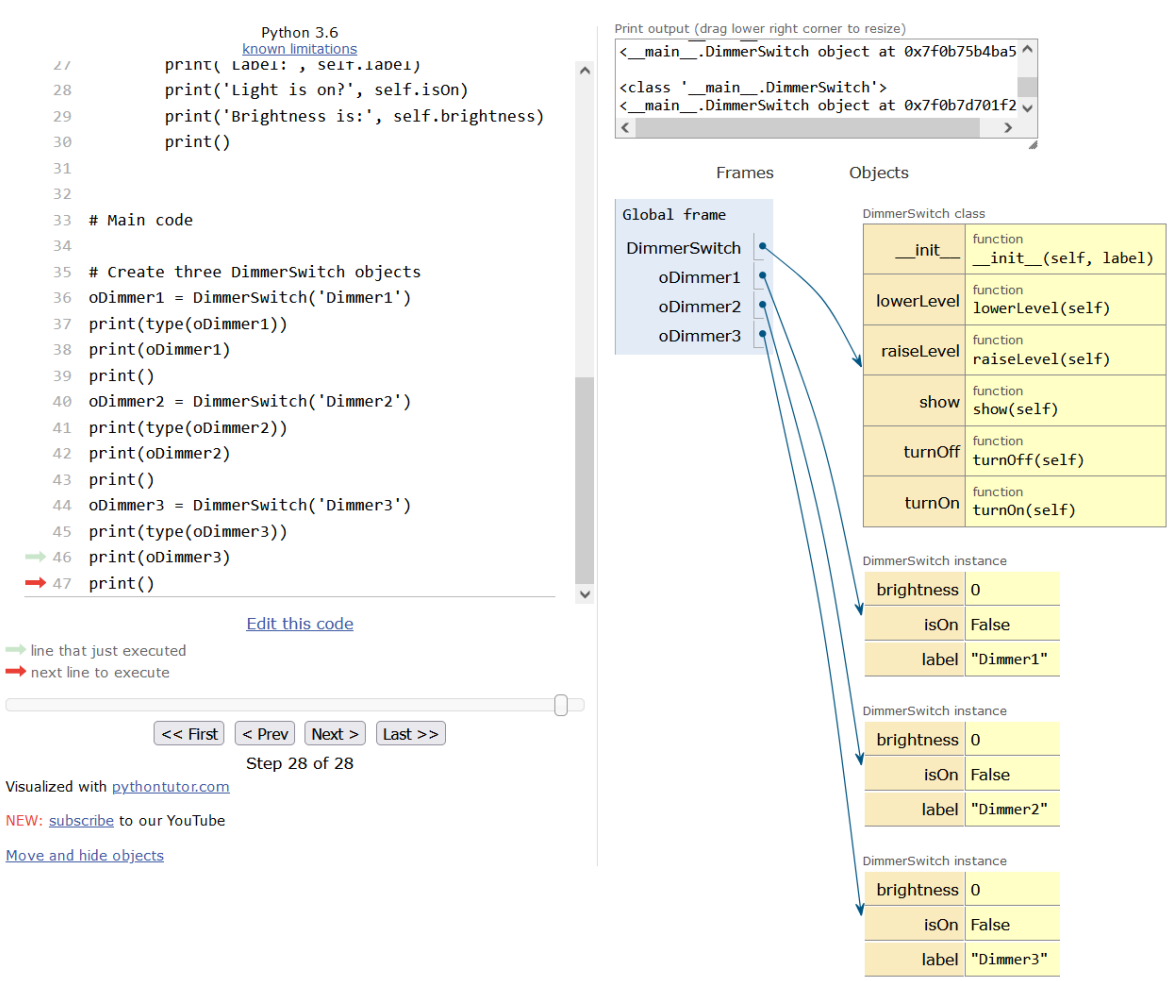
Antes de instanciar el primer objeto `DimmerSwitch`



`http://PythonTutor.com`

¿Qué pasa por dentro?

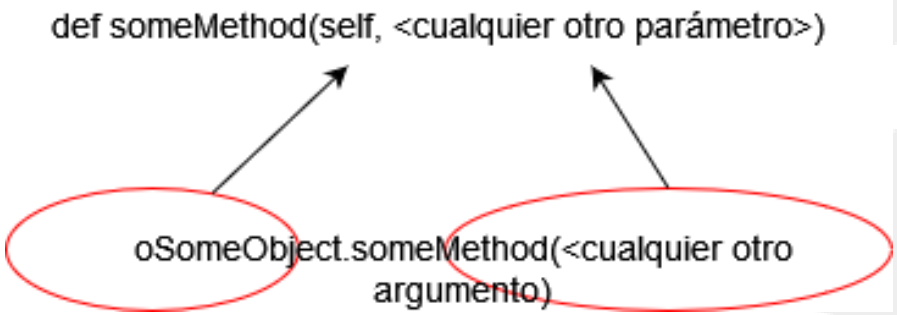
Después de instanciar el tercer objeto `DimmerSwitch`



`http://PythonTutor.com`

¿Qué significa "self"?

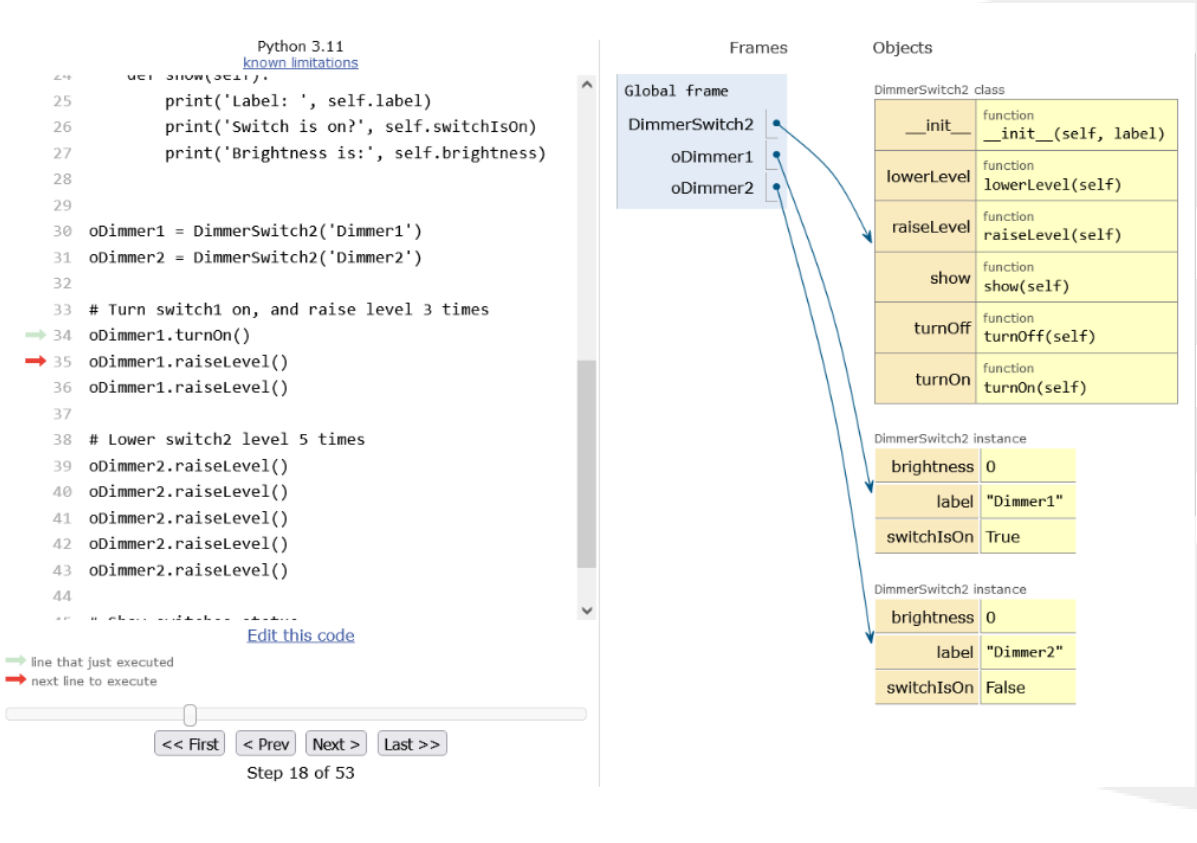
- Si instanciamos un objeto oSomeObject...
  `oSomeObject = SomeClass(<args opcionales>)`
- ...cada método de la clase es...
  `def someMethod(self, <otro parametro>):`
- ...y se invoca así:
  `oSomeObject.someMethod(<otro argumento>)`
- ...el objeto (self) es el primer argumento automático.



- Un método accede a las variables de instancia de cualquier objeto instanciado de la clase con...
  `self.<nombreVariableInstancia>`

Evolución de self

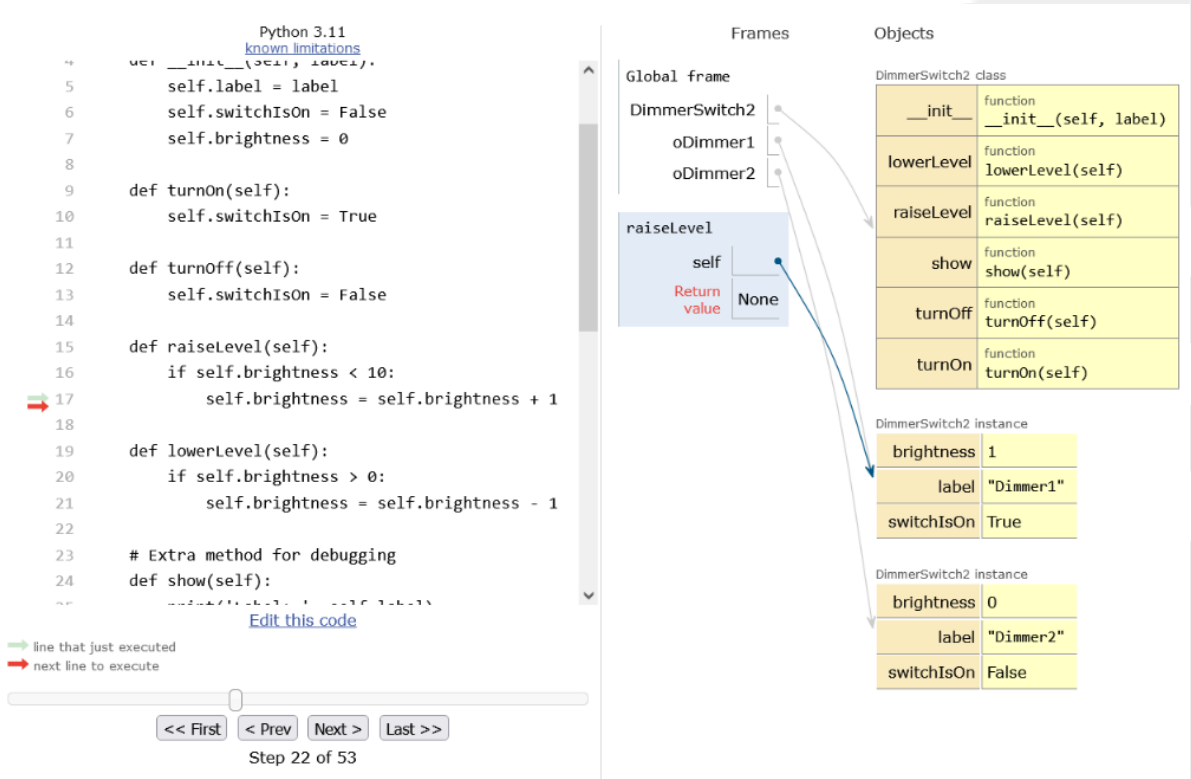
Antes de la primera llamada a `raiseLevel()`



`http://PythonTutor.com`

Evolución de self

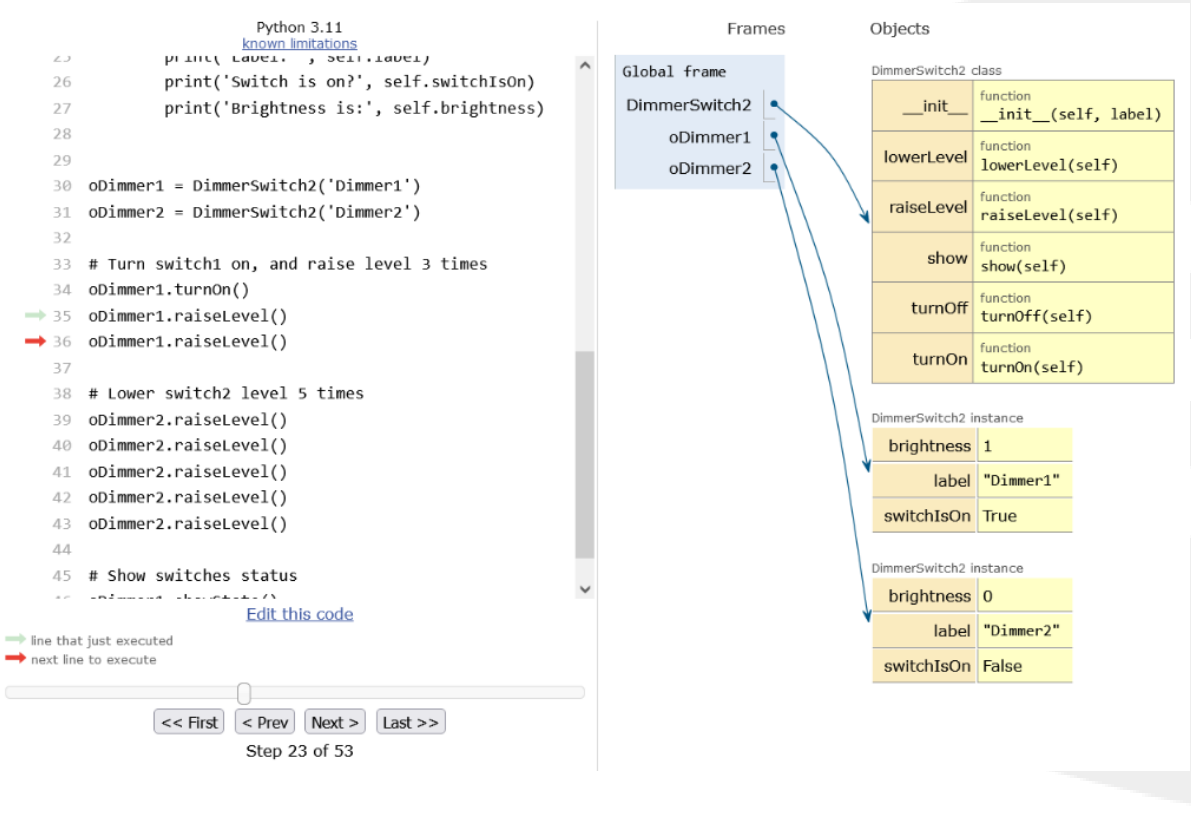
Se realiza la primera llamada a `raiseLevel()`



`http://PythonTutor.com`

Evolución de self

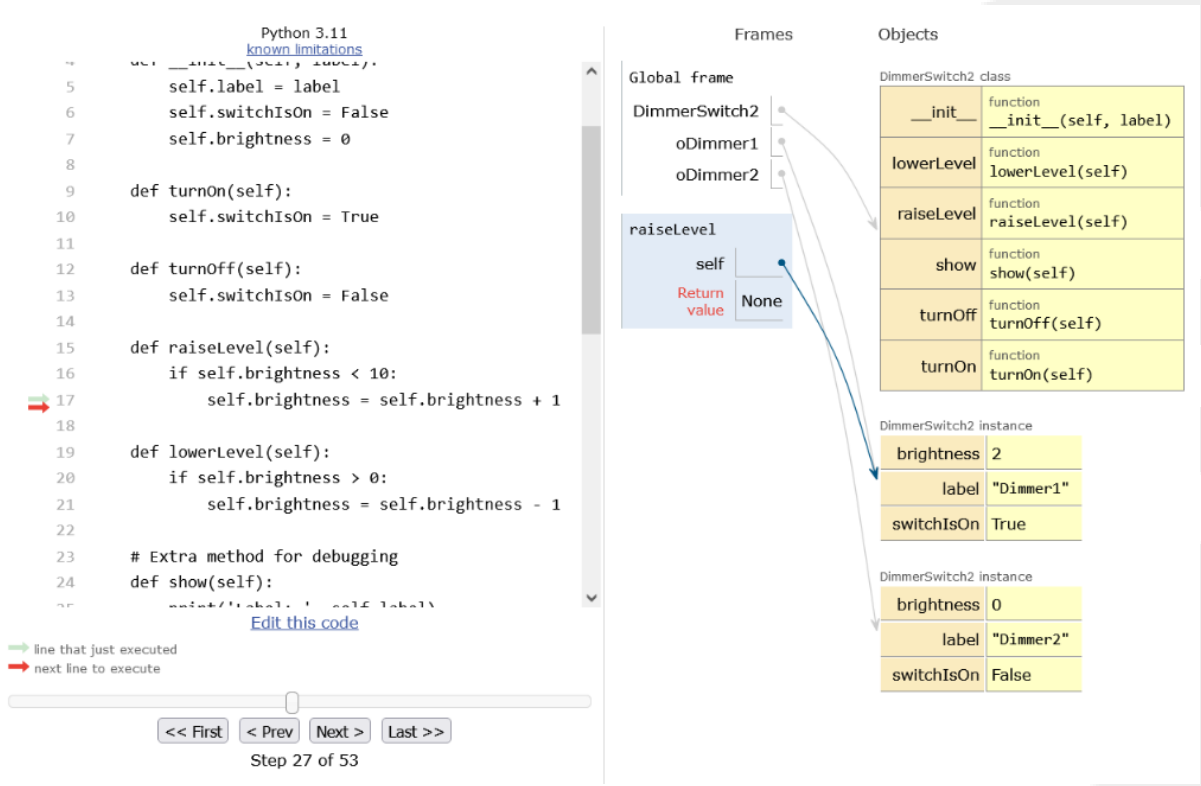
Antes de la segunda llamada a `raiseLevel()`



`http://PythonTutor.com`

Evolución de self

Se realiza la segunda llamada a `raiseLevel()`



`http://PythonTutor.com`

Gestión de múltiples objetos

Representación de un objeto no físico: Account*

```
class Account:
    def __init__(self, name, balance, password):
        self.name = name
        self.balance = int(balance)
        self.password = password
```

- Abrir una cuenta:

```
oAccount = Account('Joe Schmoe', 1000, 'magic')
```

*Definición completa en transparencias 21 y 22

Representación de un objeto no físico: Account

```
class Account:
    def deposit(self, amountToDeposit, password):
        if password != self.password:
            print('Sorry, incorrect password')
            return None

        if amountToDeposit < 0:
            print('You cannot deposit a negative amount')
            return None

        self.balance = self.balance + amountToDeposit
        return self.balance
```

- Hacer un depósito:

```
newBalance = oAccount.deposit(500, 'magic')
```

Representación de un objeto no físico: Account

```
class Account:
    def withdraw(self, amountToWithdraw, password):
            if password != self.password:
            print('Sorry, incorrect password')
            return None

        if amountToWithdraw < 0:
            print('You cannot withdraw a negative amount')
            return None

        if amountToWithdraw > self.balance:
            print('You cannot withdraw more than you have in your account')
            return None

        self.balance = self.balance - amountToWithdraw
        return self.balance
```

- Retirar fondos:

```
newBalance = oAccount.withdraw(250, 'magic')
```

Representación de un objeto no físico: Account

```
class Account:
    def getBalance(self, password):
        if password != self.password:
            print('Sorry, incorrect password')
            return None

        return self.balance
```

- Comprobar el saldo:

```
currentBalance = oAccount.getBalance('magic')
```

Importar el código de una clase

¿Cómo utilizar una clase en el propio código?

- Opción 1: Todo el código de la clase en el archivo fuente principal `main.py`. Dificulta la reutilización.
- Opción 2: Incluirlo en un archivo `<className>.py` e importarlo en el programa principal `main.py`. La ejecución del archivo no produce resultados. Incluir *sólo* una clase por fichero.

```
from <ExternalFile> import <ClassName1>, <ClassName2>
from account import *
```

Probando la clase `Account`: Alternativas

- Creando cada cuenta y guardándola en una variable (global)
- Usando una lista (global) de objetos `Account`: al abrir la cuenta se instancia el objeto `Account` y se añade a una lista (el índice será el número de cuenta)
  - Si se borra una cuenta, las siguientes tendrán índices no válidos
  - No se cambia el código de la clase `Account`

```
accountsList = []
oAccount = Account(userName, userBalance, userPassword)
accountsList.append(oAccount)
```

Probando la clase Account: Alternativas

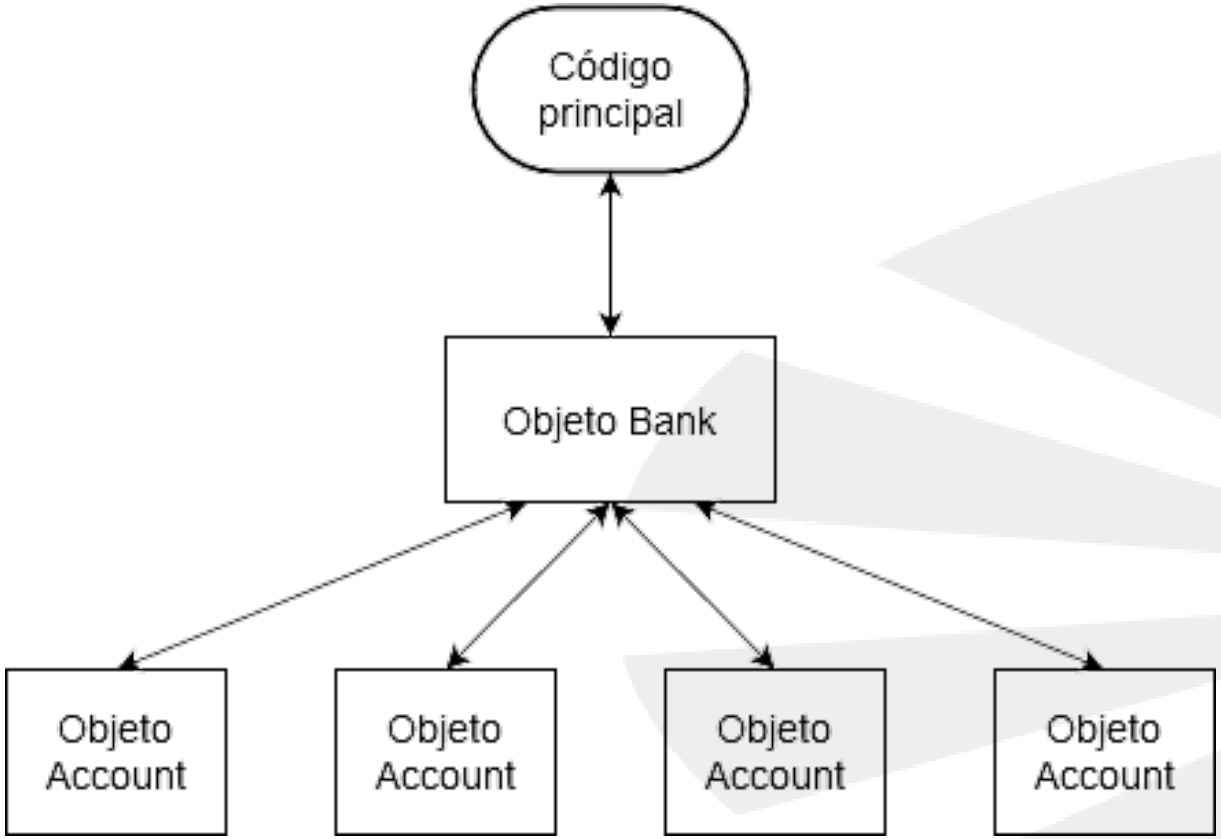
- Usando un diccionario de pares clave (número de cuenta) y valor (objeto Account)
  - Si se borra una cuenta, ninguna se ve afectada
  - No se cambia el código de la clase Account

```
accountsDict = {}
oAccount = Account('Joe Schmoe', 1000, 'magic')
accountsDict[123456789] = oAccount

accountsDict[123456789].deposit(500, 'magic')
```

Probando la clase `Account`: Alternativas

- Usando un *objeto gestor de objetos*: mantiene una lista o diccionario de objetos gestionados. En este caso instanciaremos un objeto de una nueva clase `Bank`
  - El objeto Bank (global) gestiona una lista/diccionario de objetos `Account`
  - No se cambia el código de la clase `Account`
  - El código principal pedirá una acción (obtener saldo, depositar, retirar, ...), se invoca el método correspondiente del objeto Bank y éste ejecutará el método correspondiente del objeto Account buscado.
  - La subdivisión del código en partes hace mucho más fácil de programar cada pieza porque el alcance es menor y la responsabilidad es más clara

La gestión de objetos o Composición



Ejemplo: el objeto coche se compone de un objeto motor, un objeto volante, varios objetos puerta, cuatro objetos rueda, etc. Un coche "tiene" un volante, un motor, un cierto número de puertas, etc. Por tanto, el objeto coche está compuesto de otros objetos.

Diseñando el gestor Bank de objetos Account

```
from account import *

class Bank:

    def __init__(self):
        self.accountsDict = {}
        self.nextAccountNumber = 0

    def createAccount(self, theName, theStartingAmount, thePassword):
        oAccount = Account(theName, theStartingAmount, thePassword)
        newAccountNumber = self.nextAccountNumber
        self.accountsDict[newAccountNumber] = oAccount

        # Increment to prepare for next account to be created
        self.nextAccountNumber = self.nextAccountNumber + 1
        return self.newAccountNumber

    def openAccount(self):
        # User is prompted for data of new customer
        # userName, userAmount, userAccountPassword

        userAccountNumber = self.createAccount(userName, userAmount, userAccountPassword)
        print('Your new account number is:', userAccountNumber)
        ...
```

Diseñando el gestor Bank de objetos Account

```
from account import *

class Bank:

    def closeAccount(self):
        # User is prompted for customer data
        # userAccountNumber, userAccountPassword

        oAccount = self.accountsDict[userAccountNumber]
        theBalance = oAccount.getBalance(userPassword)

        if theBalance is not None:
            print('Returned ', theBalance)
            # Remove user's account from the dictioary of accounts
            del self.accountsDict[userAccountNumber]
        ...
```

Diseñando el gestor Bank de objetos Account

```
from account import *

class Bank:
    def balance(self):
        # User is prompted for customer data
        # userAccountNumber , userAccountPassword

        oAccount = self.accountsDict[userAccountNumber]
        theBalance = oAccount.getBalance(userAccountPassword)
        if theBalance is not None:
            print('Your balance is:', theBalance)

    def deposit(self):
        # User is prompted for customer data
        # userAccountNumber , depositAmount, userAccountPassword

        oAccount = self.accountsDict[userAccountNumber]
        theBalance = oAccount.deposit(depositAmount, userAccountPassword)
        if theBalance is not None:
            print('Your new balance is:', theBalance)
```

Creando el gestor Bank de objetos Account

```
from bank import *

oBank = Bank()

# Create two test accounts
joesAccountNumber = oBank.createAccount('Joe', 100, 'JoesPassword')
marysAccountNumber = oBank.createAccount('Mary', 12345, 'MarysPassword')

while True:
    # Prompts to the user for action

    if ...:
        oBank.openAccount()
    elif ...:
        oBank.balance()
    elif ...:
        oBank.withdraw()
    ...
```

Usando una lista de objetos

- Si no es necesario identificar los objetos de forma única és factible el uso de una lista de objetos.

```
import random

class Monster:
    def __init__(self, nRows, nCols, maxSpeed):
        self.nRows = nRows  # save away
        self.nCols = nCols  # save away
        self.myRow = random.randrange(self.nRows)
        self.myCol = random.randrange(self.nCols)
        self.mySpeedX = random.randrange(-maxSpeed, maxSpeed) + 1
        self.mySpeedY = random.randrange(-maxSpeed, maxSpeed + 1)
        # Set other instance variables like health, power, etc.

    def move(self):
        self.myRow = (self.myRow + self.mySpeedY) %  self.nRows
        self.myCol = (self.myCol + self.mySpeedX) % self.nCols
```

Usando una lista de objetos

```
from monster import *

N_MONSTERS = 20
N_ROWS = 100    # could be any size
N_COLS = 200    # could be any size
MAX_SPEED = 4

monsterList = []  # start with an empty list
for i in range(N_MONSTERS):
    oMonster = Monster(N_ROWS, N_COLS, MAX_SPEED)  # create a Monster
    monsterList.append(oMonster)  # add Monster to our list
...
```

- Cada objeto `Monster` guarda su ubicación y velocidad. En el método `move()` cada Monstruo se mueve recordando su nueva ubicación.

```
for oMonster in monsterList:
    oMonster.move()
```

Interfaz frente a Implementación

- Interfaz: Muestra lo que puede hacer un objeto (métodos y parámetros) creado a partir de la clase.
- Implementación: El código real de la clase, que muestra cómo un objeto hace lo que hace.
- Creadores o mantenedores de una clase: comprender perfectamente la implementación (código de métodos y variables de instancia que afectan).
- Usuarios de una clase: sólo importa la interfaz (métodos disponibles, argumentos y valor(es) devueltos).
- VENTAJA: Si la interfaz no cambia, puede hacerlo la implementación de la clase (un método puede implementarse de forma más rápida o eficiente) sin efecto en otra parte del programa.

Encapsulación

Definición de Encapsulación

- Uno de los tres principios fundamentales de la POO (+ herencia y polimorfismo)

Definición

Ocultar detalles internos del estado y el comportamiento a cualquier código externo y tener todo el código en un solo lugar.

Encapsulación con funciones

- Lo importante de una función bien escrita es que contiene una serie de pasos que conforman una única tarea mayor. Nada importa cómo funcione internamente.
- El nombre de la función debe describir la acción que su código representa.
  - P.e. la función `len()`.
  - ¿Tendrá dos líneas o dos mil? ¿Usa una variable local?
- Una vez depurada y probada no hay que preocuparse de los detalles de implementación: sólo saber qué argumento pasar y cómo utilizar el resultado devuelto.
- Mientras no cambia la interfaz (nombre, entradas y salidas) no se cambia el código que la usa (y siempre se puede rescribir su código)

Encapsulación en la práctica

```
# Initial version
def calculateAverage(numbersList):
    total = 0.0
    for number in numbersList:
        total = total + number
    return total / len(numbersList)

# Optimized version
def calculateAverage(numbersList):
    return sum(numbersList) / len(numbersList)

# Main code
numberList = [1, 2, 3, 4, 5]
print('Average is ', calculateAverage1(numberList)
```

Encapsulación con objetos

Definición

Cliente: Cualquier software que crea un objeto a partir de una clase y hace llamadas a los métodos de ese objeto.

- Codificación *dentro* de la clase. Preocuparse de diseño e implementación:
  - Variables de instancia y eficiencia de algoritmos
  - La interfaz: métodos, parámetros (por defecto)?
- Codificación *fuera* de la clase. Conocer la interfaz:
  - ¿Qué hacen los métodos de la clase? ¿Qué argumentos necesitan? ¿Qué datos devuelven?
- Una clase proporciona encapsulación:
  - Oculta detalles de implementación en sus métodos y variables de instancia.
  - Proporciona toda la funcionalidad que un cliente necesita de un objeto a través de su interfaz (métodos de la clase)

Los objetos deberían *ser dueños de sus datos*

- Principio de diseño: el código cliente sólo debe preocuparse por la interfaz de una clase y no por la implementación de los métodos.
- Al instanciar nuevos objetos Person toman valor las variables de instancia `self.name` y `self.salary`.
- Cada objeto posee su propio conjunto de datos.

```
class Person:

        def __init__(self, name, salary):
                self.name = name
                self.salary = salary

oPersona1 = Persona('Joe Schmoe', 90000)
oPersona2 = Persona('Jane Smith', 99000)
```

¡Acceso directo a variables de instancia!

```
from person import *
oPerson1 = Person('Joe Schmoe', 90000)
oPerson2 = Person('Jane Smith', 99000)

# Get the values of the salary variable directly
print(oPerson1.salary)
print(oPerson2.salary)

# Change the salary variable directly
oPerson1.salary = 100000
oPerson2.salary = 111111

# Get the updated salaries and print again
print(oPerson1.salary)
print(oPerson2.salary)
```

- Python permite el acceso directo a variables de instancia.
- Guido van Rossum (el creador de Python) dijo en referencia a este tema, “Aquí todos somos adultos”

El acceso directo puede ser problemático

- Cambio del nombre de una variable de instancia:
  - El nombre no representa los datos
  - La variable es booleana, y se intercambia lo que `True` y `False` representan (p.e. `cerrado` por `abierto`)
  - Error ortográfico o de mayúsculas. P.e. *salari*
  - La variable cambia su tipo inicial (de booleano a más de dos valores)
- Cambio de una variable de instancia en un cálculo
- Validación de datos

El acceso directo puede ser problemático

- Cambio del nombre de una variable de instancia
- Cambio de una variable de instancia en un cálculo
  - Se añade `self.interestRate` a la clase `Account`
  - ....que dependa del **saldo** de la cuenta...
  - ....y se calcula invocando `calculateInterestRate()`.
  - Problema: cualquier cliente accede directamente a `oAccount.interestRate` para cambiarlo, independientemente del saldo)
  - Solución: obtener dato y calcularlo mediante método `getInterestRate()`
- Validación de datos

El acceso directo puede ser problemático

```
class Account():
    def __init__(self, name, balance, password):
        self.name = name
        self.balance = int(balance)
        self.password = password
        self.calculateInterestRate()

    def calculateInterestRate(self):
        if self.balance < 1000:
            self.interestRate = 1.0
        elif self.balance < 5000:
            self.interestRate = 1.5
        else:
            self.interestRate = 2.0

    def getInterestRate(self):
        self.calculateInterestRate()
        return self.interestRate
    ...
```

El acceso directo puede ser problemático

- Cambio del nombre de una variable de instancia
- Cambio de una variable de instancia en un cálculo
- Validación de datos
  - Problema: El cliente da un valor inválido (p.e. negativo) a una variable de instancia
  - Solución: llamar a un método para establecer y **validar** el nuevo valor

El acceso directo puede ser problemático

```python
class Club:

    def __init__(self, clubName, maxMembers):
        self.clubName = clubName
        self.maxMembers = maxMembers
        self.membersList = []

    def addMember(self, name):
        if len(self.membersList) < self.maxMembers:
            self.membersList.append(name)
        else:
            print('Maximum of members reached.\n')

    def report(self):
        print('Club: ', self.clubName)
        print('Members: ', len(self.membersList))
        for name in self.membersList:
            print('    ' + name)
        print()
```

```python
from club import *

oProgrammingClub = Club('Programming', 3)

for name in ['Joe', 'Cindy', 'Dino']:
    oProgrammingClub.addMember(name)

# This is OK, but Club is full
oProgrammingClub.addMember('Nancy')

# This is not OK
oProgrammingClub.membersList.append('Nancy')
oProgrammingClub.report()

# Assign a value from another data type
oProgrammingClub.maxMembers = 'Tres'
oProgrammingClub.addMember('Fred')
```

Salida por pantalla

```
Maximum of members reached.

Club:  Programming
Members:  4
   Joe
   Cindy
   Dino
   Nancy

TypeError: '<' not supported between instances of
↪  'int' and 'str'
```

Interpretación estricta de la Encapsulación

- El código cliente *nunca* debe acceder a una variable de instancia directamente (atributos públicos).
- Para acceder a la información contenida dentro de un objeto (estado), sin conocimiento explícito de la implementación de la clase, se usan *getters* y *setters*.

Definición

getter: Método que recupera datos (valores de atributos) de un objeto instanciado.

Definición

setter: Método que asigna o cambia datos a un objeto instanciado.

Interpretación estricta de la Encapsulación

```python
# Person class

class Person:

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    # Allow the caller to get the salary
    def getSalary(self):
        return self.salary

    # Allow the caller to set a new salary
    def setSalary(self, salary):
        self.salary = salary
```

```python
from Person import *

oPerson1 = Person('Joe Schmoe', 90000)
oPerson2 = Person('Jane Smith', 99000)

# Get the salaries using getter and print
print(oPerson1.getSalary())
print(oPerson2.getSalary())

# Change the salaries using setter
oPerson1.setSalary(100000)
oPerson2.setSalary(111111)

# Get the salaries and print again
print(oPerson1.getSalary())
print(oPerson2.getSalary())
```

En la literatura de Python se utilizan los términos *accessor* para un método getter y *mutator* para el método setter

Cambio de la implementación interna

```python
# Person class

class Person:

    def __init__(self, name, salary):
        self.data = [name, salary]

    # Allow the caller to get the salary
    def getSalary(self):
        return self.data[1]

    # Allow the caller to set a new salary
    def setSalary(self, salary):
        self.data[1] = salary
```

```python
from Person import *

oPerson1 = Person('Joe Schmoe', 90000)
oPerson2 = Person('Jane Smith', 99000)

# Get the salaries using getter and print
print(oPerson1.getSalary())
print(oPerson2.getSalary())

# Change the salaries using setter
oPerson1.setSalary(100000)
oPerson2.setSalary(111111)

# Get the salaries and print again
print(oPerson1.getSalary())
print(oPerson2.getSalary())
```

El programa no tiene que realizar ningún cambio pues el acceso/modificación se hace a través de getter y setter.

Validación de datos

```python
# Person class

class Person:

    MIN_SALARY = 18000

    def __init__(self, name, salary):
        self.name = name
        self.setSalary(salary)

    ...

    # Allow the caller to set a new salary
    def setSalary(self, salary):
        if salary < Person.MIN_SALARY:
            raise Exception(f'Error: salary < {Person.MIN_SALARY}')
        self.salary = salary
```

```python
from Person import *

oPerson1 = Person('Joe Schmoe', 90000)
oPerson2 = Person('Jane Smith', 99000)

# Change the salaries using setter
oPerson1.setSalary(100000)
oPerson2.setSalary(17000)
```

Salida por pantalla

```
Exception: Error: salary < 18000
```

¿Cómo privatizar variables de instancia?

- En Python todas las variables de instancia son públicas (el código externo a la clase puede acceder a ellas)
- Hay lenguajes OO que permiten marcar explícitamente ciertas variables de instancia como públicas o privadas.
- ¿En Python? Usando al comienzo del nombre uno o dos guiones
  - Se aplica a variables de instancia y métodos
  - Con un guión sólo es una convención de Python, no una imposición.
    - P.e. `self._name` y `def _internalMethod(self):`
  - Con dos guiones impide el acceso
    - P.e. `self.__privateData` y `def __internalMethod(self):`
    - El acceso produce un error:

```
AttributeError: 'Example' object has no attribute
'__privateData'
```

¿Cómo privatizar variables de instancia?

```python
class Person:

    MIN_SALARY = 18000

    def __init__(self, name, salary):
        self.name = name
        self.setSalary(salary)

    # To get the salary
    def getSalary(self):
        return self._salary

    # To set a new salary
    def setSalary(self, salary):
        if salary < Person.MIN_SALARY:
            raise Exception(f'Error: salary < {Person.MIN_SALARY}')
        self._salary = salary
```

```python
from Person import *

oPerson1 = Person('Joe Schmoe', 90000)

# Change value of atributes
oPerson1.name = 'Jane Smith'
oPerson1._salary = 17000

# Get value of atribute
print(oPerson1.name)
print(oPerson1._salary)
```

La línea roja se salta el control del salario mínimo. El uso de _ sólo advierte de un atributo no público.

Properties vs Getters y Setters: Modo *pythonico*

- Para adjuntar comportamiento a un atributo hay que convertirlo en **property**.
- Son atributos especiales con comportamiento adicional para obtener, configurar, eliminar y documentar los datos.
- Se usan de la misma manera que atributos normales. P.e., al acceder se llama automáticamente a su método *getter*; cuando muta, a su método *setter*.
- El uso de *properties* requiere (1) hacer atributos no públicos y (2) escribir métodos setters y getters para cada atributo.

Ejemplo de *properties* como getters y setters

```
# Person Class

class Person:

    MIN_SALARY = 18000

    def __init__(self, name, salary):
        self._name = name
        self._salary = salary

    # Allow the caller to get the salary
    @property
    def salary(self):
        return self._salary

    # Allow the caller to set a new salary
    @salary.setter
    def salary(self, salary):
        if salary < Person.MIN_SALARY:
            raise Exception(f'Error: salary < {Person.MIN_SALARY}')
        self._salary = salary
```

```
from Person import *

oPerson1 = Person('Joe Schmoe', 90000)

# Get the salariy using getter and print
print(oPerson1.salary)

# Change the salary using setter
oPerson1.salary = 100000

# Get the salary and print again
print(oPerson1.salary)

# Change the salary using setter
oPerson1.salary = 17000
```

Se accede (lee/escribe) como si de los mismos atributos se tratara

Abstracción

Abstracción

- Estrechamente relacionado con el encapsulamiento (algunos lo consideran el cuarto principio de la POO)
- La *encapsulación* se refiere a la implementación, ocultando los detalles del código y los datos que componen una clase.
- La *abstracción* se refiere a la percepción de una clase desde el exterior.

Definición

Abstracción: Manejar la complejidad ocultando detalles innecesarios.

- Ejemplo del mundo real–TVs, ordenadores, microondas, ...–proporcionan una abstracción de su funcionalidad a través de sus controles, pero ignoramos cómo funcionan internamente (encapsulación).

Polimorfismo

Definición de Polimorfismo

- Uno de los tres principios fundamentales de la POO (+ herencia y encapsulación)
- Procede del griego, *poly* significa "mucho" o "muchos", y *morfismo* significa "forma".
- En POO, el término "enviar un mensaje" ocurre cuando un código cliente llama a un método de un objeto.

Definición

Propiedad por la que es posible enviar el mismo mensaje a objetos de tipos distintos, y que reaccionen de forma diferente dependiendo de para qué esté diseñado y de los datos de que disponga.

Polimorfismo en el mundo real

- Todos los coches cuentan con un "acelerador" y un motor (de combustión o eléctrico)
- Pisar el pedal envía un mensaje al coche, que lo interpreta y actúa en consecuencia.
- Si se desarrolla una nueva clase de motor (p.e. nuclear), no cambiaríamos la interfaz (mismo mensaje pisando el pedal), sino la implementación (iría más rápido)

Polimorfismo en POO

```
class Dog:
    def __init__(self, name):
        self.name = name
    def speak(self):
        print(self.name, 'guau!')

class Cat:
    def __init__(self, name):
        self.name = name
    def speak(self):
        print(self.name, 'miaaaou')

class Bird:
    def __init__(self, name):
        self.name = name
    def speak(self):
        print(self.name, 'pio')

oDog = Dog('Spike')
oCat = Cat('Fluffy')
oBird = Bird('Big Bird')
petsList = [oDog, oCat, oBird]
for oPet in petsList:
    oPet.speak()
```

- El código cliente invoca un método con idéntico nombre en distintos objetos y cada uno lo implementa de forma particular.
- P.e. se tiene una colección de mascotas (perros, gatos, pájaros), se les pide que “hablen” (mensaje) con diferente sonido (“guau”, “miaou”, “pio”)

Polimorfismo en POO

```
import random

class Square:

    def __init__(self, maxWidth, maxHeight):
        self.widthAndHeight = random.randrange(10, 100)
        self.x = random.randrange(1, maxWidth - 100)
        self.y = random.randrange(25, maxHeight - 100)
        self.shapeType = 'Square'

    def getType(self):
        return self.shapeType

    def getArea(self):
        return self.widthAndHeight * self.widthAndHeight
```

Polimorfismo en POO

```
import random
import math

class Circle:

    def __init__(self, maxWidth, maxHeight):
        self.x = random.randrange(1, maxWidth - 100)
        self.y = random.randrange(25, maxHeight - 100)
        self.radius = random.randrange(10, 50)
        self.centerX = self.x + self.radius
        self.centerY = self.y + self.radius
        self.shapeType = 'Circle'

    def getType(self):
        return self.shapeType

    def getArea(self):
        return math.pi * (self.radius ** 2)
```

Polimorfismo en POO

```
import random

class Triangle:

    def __init__(self, maxWidth, maxHeight):
        self.base = random.randrange(10, 100)
        self.height = random.randrange(10, 100)
        self.x = random.randrange(1, maxWidth - 100)
        self.y = random.randrange(25, maxHeight - 100)
        self.shapeType = 'Triangle'

    def getType(self):
        return self.shapeType

    def getArea(self):
        return .5 * self.base * self.height
```

Polimorfismo en POO

```
from Square import *
from Circle import *
from Triangle import *

# Set up the constants
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
N_SHAPES = 10

# Creation list of shapes
shapesList = []
shapeClassesTuple = (Square, Circle, Triangle)
for i in range(0, N_SHAPES):
    randomlyChosenClass = random.choice(shapeClassesTuple)
    oShape = randomlyChosenClass(WINDOW_WIDTH, WINDOW_HEIGHT)
    shapesList.append(oShape)

# Main loop
for oShape in shapesList:
    area = str(oShape.getArea())
    theType = oShape.getType()
    print('Shape is ' + theType + ' whose area is ' + area)
```

Polimorfismo en POO

- Construir clases con métodos de nombre común crea un patrón consistente que nos permite extender fácilmente el programa.
  - P.e. crear la clase `Ellipse` que implemente los métodos `getArea()` y `getType()`
  - El único cambio al código es añadir `Ellipse` a la tupla de clases:

    ```
    shapeClassesTuple = (Square, Circle, Triangle, Ellipse)
    ```

- Si varias clases tienen las mismas interfaces para sus métodos (clases polimórficas), el cliente puede ignorar su implementación en todas las clases (abstracción extendida a clases) y el esfuerzo de programación se reduce.

Polimorfismo para operadores: Sobrecarga

```
>>> valor1 = 4
>>> valor2 = 5
>>> print(valor1 + valor2)
9
>>>
>>> valor1 = 'Joe'
>>> valor2 = 'Schmoe'
>>> print(valor1 + valor2)
JoeSchmoe
>>>
```

- El operador en `valor1 + valor2` es el mismo, pero realiza una acción diferente.
- Es la técnica de *sobrecarga de operadores*, que mejora la legibilidad del código cliente.

La magia en Python

- Los `magic methods` son métodos especiales con los cuales el programador añade "magia" a sus clases.
- Se escriben rodeados de doble **underscore** (e.g. `__init__` o `__add__`)
- Se utilizan sobre todo para *sobrecargar* operadores.
- Python acude a ellos cuando detecta un operador, una llamada a función especial, ... No se llaman directamente.

Métodos mágicos en Python

- Los tipos de datos integrados (entero, flotante, cadena, booleano, etc.) se implementan como clases en Python.

```
>>> print(isinstance(123, int))
>>> True
>>> print(isinstance('some string', str))
>>> True
```

- Contienen un conjunto de métodos mágicos. P.e. para el operador + con:
  - *enteros*, se llama al método mágico `__add__()` de la clase `integer` (suma de enteros).
  - *cadenas*, se llama al método mágico `__add__()` de la clase `string` (concatenación de cadenas).
- Cada operador corresponde con un nombre de método mágico específico.

La magia de los operadores matemáticos*

Symbol	Meaning	Magic method name
+	Addition	`__add__()`
–	Subtraction	`__sub__()`
*	Multiplication	`__mul__()`
/	Division (floating-point result)	`__truediv__()`
//	Integer division	`__floordiv__()`
%	Modulo	`__mod__()`
abs	Absolute value	`__abs__()`

*https://docs.python.org/3/reference/datamodel.html#special-method-names

La magia de los operadores de comparación

Symbol	Meaning	Magic method name
==	Equal to	`__eq__()`
!=	Not equal to	`__ne__()`
<	Less than	`__lt__()`
>	Greater than	`__gt__()`
<=	Less than or equal to	`__le__()`
>=	Greater than or equal to	`__ge__()`

La magia de los operadores de comparación

- ¿Cómo comparar si dos objetos Square son iguales?
  `if oSquare1 == oSquare2:`
  - Dos objetos Square son iguales si tienen la misma longitud de lado.
  - Comparar las variables de instancia `self.widthAndHeight` de los objetos y devolver un valor booleano.
- Se traduce el operador == en una llamada al método mágico del primer objeto.

```
def __eq__(self, oOtherSquare):
    if not isinstance(oOtherSquare, Square):
        raise TypeError('Second object was not a Square')
    if self.heightAndWidth == oOtherSquare.heightAndWidth:
        return True # match
    else:
        return False # not a match
```

Un ejemplo con métodos mágicos

```
class TimePeriod:

    def __init__(self, hours=0, minutes=0):
        self.hours = hours
        self.minutes = minutes
        self.value = hours * 60 + minutes

       def getValue(self):
           return self.value

    def __add__(self, oOther):
            '''Add two TimePeriod objects'''
        if not isinstance(oOther, TimePeriod):
            raise TypeError('Second object must be a TimePeriod')
        minutes = self.minutes + oOther.minutes
        hours = self.hours + oOther.hours

        if minutes >= 60:
            minutes -= 60
            hours += 1

        return TimePeriod(hours, minutes)
```

Un ejemplo con métodos mágicos

```
    def __gt__(self, oOther):
        '''Test for greater than'''
        if not isinstance(oOther, TimePeriod):
            raise TypeError('Second object must be a TimePeriod')
        if self.hours > oOther.hours:
            return True
        elif self.hours < oOther.hours:
            return False
        elif self.minutes > oOther.minutes:
            return True
        else:
            return False

    def __eq__(self, oOther):
        '''Test for equality'''
        if not isinstance(oOther, TimePeriod):
            raise TypeError('Second object must be a TimePeriod')
        return self.hours == oOther.hours and self.minutes == oOther.minutes

    def __str__(self):
        '''Create a string representation of the time period'''
        return f"{self.hours} hours, {self.minutes} minutes"
```

Un ejemplo con métodos mágicos

```
from TimePeriod import *

oTimePeriod1 = TimePeriod(1, 10)
oTimePeriod2 = TimePeriod(1, 10)
oTimePeriod3 = TimePeriod(0, 51)

# Print objects (calls the __str__() method)
print(oTimePeriod1)
print(oTimePeriod2)
print(oTimePeriod3)

oTimePeriod4 = oTimePeriod1 + oTimePeriod3
print(oTimePeriod4)

print(oTimePeriod1 == oTimePeriod2) # True
print(oTimePeriod1 > oTimePeriod2) # False
```

Salida por pantalla

```
1 hours, 10 minutes
1 hours, 10 minutes
0 hours, 51 minutes
2 hours, 1 minutes
True
False
```

Herencia

Concepto de Herencia

- Uno de los tres principios fundamentales de la POO (+ polimorfismo y encapsulación)
- Mecanismo para derivar (extender) una nueva clase a partir de una existente.
- Se crea una nueva clase que incluye los métodos y variables de instancia de una clase, añadiendo funcionalidad nueva y/o diferente.

Donald/Kiefer Sutherland

Definición de Herencia

- Relación entre:
  - Clase base de la que se hereda
  - Subclase que hereda y mejora la clase base
- Otros nombres: Superclase/subclase, Clase base/clase derivada, Clase padre/clase hija
- La subclase extiende la clase base de dos formas:
  - *Redefiniendo* un método definido en la clase base, con el mismo nombre pero funcionalidad diferente (**sobrescritura** de método)
  - Añadiendo nuevos métodos y variables de instancia que no aparecen en la clase base
- *Codificación por diferencia*: Si hereda no necesita repetir, sólo añadir código que la diferencie (variables de instancia, nuevos métodos y/o sobrescritos)

Un ejemplo de Clase Base

```
class Employee:

    def __init__(self, name, title, ratePerHour=None):
        self._name = name
        self._title = title
        if ratePerHour is not None:
            ratePerHour = float(ratePerHour)
        self._ratePerHour = ratePerHour

    @property
    def name(self):
        return self._name

    @property
    def title(self):
        return self._title

    def payPerYear(self):
        # 52 weeks * 5 days a week * 8 hours per day
        pay = 52 * 5 * 8 * self._ratePerHour
        return pay
```

```
    @name.setter
     def name(self, name):
            self._name = name

    @title.setter
      def title(self, title):
        self._title = title
```

Un ejemplo de Subclase

```
class Manager(Employee):

    def __init__(self, name, title, salary, staffList=None):
        self._salary = float(salary)
        if staffList is None:
            staffList = []
        self._staffList = staffList
        super().__init__(name, title)

    def getStaffList(self):
        return self._staffList

    def payPerYear(self, giveBonus=False):
        pay = self._salary
        if giveBonus:
            pay = pay + (.10 * self._salary)  # 10% bonus
        return pay

    # Additional methods unique to Manager
    def addEmployee(self, oEmployeeToAdd):
        self.staffList.append(oEmployeeToAdd)

    def removeEmployee(self, oEmployeeToRemove):
        self.staffList.remove(oEmployeeToRemove)
```

Un ejemplo de Subclase: aclaraciones

- `class Manager(Employee):` Manager es una subclase de Employee
- El método `__init__()` de la subclase inicializa todo lo que no inicializa la superclase (codificación por diferencia).
  - `salary` y `staffList` se guardan en variables de instancia.
  - `super().__init__()` busca la clase base y llama al método para inicializar `name` y `title`.
- El método `payPerYear()` sobrescribe al de la clase base (basado en la tarifa por hora).

Utilizando la herencia

```
# Create objects
oEmployee1 = Employee('Joe', 'Pizza Maker', 16)
oEmployee2 = Employee('Chris', 'Cashier', 14)
oManager = Manager('Sue', 'Manager', 55000,
 ↪  [oEmployee1])

# Add more staff members
oManager.addEmployee(oEmployee2)

# Call methods of the Employee object
for oEmployee in oManager.getStaffList():
    print(f'Employee name: {oEmployee.name}')
    print(f'Employee salary:',
     ↪  '{oEmployee.payPerYear():.2f}')
print()

# Give the manager a bonus
print(f'{oManager.name} ({oManager.title}) staff members:')
for oEmployee in oManager.getStaffList():
    print(f'\t {oEmployee.name} ({oEmployee.title})')
print(f'Manager salary: {oManager.payPerYear(True):.2f}')
```

Salida por pantalla

```
Employee name: Joe
Employee salary: 33280.00
Employee name: Chris
Employee salary: 29120.00

Sue (Manager) staff:
    Joe (Pizza Maker)
    Chris (Cashier)
Manager salary: 60500.00
```

Múltiples subclases de la misma clase base

- Las clases Square, Triangle y Circle tienen código repetido:

```
        self.x = random.randrange(1, maxWidth - 100)
        self.y = random.randrange(25, maxHeight - 100)

    def getType(self):
        return self.shapeType
```

- Cada clase establece la variable de instancia `self.shapeType` a un valor diferente.
- Construyamos una clase base común Shape con el código repetido.

Múltiples subclases de la misma clase base

```python
# Shape Class
import random

class Shape:

    def __init__(self, shapeType, maxWidth, maxHeight):
        self.shapeType = shapeType
        self.x = random.randrange(1, maxWidth - 100)
        self.y = random.randrange(25, maxHeight - 100)

    def getType(self):
        return self.shapeType

    def __str__(self):
        output = f'{self.shapeType}\n'
        output += f'Coordinates x={self.x}, y={self.y}\n'
        return output
```

Múltiples subclases de la misma clase base

```
# Square class
from Shape import *
class Square(Shape):

    def __init__(self, maxWidth, maxHeight):
        super().__init__('Square', maxWidth, maxHeight)
        self.widthAndHeight = random.randrange(10, 100)

    def getArea(self):
        return self.widthAndHeight ** 2

    def __str__(self):
        output = super().__str__()
        output += f'Side {self.widthAndHeight}\n'
        return output
```

Múltiples subclases...: Aclaraciones

- `class Square(Shape)`: Square es una subclase de Shape
- El método `__init__()` de la subclase inicializa todo lo que no inicializa la superclase (codificación por diferencia).
  - `widthAndHeight` se guarda en la variable de instancia.
  - `super().__init__()` busca la clase base y llama al método para inicializar `x`, `y` y `shapeType`.
- El método mágico `__str__()` invoca al mismo método de la case base.
- Obtiene una representación en cadena de los valores de un objeto.

Múltiples subclases de la misma clase base

```
# Circle class
from Shape import *
import math
class Circle(Shape):

    def __init__(self, maxWidth, maxHeight):
        super().__init__('Circle', maxWidth, maxHeight)
        self.radius = random.randrange(10, 50)
        self.centerX = self.x + self.radius
        self.centerY = self.y + self.radius

    def getArea(self):
        return math.pi * (self.radius ** 2)

    def __str__(self):
        output = super().__str__()
        output += f'Radius {self.radius}'
        return output
```

Múltiples subclases de la misma clase base

```
# Triangle class
from Shape import *
class Triangle(Shape):

    def __init__(self, maxWidth, maxHeight):
        super().__init__('Triangle', maxWidth, maxHeight)
        self.base = random.randrange(10, 100)
        self.height = random.randrange(10, 100)

    def getArea(self):
        return .5 * self.base * self.height

    def __str__(self):
        output = super().__str__()
        output += f'Base {self.base}\n'
        output += f'Height {self.height}'
        return output
```

Utilizando la herencia

- Cualquier llamada a `getType()` será manejada por la clase heredada (de Shape).
- El método `getArea()` tiene una implementación específica.

```
from Square import *
from Circle import *
from Triangle import *
...
# Main loop
# shapesList contains instances of Squares, Circles, ...
for oShape in shapesList:
    print(oShape)
    area = oShape.getArea()
    print(f'Area is {area:.2f}\n')
```

Clases abstractas y métodos abstractos

Bug

Cualquier cliente podría instanciar un objeto `Shape` genérico (sin método `getArea()` que implementan las subclases).

- La solución es declarar `getArea()` como *método abstracto* (sin implementación) que deberá sobrescribirse en cada subclase.
- La clase que contiene métodos abstractos es una *clase abstracta*, no se puede instanciar directamente, sólo se usan como clase base.
- P.e. la clase `Shape` debe ser una clase abstracta, no debe instanciar objetos y todas sus subclases deben implementar el método getArea().
- Python no proporciona clases abstractas. Usar el módulo ABC.

Clases y métodos abstractos

```
import random
from abc import ABC, abstractmethod

class Shape(ABC):  # identifies this as an abstract base class

    def __init__(self, shapeType, maxWidth, maxHeight):
        self.shapeType = shapeType
        self.x = random.randrange(1, maxWidth - 100)
        self.y = random.randrange(25, maxHeight - 100)

    def getType(self):
        return self.shapeType

    @abstractmethod
    def getArea(self):
        pass
```

Salida por pantalla

```
oShape = Shape('NoType', 640, 480)
...
TypeError: Can't instantiate abstract class Shape with abstract
↪  method getArea
```

Clases y métodos abstractos

```
import random
from shape import *

class Rectangle(Shape):

    def __init__(self, maxWidth, maxHeight):
        super().__init__('Rectangle', maxWidth, maxHeight)
        self.width = random.randrange(10, 100)
        self.height = random.randrange(10, 100)

     # Uncomment to remove error
    # def getArea(self):
            theArea = self.width * self.height
            return theArea
```

Salida por pantalla

```
oRectangle = Rectangle(640, 480)
...
TypeError: Can't instantiate abstract class Rectangle with
 ↪  abstract method getArea
```

Bibliografía

- Irv Kalb. Object-oriented Python: master OOP by building games and GUIs. No Starch Press, 2021

1. Programación orientada a objetos

Programación II

Grado en Inteligencia Artificial

M. Cabrero

facultade de informática da coruña

UNIVERSIDADE DA CORUÑA