Unidad 2
Programación Orientada a Objetos

Asignatura: Programación Orientada a Objetos

Licenciatura en Ciencias de la Computación

Licenciatura en Sistemas de Información

Tecnicatura en Programación WEB

Departamento de Informática

FCEFN – UNSJ

Año 2025

Propósitos que persigue la unidad

- Investigar los elementos básicos que provee Python para el desarrollo de algoritmos orientados a objetos

- Revisar los conceptos analizados en la **unidad 1**, a través del desarrollo de aplicaciones en Python

- Estudiar algunas de las colecciones que implementa el lenguaje Python y utilizarlas a través de aplicaciones en dicho lenguaje.

Bibliografía y Sitios WEB

(*1) Mark Lutz - Learning Python – Fifth edition - O'Reilly Media (2013)
(*2) Gaston C. Hillar - Learning Object-Oriented Programming-Packt Publishing (2015)
(*3) James Payne - Beginning Python_ Using Python 2.6 and Python 3.1 Wrox (2010)
(*4) Tim Hall, J-P Stacey - Python 3 for Absolute Beginners (2009)
(*5) Steven F. Lott – Mastering Object-Oriented Python – Second Edition – Pact Publishing (2019)

Built-in Functions
https://docs.python.org/3.10/library/functions.html

Referencias en la WEB:
https://docs.python.org/3.10/library/stdtypes.html
https://docs.python.org/3.10/reference/datamodel.html
https://docs.python.org/3.10/library/gc.html

Paradigmas de Programación

Algoritmos y Resolución de Problemas

Programación Prodedural

Caracterizado por secuencia, selección e iteración

Programación modular

Programación funcional

Programación declarativa, basada en funciones matemáticas

Programación Lógica

Programación declarativa, que utiliza la lógica de primer orden para la resolución de problemas

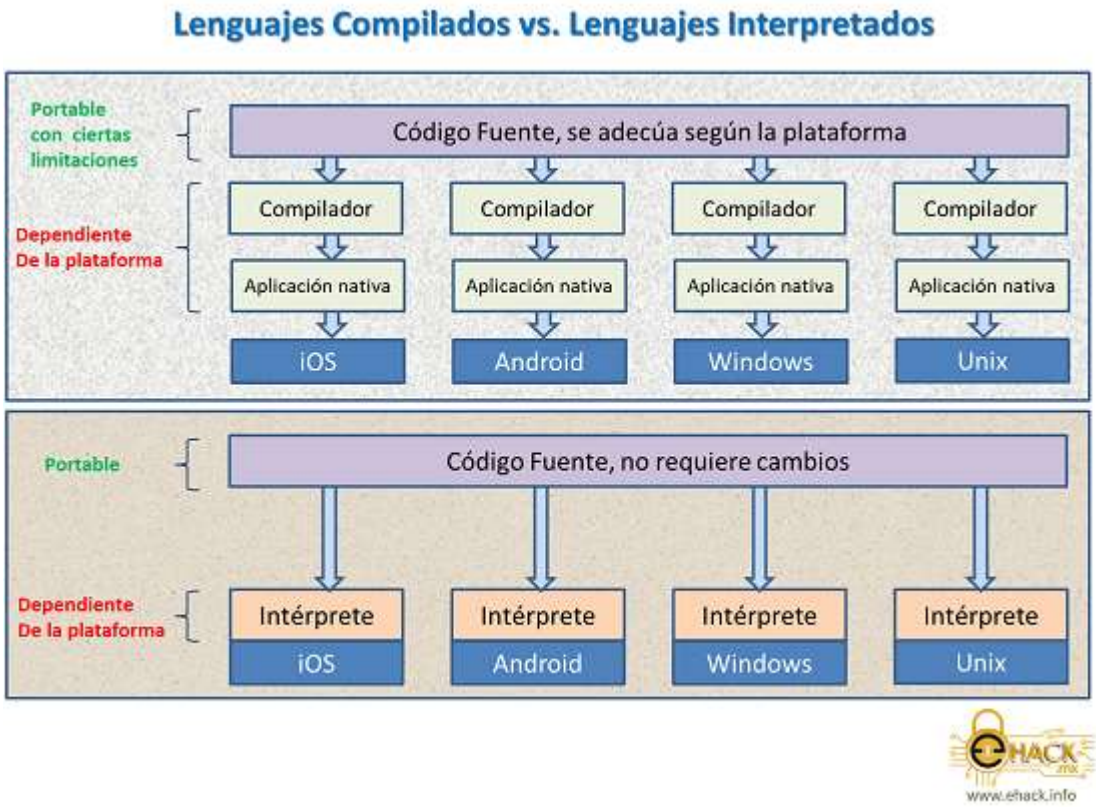
Programación Orientada a Objetos

Caracterizada por objetos que interactúan entre si.

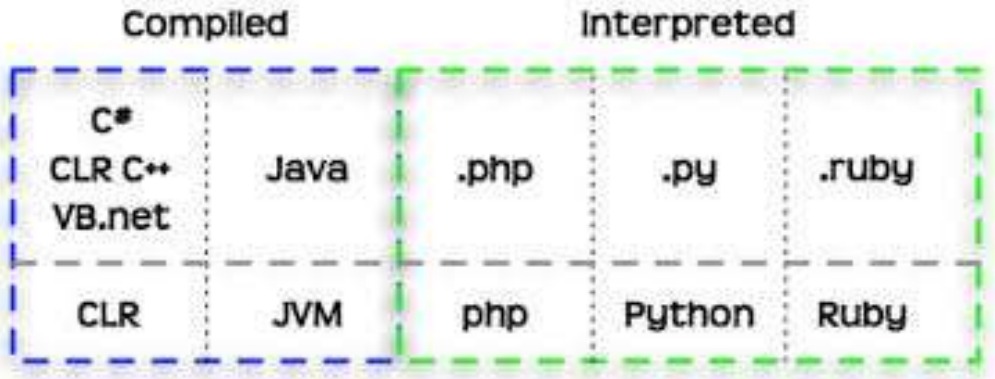
Programación Funcional y Programación Lógica, se estudian en la asignatura **Paradigma de Lenguajes**

Lenguajes de programación (I)

Lenguajes de Programación (II)

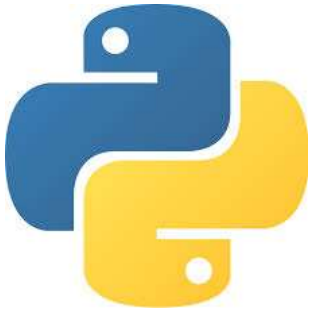


Lenguajes de Programación (III)



Lenguajes de Programación (IV)

Lenguaje Python



El lenguaje Python surgió a principios de los 90 e inicialmente fue desarrollado por Guido Van Rossum, y le puso Python, por el grupo humorístico de origen británico.

Python nació como un proyecto de software libre y posiblemente deba parte de su éxito a la decisión de hacerlo código abierto (Python Software Foundation)

Características:

Python es un lenguaje de alto nivel multiparadigma, si bien todo en Python es un objeto, incorporando las características de la POO (clases, objetos, herencia, herencia múltiple, polimorfismo), soporta también programación imperativa y algunas características de la programación funcional (funciones lambda). Python permite la creación de módulos y librerías.

Python **es un lenguaje interpretado**, lo que lo convierte en multiplataforma (se puede ejecutar en distintos sistemas operativos).

Los principios de diseño del lenguaje están guiados por una serie de aforismos recogidos en el "Zen de Python".

En estos principios se puede ver que **la legibilidad del código y favorecer la simplicidad del mismo son partes esenciales del diseño del lenguaje desde el principio**.

El Zen de Python (easter egg)

```
import this
```

Run

Consola Python

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Tipos de Datos en Python

✓ Python es un **lenguaje fuerte y dinámicamente tipado, con conteo de referencias**. Todos los objetos que se crean y se usan tienen un tipo.

Un lenguaje de programación es **fuertemente tipado** si no se permiten violaciones de los tipos de datos, es decir, dado una variable de un tipo concreto, no se puede usar como si fuera una variable de otro tipo distinto a menos que se haga una conversión (cast). No hay una única definición de este término.

Un lenguaje de programación es dinámicamente **tipado** si una misma variable puede tomar valores de distinto tipo en distintos momentos. La mayoría de lenguajes de **tipado dinámico** son lenguajes interpretados, como **Python** o Ruby.

✓En Python todos los datos son objetos que pertenecen a clases, y son tipo referencia.

✓Una variable tipo referencia guarda su dirección en la pila pero el objeto propiamente dicho se almacena en el heap (memoria dinámica)

✓ Una clase es un **tipo referencia**, de ahí que los objetos instancias de esa clase se almacenan el montículo (heap).

Tipos de datos Inmutables y Mutables (I)

Los tipos de datos **inmutables** son los más sencillos de utilizar en programación (**suelen ser los tipos de datos simples**: String, Integer, Boolean, etc.) pues hacen exactamente lo que se espera que hagan en cada momento, y paradójicamente para funcionar así son los que más castigan a la memoria (no están optimizados para ocupar menos memoria al copiarse, y degradan más la vida útil de la memoria al escribirse más veces).

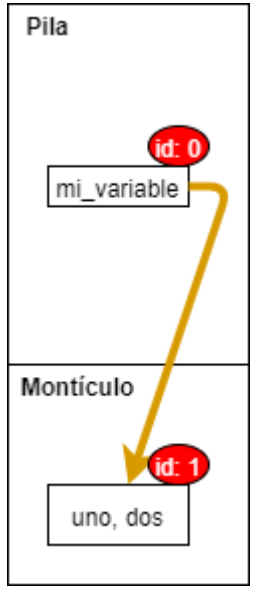
Cuando se declara una variable (en la siguiente imagen "mi_variable") en Python (o cualquier otro lenguaje de programación) se crea en memoria una **variable en una dirección de la pila** (en la imagen se guardará en la dirección de memoria "id: 0"). Si a continuación se le asigna (la variable es un puntero que «**apuntará a**» la dirección de memoria de su valor) **un valor**, que para este ejemplo será un texto (la cadena, "uno, dos") y se creará **en otra dirección, pero esta vez del montículo** (en la imagen en la dirección de memoria "id: 1"); entonces la variable **mi_variable** apuntará al objeto inmutable que contiene su valor.

```
def main():
    mi_variable = 'uno, dos'
    print('Dirección de memoria {}, valor almacenado: {}'.format(hex(id(mi_variable)), mi_variable))

if __name__== '__main__':
    main()
```

Consola Python

```
Dirección de memoria 0x1d258bff030, valor almacenado: uno, dos
```



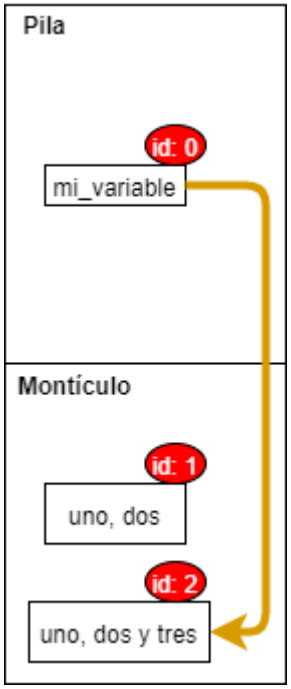
Tipos de datos Inmutables y Mutables (II)

Luego, si el valor de la variable **mi_variable** cambia, asignándole el valor que tenía, concatenado con 'y tres', la disposición de memoria se puede observar en la imagen.

```
def main():
    mi_variable = 'uno, dos'
    print('Dirección de memoria {}, valor almacenado: {}'.format(hex(id(mi_variable)), mi_variable))
    mi_variable += ' y tres'
    print('Dirección de memoria {}, valor almacenado: {}'.format(hex(id(mi_variable)), mi_variable))
if __name__== '__main__':
    main()
```

Consola Python

```
Dirección de memoria 0x1d258bff030, valor almacenado: uno, dos
Dirección de memoria 0x1d258bff330, valor almacenado: uno, dos y tres
```



Puede observarse, que al asignar el nuevo valor, la nueva cadena tiene una nueva ubicación de memoria, lo que hace que la ubicación identificada con **id: 1**, quede disponible para ser liberada por el Garbage Collector (se verá al final de la unidad).

Conclusión: cada vez que una variable, que apunta a un valor inmutable, «cambia» su valor, la misma apuntará a una un nueva ubicación de memoria.

Tipos de datos Inmutables y Mutables (III)

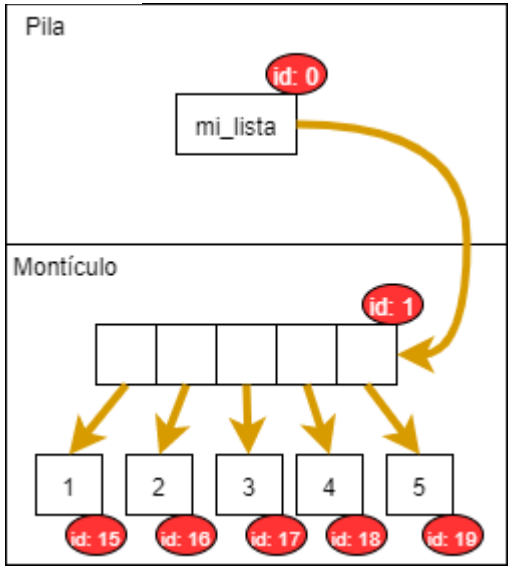
Los tipos de datos **mutables** son los más "complejos" de utilizar en programación (**suelen ser las estructuras de datos** como: dict, list, clases, etc.), no solo son más complejos porque son estructuras datos que almacenan datos inmutables o mutables, sino que suelen complicarse con el tema de los punteros; y paradójicamente son los que menos perjudican a la memoria (se escriben una sola vez y se reutilizan siempre).
Hay que decir que los tipos de datos mutables están diseñados así, porque copiar una estructura de datos entera (aunque se puede) tardaría mucho e implicaría utilizar mucha memoria para seguramente no aprovechar la copia (no es lo mismo copiar un objeto cadena que copiar una lista con millones de objetos cadena, para luego no haber necesitado la copia; llega a ser un derroche de procesador y de memoria).

```
def mutable():
    mi_lista = [1,2,3,4,5]
    print('Dirección de memoria {}, valor almacenado: {}'.format(hex(id(mi_lista)), mi_lista))
if __name__== '__main__':
    mutable()
```

Consola Python

Dirección de memoria 0x1adc2ba6248, valor almacenado: [1, 2, 3, 4, 5]

Cómo mostrar los elementos de la lista, y sus respectivas direcciones?



Tipos de datos Inmutables y Mutables (IV)

Luego, si se ejecutan instrucciones para cambiar valores, y agregar nuevos valores a la **mi_lista**, tales como:

mi_lista[0]=15

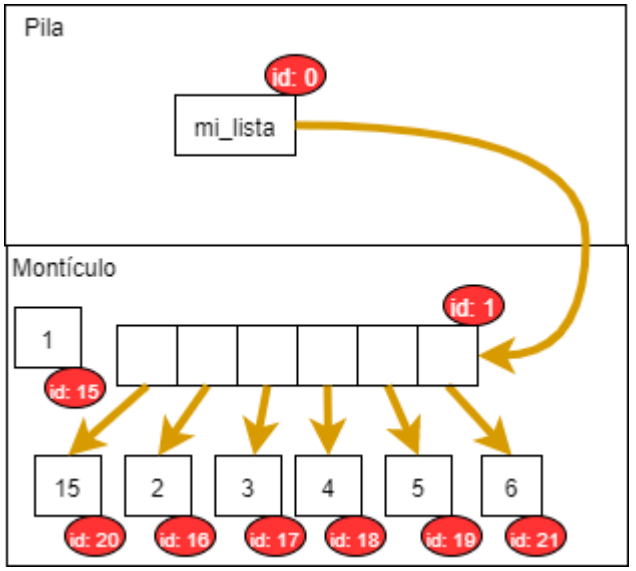
mi_lista.append(6)

```
def mutable():
    mi_lista = [1,2,3,4,5]
    print('Dirección de memoria {}, valor almacenado: {}'.format(hex(id(mi_lista)), mi_lista))
    mi_lista[0]=15
    mi_lista.append(6)
    print('Dirección de memoria {}, valor almacenado: {}'.format(hex(id(mi_lista)), mi_lista))

if __name__== '__main__':
    mutable()
```

Consola Python

Dirección de memoria 0x13c3b226248, valor almacenado: [1, 2, 3, 4, 5]
Dirección de memoria 0x13c3b226248, valor almacenado: [15, 2, 3, 4, 5, 6]



Tipos de datos Inmutables y Mutables (V)

Qué sucede si en lugar de ejecutar el código:

```
def mutable():
    mi_lista = [1,2,3,4,5]
    print('Dirección de memoria {}, valor almacenado: {}'.format(hex(id(mi_lista)), mi_lista))
    mi_lista[0]=15
    mi_lista.append(6)
    print('Dirección de memoria {}, valor almacenado: {}'.format(hex(id(mi_lista[0])), mi_lista[0]))
if __name__== '__main__':
    mutable()
```

Se ejecuta el siguiente código:

```
def mutable():
    mi_lista = [1,2,3,4,5]
    print('Dirección de memoria {}, valor almacenado: {}'.format(hex(id(mi_lista)), mi_lista))
    nuevo_valor = 15
    mi_lista[0]=15
    mi_lista.append(6)
    print('Dirección de memoria {}, valor almacenado: {}'.format(hex(id(mi_lista[0])), mi_lista[0]))
    print('Dirección de memoria {}, valor almacenado: {}'.format(hex(id(nuevo_valor)), nuevo_valor))
if __name__== '__main__':
    mutable()
```

Consola Python

Dirección de memoria 0x1aff1116248, valor almacenado: [1, 2, 3, 4, 5]
Dirección de memoria 0x7ff83c48d5e0, valor almacenado: 15
Dirección de memoria 0x7ff83c48d5e0, valor almacenado: 15

Observan algo raro???

Tipos de datos Inmutables y Mutables (VI)

Inmutables	
String	Texto
Int	Número entero
Float	Número de coma flotante
Decimal	Número decimal
Complex	Número complejo
Bool	Booleano
Tuple	Tupla, actúa como una lista inmutable
Frozenset	Conjunto congelado, actúa como un conjunto inmutable
Byte	Tipo byte
Range	Rango
None	Nulo
Clase	Si se define __hash__ y __eq__ entonces será «hasheable», y por lo tanto inmutable

Tipos de datos Inmutables y Mutables (VII)

Mutables	
List	Lista, admite cualquier tipo de valores
Dict	Diccionario, admite solo claves que sean «hasheables», y valores de cualquier tipo
Set	Conjunto, admite solo valores «hasheables»
Bytearray	Arreglo de bits, entre otros usos, se puede usar como un string mutable
MemoryView	Vista de memoria, referencia a objetos
Clase	Por efecto es mutable, aunque se puede crear como inmutable.

Referencia: https://docs.python.org/3/library/stdtypes.html

En Python todas las variables son objetos instancias de una clase

```
def objetosPython():
    i = 1
    interes = 2.3
    bandera = False
    cadena = 'CUIL erróneo'
    lista = [1, 2, 3]
    print('Tipo de datos de i', type(i))
    print('Tipo de datos de interes: ', type(interes))
    print('Tipo de datos de bandera: ', type(bandera))
    print('Tipo de datos de cadena: ', type(cadena))
    print('Tipo de datos de lista: ', type(lista))
    print('Tipo de datos de objetosPython(): ', type(objetosPython))
if __name__ == '__main__':
    objetosPython()
```

Consola Python

Tipo de datos de i <class 'int'>
Tipo de datos de interes: <class 'float'>
Tipo de datos de bandera: <class 'bool'>
Tipo de datos de cadena: <class 'str'>
Tipo de datos de lista: <class 'list'>
Tipo de datos de objetosPython(): <class 'function'>

Refrence Counting en Python (I)


```
def main():
    i = 300
    j = 300
    print('dirección del objeto apuntado por i = ', hex(id(i)))
    print('dirección del objeto apuntado por j =', hex(id(j)))
    i = i + 1
    print('dirección del objeto apuntado por i incrementado en 1 =', hex(id(i)))
    print('dirección del objeto apuntado por j antes de incrementar =', hex(id(j)))
    j = j + 1
    print('dirección del objeto apuntado por j incrementado en 1 = ', hex(id(j)))
if __name__ == '__main__':
    main()
```

Consola Pyton

dirección del objeto apuntado por i = 0x1d964a8d750
dirección del objeto apuntado por j = 0x1d964a8d750
dirección del objeto apuntado por i incrementado en 1 = 0x1d964b37f90
dirección del objeto apuntado por j antes de incrementar = 0x1d964a8d750
dirección del objeto apuntado por j incrementado en 1 = 0x1d964b37ef0

Refrence Counting en Python (II)

```
def ejemploRefcount():
    uno = 1
    dos = uno
    tres = uno
    cuatro = 1
    for k, v in locals().items():
        print('Direccion de {}, {}'.format(k,hex(id(v))))
if __name__ == '__main__':
    ejemploRefcount()
```

Consola Python

```
Direccion de uno, 0x7ff83c48d420
Direccion de dos, 0x7ff83c48d420
Direccion de tres, 0x7ff83c48d420,
Direccion de cuatro, 0x7ff83c48d420
```

Todas las variables apuntan a la misma ubicación de memoria, el valor 1 se almacena una sola vez, y cualquier otra cosa que apunte a 1, hará referencia a esa ubicación de memoria.

```
def ejemploRefcount():
    uno = 1
    dos = uno
    tres = uno
    cuatro = 1
    print('Direccion de {}, {}'.format(hex(id(uno)),'uno'))
    for k, v in locals().items():
        print('Direccion de {}, {}'.format(k, hex(id(v))))
    lista=[1,2,3,4,1]
    for i in range(len(lista)):
        print('Dirección de valor {}, {}'.format(lista[i], hex(id(lista[i]))))
```

Consola Python

```
Direccion de uno, 0x7ff83c48d420
Direccion de dos, 0x7ff83c48d420,
Direccion de tres, 0x7ff83c48d420
Direccion de cuatro, 0x7ff83c48d420
Dirección de valor 1, 0x7ff83c48d420
Dirección de valor 2, 0x7ff83c48d440
Dirección de valor 3, 0x7ff83c48d460
Dirección de valor 4, 0x7ff83c48d480
Dirección de valor 1, 0x7ff83c48d420,
```

https://docs.python.org/3.10/library/functions.html#locals

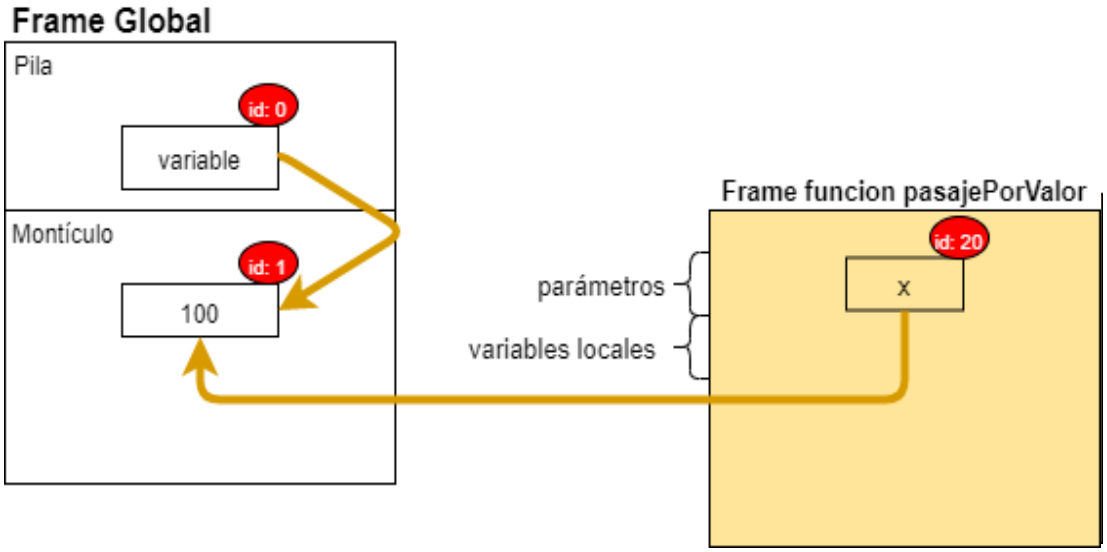
Pasaje de parámetro por valor y por referencia (I)

```
def pasajePorValor(x):
    print('En el interior de la función')
    x = x + 1
    print('Dirección de x: {}, valor: {} '.format(hex(id(x)),x))
if __name__ == '__main__':
    v = 100
    pasajePorValor(v)
    print('En el programa principal')
    print('Dirección de variable: {}, valor: {} '.format(hex(id(v)), v))
```

Consola Python

En el interior de la función
Dirección de x: 0x7ff83b8ae0a0, valor: 101
En el programa principal
Dirección de v: 0x7ff83b8ae080, valor: 100

Los datos del tipo inmutables se pasan por valor



Pasaje de parámetro por valor y por referencia (II)

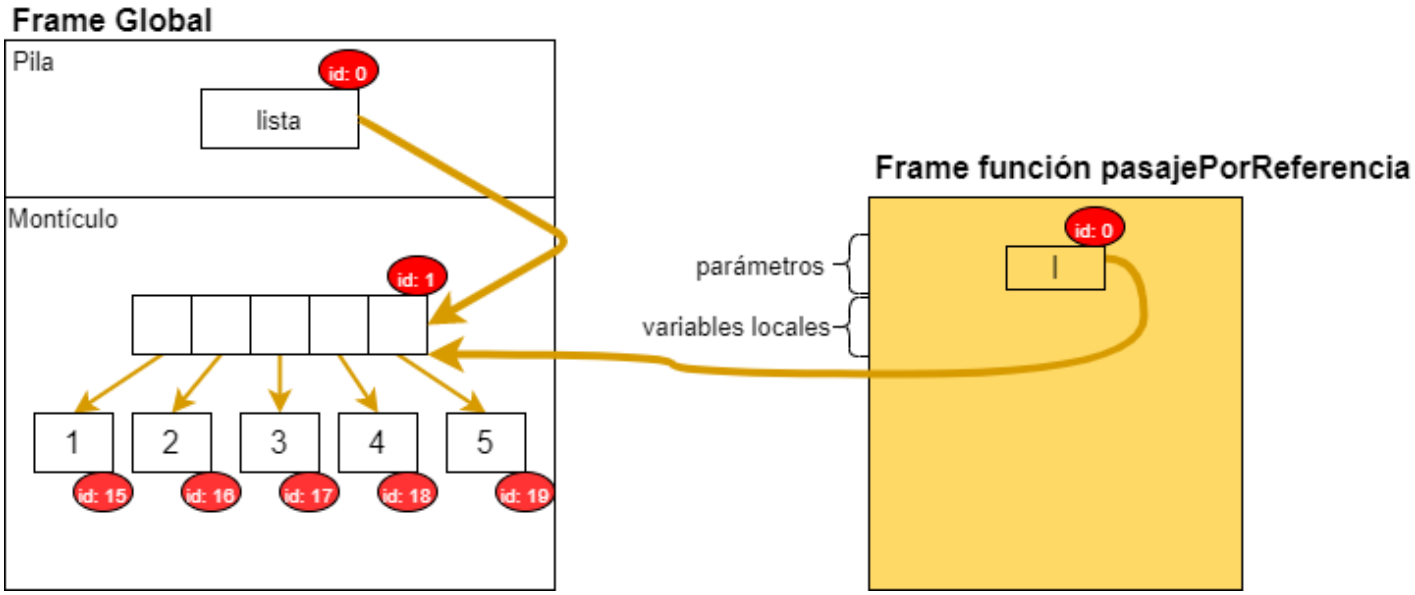
```
def pasajePorReferencia(l):
    print('En el interior de la función')
    print('Dirección de l: {}, valor: {} '.format(hex(id(l)), l))
    l[0] = 100
    l[1] = 200
    l[2] = 300
    print('Dirección de l: {}, valor: {} '.format(hex(id(l)), l))
if __name__ == '__main__':
    lista = [1,2,3,4,5]
    print('En el programa principal')
    print('Dirección de lista: {}, valor: {} '.format(hex(id(lista)), lista))
    pasajePorReferencia(lista)
    print('Luego de probar función pasajePorRerencia')
    print('Dirección de lista: {}, valor: {} '.format(hex(id(lista)), lista))
```

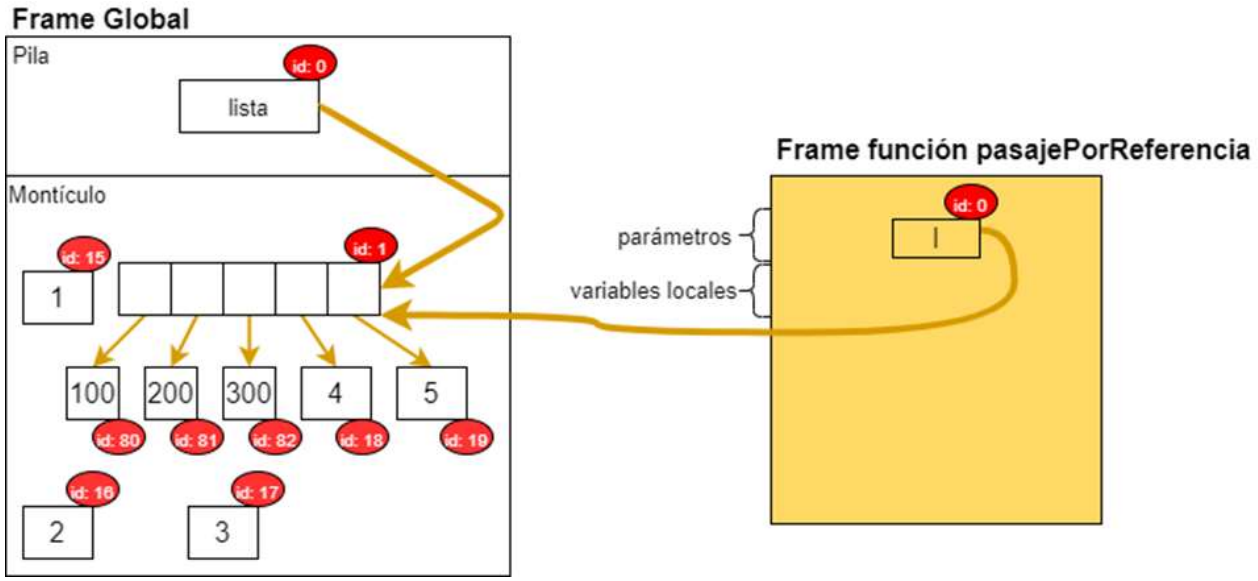
Los datos del tipo mutables, se pasan por referencia

Consola Python

```
En el programa principal
Dirección de lista: 0x1bf75336248, valor: [1, 2, 3, 4, 5]
En el interior de la función
Dirección de l: 0x1bf75336248, valor: [1, 2, 3, 4, 5]
Dirección de l: 0x1bf75336248, valor: [100, 200, 300, 4, 5]
Luego de probar función pasajePorRerencia
Dirección de lista: 0x1bf75336248, valor: [100, 200, 300, 4, 5]
```

Pasaje de parámetro por valor y por referencia (III)





Estructura de un programa en Python

```
print('Bienvenidos a POO 2020')
```

Consola Python
Bienvenidos a POO 2020

- Todo programa en Python tendrá al menos una sentencia, en el ejemplo, la instrucción **print**, que muestra por pantalla la texto que se le pasa como parámetro.

```
if __name__ == '__main__':
    print('Bienvenidos a POO
2020')
```

```
class Minima:
    pass

if __name__ == '__main__':
    unObjeto = Minima()
    print(type(unObjeto))
    print(dir(unObjeto))
```

Consola Python
<class '__main__.Minima'>
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']

- Todo programa Python comienza a ejecutarse en la primer sentencia fuera de una función. Es una buena práctica para los programadores Python identificar un punto de inicio, para ello se provee la siguiente sentencia en un programa:

if __name__==‘__main__’:

sentencia

De ahora en adelante se trabajará de esta forma.

Clases en Python (I)

Una clase describe un conjunto de objetos, de características similares. Es como una plantilla que describe cómo deben ser las instancias de dicha clase, de forma que cuando creamos una instancia ésta tendrá exactamente los mismos métodos y atributos que los que tiene la clase.
Una clase es un nuevo tipo de dato.

Contiene :

- otros datos (que pueden ser de cualquier tipo), denominados formalmente: **atributos. DEBEN SER PRIVADOS.**
- funciones, que operan sobre esos datos, denominadas formalmente: **funciones miembro o métodos. Estas funciones deben ser las únicas que puedan acceder a los datos.**

Se declaran en el código de la siguiente forma:

```
class NuevaClase:
        código_de_la_clase
```

Donde el código_de_la_clase incluye la declaración de atributos y funciones miembro o métodos.

Clases en Python (II): Atributos

- Las variables incluidas en una clase se denominan **ATRIBUTOS**.
- Existen múltiples formas de crear atributos en una clase. La más simple:

```
class NuevaClase:
        atributo1: int
        atributo2: str
```

Clases en Python (III): Métodos

Las clases pueden contener funciones.
A éstas se les denomina **MÉTODOS**.
La forma de crearlos en Python es en la declaración de la clase

```
class NuevaClase:
        def metodo1(self,[parámetros]):
                codigo_metodo1
```

donde **self**
Es el primer parámetro de cualquier método (**que no sea método de clase**).
Hace referencia al objeto que envía el mensaje,.
Nunca se pasa como parámetro cuando se llama a un método. Es un ***parámetro implícito***.

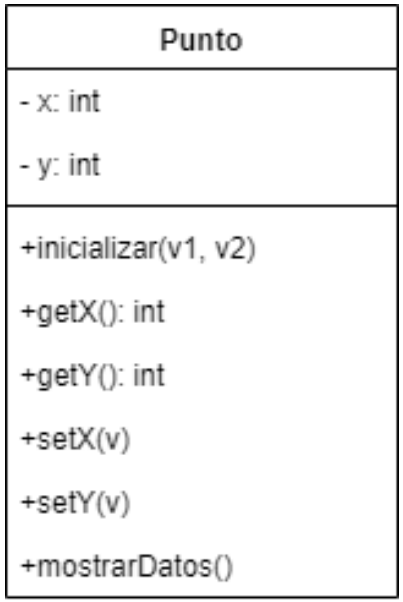
La llamada a este método en el código se haría tras la creación de un objeto:

```
unObjeto=NuevaClase()
```

La sintaxis:

```
unObjeto.metodo1([parámetros])
```

Una primer clase y su correspondiente aplicación (1)



x, e **y** son los atributos de la clase

(en este caso, variables de instancias). Son todos privados.

inicializar, **getX**, **getY**, **setX**, **setY** y **mostrarDatos** son los métodos de la clase. Todos son públicos y a través de ellos podemos acceder a las variables de instancias.

inicializar (v1, v2) recibe como parámetros la ordenada y la abscisa del punto y permite cambiar los valores de las variables de instancia.

getX() devuelve la abscisa del punto y **getY()** la ordenada.

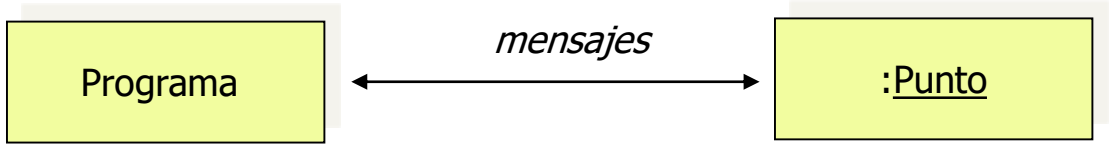
setX(v) recibe un parámetro que permite establecer el valor de la abscisa y **setY(v)** hace lo mismo con la ordenada.

El método **mostrarDatos**, muestra por pantalla los valores de las variables de instancia.

Ejemplo: clase Punto

```
class Punto:
    __x: int
    __y: int
    def inicializar(self,v1, v2):
        self.__x = v1
        self.__y = v2
    def setX(self, v):
        self.__x = v
    def getX(self):
        return self.__x
    def setY(self, v):
        self.__y = v
    def getY(self):
        return self.__y
    def mostrarDatos(self):
        print("(x,y) = (",self.__x,',', self.__y,")")

if __name__=='__main__':
    unPunto = Punto()
    otroPunto = Punto()
    unPunto.inicializar(3,4)
    otroPunto.inicializar(4,7)
    unPunto.mostrarDatos()
    otroPunto.mostrarDatos()
```

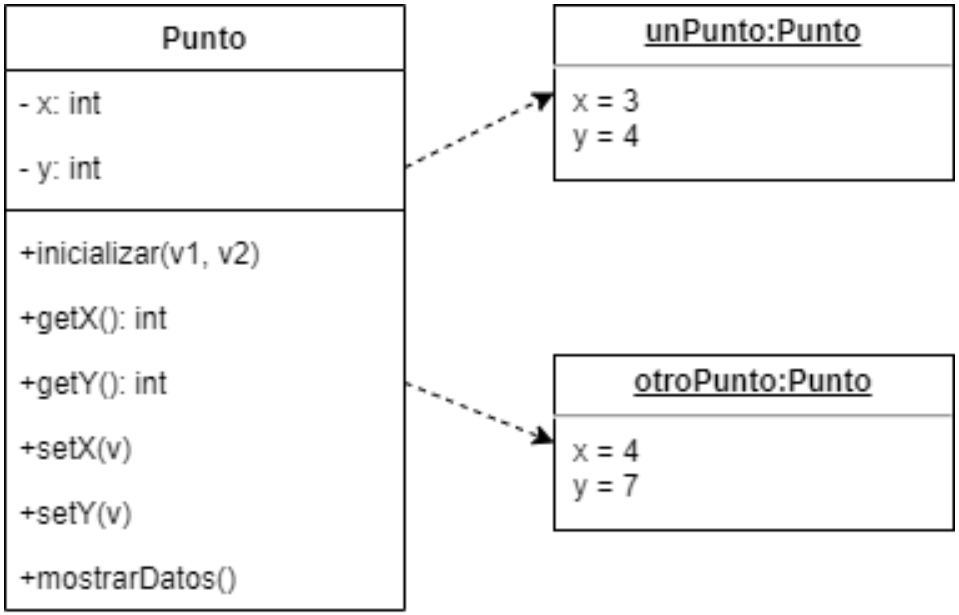


Consola Python
(x,y) = (3 , 4)
(x,y) = (4 , 7)

Un **objeto** es un agregado de datos y de métodos que permiten manipular dichos datos, y un programa en Python es un conjunto de sentencias que interaccionan con los objetos de la aplicación a través de mensajes.

Clases y Objetos en Python

Los datos y métodos contenidos en una clase se llaman miembros de la clase y se accede a ellos siempre mediante el operador "."



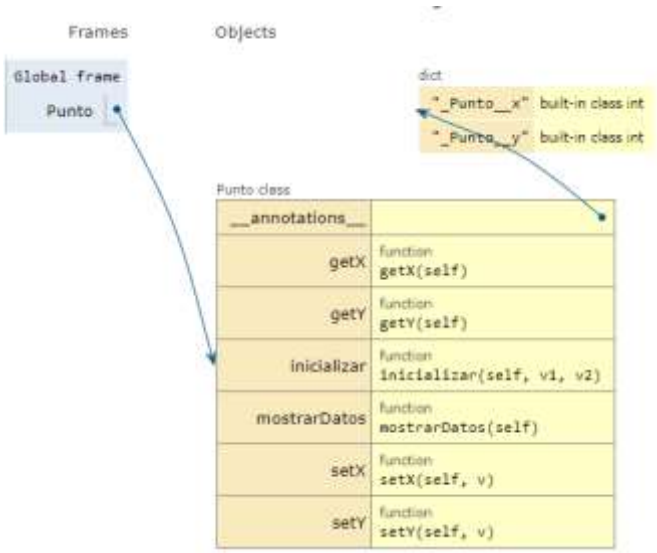
unPunto.inicializar(3,4)
unPunto.mostrarDatos()

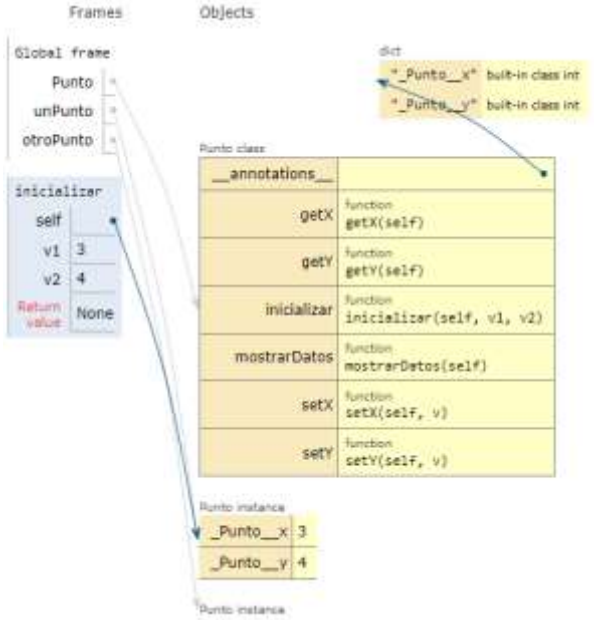
otroPunto.inicializar(4,7)

Ejecución de un programa Python – Parte I

```
class Punto:
    __x: int
    __y: int
    def inicializar(self,v1, v2):
        self.__x = v1
        self.__y = v2
    def setX(self, v):
        self.__x = v
    def getX(self):
        return self.__x
    def setY(self, v):
        self.__y = v
    def getY(self):
        return self.__y
    def mostrarDatos(self):
        print("(x,y) = (",self.__x,',', self.__y,")")

if __name__=='__main__':
    unPunto = Punto()
    otroPunto = Punto()
    unPunto.inicializar(3,4)
    otroPunto.inicializar(4,7)
    unPunto.mostrarDatos()
    otroPunto.mostrarDatos()
```



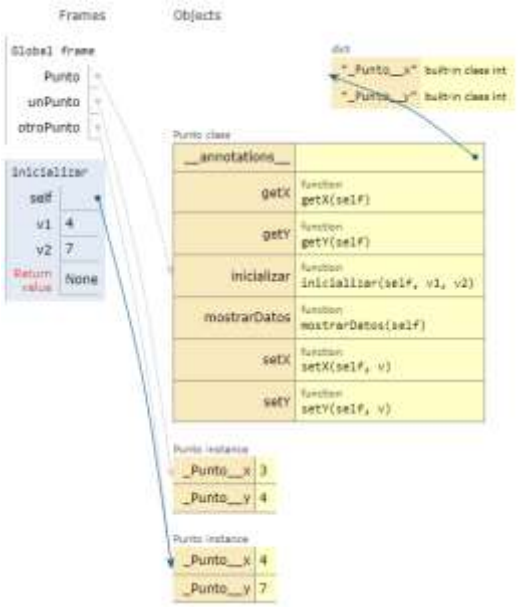


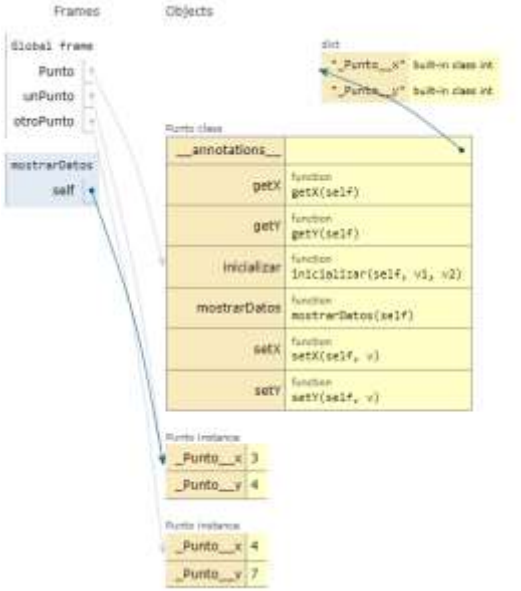
https://pythontutor.com/render.html#mode=display

Ejecución de un programa Python – Parte II

```
class Punto:
    __x: int
    __y: int
    def inicializar(self,v1, v2):
        self.__x = v1
        self.__y = v2
    def setX(self, v):
        self.__x = v
    def getX(self):
        return self.__x
    def setY(self, v):
        self.__y = v
    def getY(self):
        return self.__y
    def mostrarDatos(self):
        print("(x,y) = (",self.__x,',', self.__y,")")

if __name__=='__main__':
    unPunto = Punto()
    otroPunto = Punto()
    unPunto.inicializar(3,4)
    otroPunto.inicializar(4,7)
    unPunto.mostrarDatos()
    otroPunto.mostrarDatos()
```





https://pythontutor.com/render.html#mode=display

Encapsulamiento de una clase – Visibilidad

El control de acceso a miembros de una clase se hace de tres formas:

Público: Puede ser accedido desde cualquier código, inclusive fuera de la clase. Es un atributo o método **sin ningún decorador**. **PARA ATRIBUTOS, ESTÁ TOTALMENTE PROHIBIDO USARLO EN POO.**
Protegido: Desde una clase sólo puede accederse a miembros **potegidos** de objetos de esa misma clase, también pueden accederlo las subclases también sin hacer uso de métodos, es decir en forma directa, usando el operador «**.**». Un atributo o método **precedido por un símbolo underscore** (_), es un **miembro protegido**.

Privado: Sólo puede ser accedido desde el código de la clase a la que pertenece. Un **atributo o método precedido por doble símbolo underscore (__)**, es un **miembro privado**. Es te tipo de visibilidad, garantiza el encapsulamiento de la clase. **LOS ATRIBUTOS DEBEN SER PRIVADOS.**

(*2) Chapter 3 – Páginas 43 a 46

Tipo Abstracto de Datos

Generalmente los datos miembros o atributos de una clase han de definirse privados, es decir, ocultos a otras clases y programas, siendo posible el acceso a ellos solo a través de los métodos públicos de la clase.

El proceso de **ocultar** la estructura interna de datos de un objeto y permitir el acceso sólo a través de la interfaz pública definida se llama **Encapsulamiento**.

Tipo de dato abstracto : Definir un tipo abstracto de datos implica:

1. definir un tipo de datos
2. definir un conjunto de operaciones abstractas sobre objetos de ese tipo
3. el encapsulamiento completo. Esto implica, que se pueden manipular objetos de ese tipo exclusivamente a través del uso de las operaciones definidas para él.

Se puede trabajar con tipos de datos abstractos completos solo cuando se trabaje con Lenguajes Orientados a Objetos, donde se encapsulan en un solo módulo, los datos y los procesos que actúan sobre los mismos, dando lugar a un tipo de datos abstracto, «las clases».

Creación de Objetos (I)

- Toda clase tiene un método predeterminado especial, llamado **constructor** y que tiene como función inicializar los atributos del objeto.

- La creación de un objeto se hace a través de la invocación del método constructor

unObjeto = Punto()

- La invocación al método constructor de la clase, genera espacio en memoria (heap) para almacenar un objeto de clase, y devuelve una referencia a dicha ubicación de memoria.

- Los **objetos** se almacenan en memoria dentro del **heap o montículo**, ya que pertenecen a los tipos de datos mutables (aunque se los puede hacer inmutables)

(*1) Chapter 28 – Páginas 817 a 821

Creación de Objetos (II)

```
class Punto:
    __x: int
    __y: int
    def inicializar(self,v1, v2):
        self.__x = v1
        self.__y = v2
    def setX(self, v):
        self.__x = v
    def getX(self):
        return self.__x
    def setY(self, v):
        self.__y = v
    def getY(self):
        return self.__y
    def mostrarDatos(self):
        print("(x,y) = (",self.__x,',', self.__y,")")

if __name__=='__main__':
    unPunto = Punto()
    otroPunto = Punto()
    unPunto.inicializar(3,4)
    otroPunto.inicializar(4,7)
    unPunto.mostrarDatos()
    otroPunto.mostrarDatos()
```

En este ejemplo no hay un constructor definido explícitamente, pero la sentencia:

unPunto = Punto()

Asigna espacio e invoca al constructor por defecto el cual inicializa en cero los atributos del objeto.

Consola Python

```
(x,y) = ( 3 , 4 )
(x,y) = ( 4 , 7 )
```

Creación de Objetos (III) - Constructor

Un **constructor** es un método que es invocado por el intérprete al momento de creación de un objeto y se encarga de **llevar a cabo todas las tareas de inicialización de los datos miembro del objeto**.

El constructor, en la clase, tiene el nombre reservado __init__, y recibe como primer parámetro, una referencia al objeto que invoca el método (self), puede contener otros parámetros formales, que normalmente, representan los valores con los que se inicializarán los atributos. El constructor no devuelve ningún valor explícito.

No es obligatorio definir un constructor para cada clase, y en caso de que no se defina ninguno el intérprete creará uno por nosotros sin parámetros ni instrucciones. (**constructor por defecto u omisión**).

EL **constructor por defecto u omisión** asigna valores por defecto a los datos miembros del objeto.

Se puede definir solo un constructor para cada clase. Si se define un constructor, este ocultará al constructor provisto por el lenguaje.

Creación de Objetos (IV) - Constructor

```
class Punto:
    __x: int
    __y: int
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
    def setX(self, v):
        self.__x = v
    def getX(self):
        return self.__x
    def setY(self, v):
        self.__y = v
    def getY(self):
        return self.__y
    def mostrarDatos(self):
        print("(x,y) = (",self.__x,',', self.__y,")")

if __name__=='__main__':
    unPunto = Punto(3,4)
    otroPunto = Punto(4,7)
    unPunto.mostrarDatos()
    otroPunto.mostrarDatos()
#    tercerPunto = Punto()
#    tercerPunto.mostrarDatos()
```

Constructor de la clase

```
def __init__(self, x, y):
        self.__x = x
        self.__y = y
```

En lugar de:

```
unPunto = Punto()
unPunto.inicializar(3,4)
otroPunto = Punto()
otroPunto.inicializar(4,7)
```

Que pasa si se agrega la siguiente sentencia:

tercerPunto = Punto()

Creación de Objetos (V) – Constructor

Argumentos por defecto

```
class Punto:
    __x: int
    __y: int
    def __init__(self, x = 0, y = 0):
        self.__x = x
        self.__y = y
    def setX(self, v):
        self.__x = v
    def getX(self):
        return self.__x
    def setY(self, v):
        self.__y = v
    def getY(self):
        return self.__y
    def mostrarDatos(self):
        print("(x,y) = (",self.__x,',', self.__y,")")

if __name__=='__main__':
    unPunto = Punto(3,4)
    otroPunto = Punto(4,7)
    unPunto.mostrarDatos()
    otroPunto.mostrarDatos()
    tercerPunto = Punto()
    tercerPunto.mostrarDatos()
```

Constructor con valores por defecto

```
def __init__(self, x = 0, y = 0):
    self.__x = x
    self.__y = y
```

Permite la construcción e inicialización de objetos:

unPunto = Punto(3, 4)
otroPunto = Punto(5)
tercerPunto = Punto()

A esto, algunos autores lo denominan **sobrecarga de métodos**, ya que permite la invocación de una función de distintas formas, y el intérprete puede ejecutar cualquiera de ellas.

Ejercicios:

a) Usando UML, dibuje las instancias **unPunto**, **otroPunto** y **tercerPunto**.
b) Escriba el método **mover**, que recibe dos parámetros, x e y, que serán las nuevas coordenadas del punto. Utilice los métodos existentes, **setX** y **setY** para cambiar las coordenadas del punto.

Curiosidad sobre el constructor

El método **__init__** recibe como primer parámetro, una referencia del objeto que recibe el mensaje de creación de un objeto de la clase, pero… quien crea esa referencia?

La referencia **self**, la crea el método de clase **__new__**, que recibe una referencia de la clase que se denomina **cls** (los métodos de clase se estudian más adelante).

```
class A:
    def __new__(cls):
        print("A.__new__ llamado")
        this = super(A, cls).__new__(cls)
        print(this,hex(id(this)))
        return this

    def __init__(self):
        print(self, hex(id(self)))
        print ("A.__init__ llamado")

if __name__=='__main__':
    a = A()
```

Consola Python

```
A.__new__llamado
<__main__.A object at 0x000001C2482644C8>
0x1c2482644c8
<__main__.A object at 0x000001C2482644C8>
0x1c2482644c8
A.__init__ llamado
```

Conclusión: cuando se crea una instancia de una clase, se ejecutan dos métodos: __new__ e __init__.
El método __new__ es un **método de clase**, que necesariamente pide el espacio de memoria, y devuelve la referencia que luego es usada en todos los métodos de la clase (que no sea métodos de clase).
Generalmente no reescribiremos el método __new__ porque su función no puede ser reemplazada, puede ser extendida.

Listas en Python (built-in)

Las listas en Python son estructuras de datos secuenciales, que vienen incluidas con el core del lenguaje (no es necesario importar paquetes para poder usarlos), y que permiten almacenar en su interior referencias a objetos.


```
if __name__ == '__main__':
    listaPersonas = []
```

Luego con el método **append**, se pueden agregar objetos a la lista.

```
if __name__ == '__main__':
    listaPersonas = []
    unaPersona = Persona()
    listaPersonas.append(unaPersona)
```

El acceso a los componentes de la lista, se puede hacer de dos formas:

1) Accediendo componente a componente, como si fuera un arreglo, la primer componente es indexada desde 0.
2) Obteniendo un objeto de la lista a través de un iterador.

```
if __name__ == '__main__':
    listaPersonas = []
    unaPersona = Persona(11203567,'Nahuel', `Alfaro',9996)
    listaPersonas.append(unaPersona)
    otraPersona = Persona(1234533,'Liza', 'Laine', 45678)
    listaPersonas.append(otraPersona)
    for i in range(len(listaPersonas)):
        listaPersonas[i].mostrarDatos()
```

```
if __name__ == '__main__':
    listaPersonas = []
    unaPersona = Persona(11203567,'Nahuel', 'Catanzaro',9996)
    listaPersonas.append(unaPersona)
    otraPersona = Persona(1234533,'Liza', 'Laine', 45678)
    listaPersonas.append(otraPersona)
    for objeto in listaPersonas:
        objeto.mostrarDatos()
```

Ejemplo – Clase Lista (I)

Dada la clase Punto, definida anteriormente, generar una clase lista, que permita:

1) Almacenar instancias de la clase punto
2) Calcular la distancia euclídea entre el primer punto y el resto de los puntos.
3) Mostrar los datos de la colección de puntos

```
from math import sqrt
class Punto:
    __x: int
    __y: int
    def __init__(self, x = 0, y = 0):
        self.__x = x
        self.__y = y
    def __str__(self):
        return '({}, {})'.format(self.__x, self.__y)
    def setX(self, v):
        self.__x = v
    def getX(self):
        return self.__x
    def setY(self, v):
        self.__y = v
    def getY(self):
        return self.__y
    def mostrarDatos(self):
        print("(x,y) = (",self.__x,',', self.__y,")")
    def mover(self, x, y):
        self.setX(x)
        self.setY(y)
    def distanciaEuclidea(self, otroPunto):
        distancia = sqrt((otroPunto.__x-self.__x)**2+(otroPunto.__y-self.__y)**2)
        return distancia
```

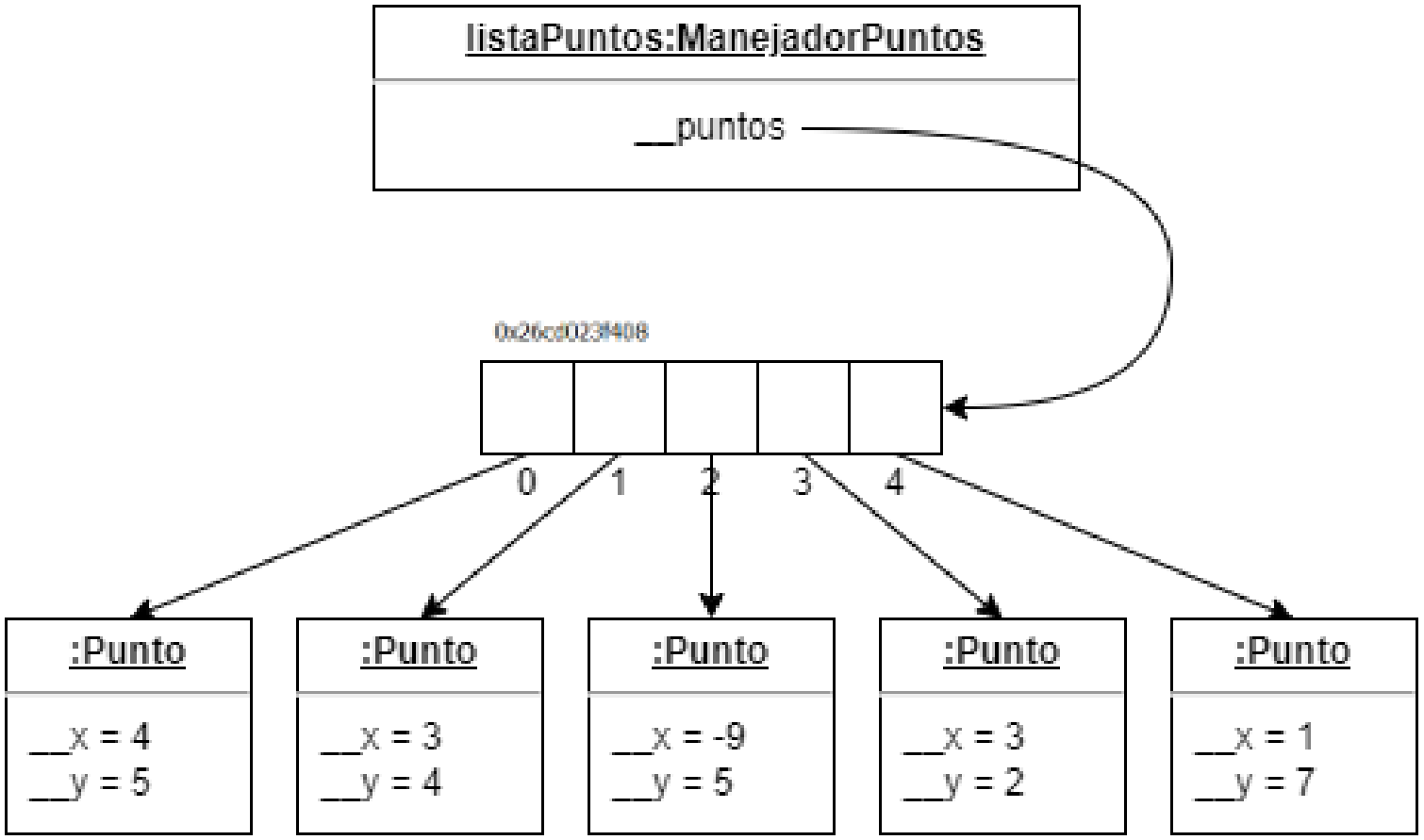
Ejemplo – Clase Lista (II)

```
from clasePunto import Punto
class ManejadorPuntos:
    __puntos: list
    def __init__(self):
        self.__puntos=[]
    def agregarPunto(self, unPunto):
        self.__puntos.append(unPunto)
    def mostrarPuntos(self):
        for punto in self.__puntos:
            punto.mostrarDatos()
    def testListaPuntos(self):
        p1 = Punto(4, 5)
        p2 = Punto(3, 4)
        p3 = Punto(-9, 5)
        p4 = Punto(3, 2)
        p5 = Punto(1, 7)
        self.agregarPunto(p1)
        self.agregarPunto(p2)
        self.agregarPunto(p3)
        self.agregarPunto(p4)
        self.agregarPunto(p5)
    def getUnPunto(self, indice):
        return self.__puntos[indice]
    def calcularDistanciasP0(self, unPunto):
        for i in range(len(self.__puntos)):
            distancia = unPunto.distanciaEuclidea(self.__puntos[i])
            print('Distancia del punto {} al punto {} es {}'.format(unPunto, self.__puntos[i], distancia))

if __name__ == '__main__':
    listaPuntos = ManejadorPuntos()
    listaPuntos.testListaPuntos()
    punto0 = listaPuntos.getUnPunto(0)
    listaPuntos.calcularDistanciasP0(punto0)
    listaPuntos.mostrarPuntos()
```

Observan algún problema???

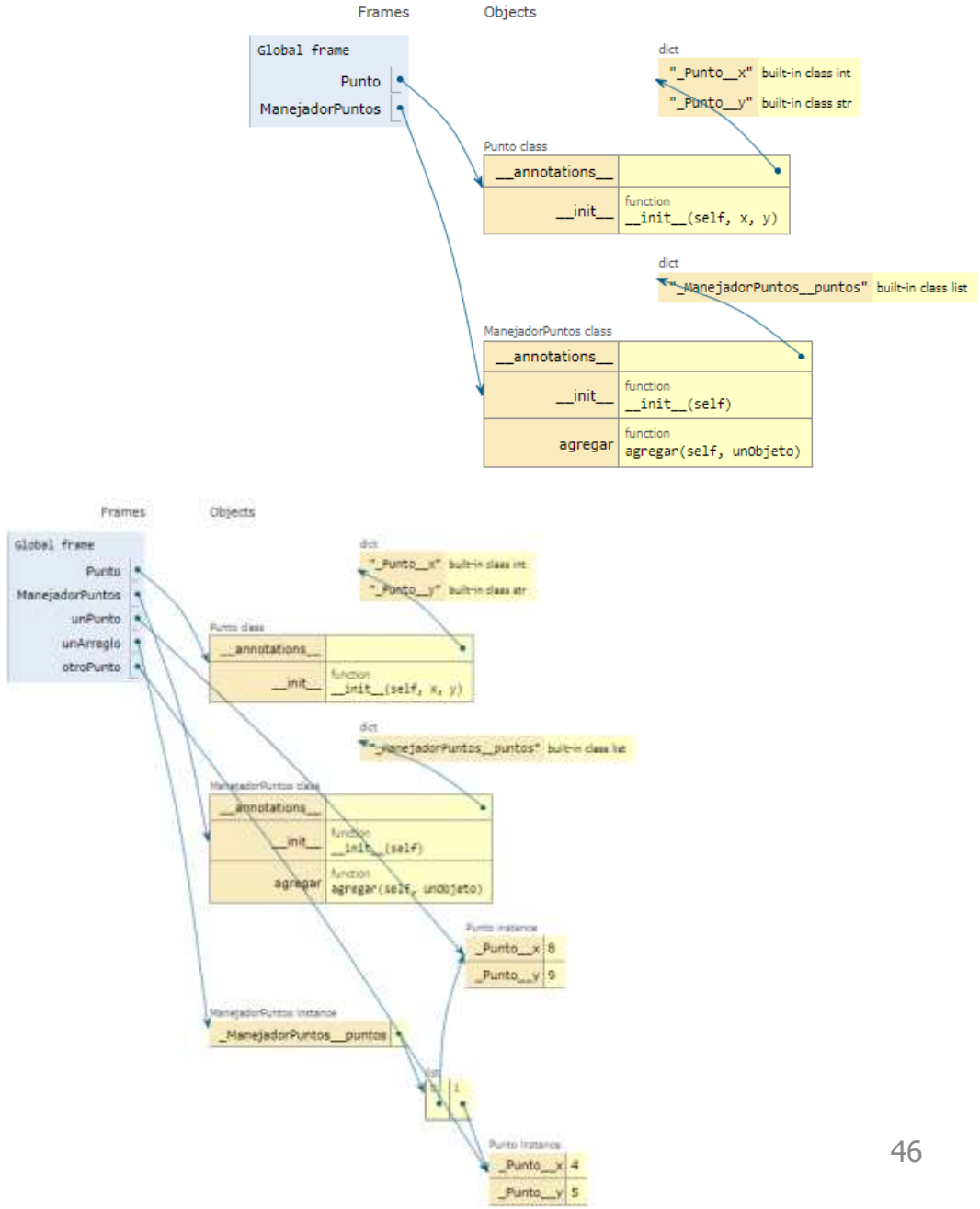
Lista de Puntos - Diagrama de Objetos



Ejecución del código de una lista de objetos de la clase Punto

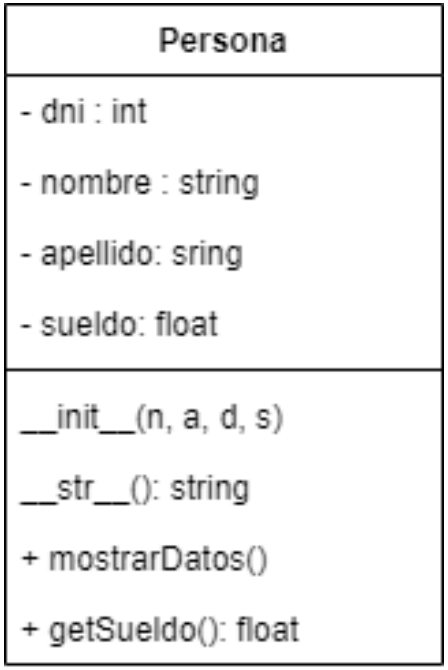
```
class Punto:
    __x: int
    __y: str
    def __init__(self, x, y):
        self.__x=x
        self.__y=y

class ManejadorPuntos:
    __puntos: list
    def __init__(self):
        self.__puntos=[]
    def agregar(self, unObjeto: Punto):
       self.__arreglo.append(unObjeto)
if __name__=='__main__':
    unPunto=Punto(8, 9)
    unArreglo=ManejadorPuntos()
    unArreglo.agregar(unPunto)
    otroPunto=Punto(4, 5)
    unArreglo.agregar(otroPunto)
```



Entrada Salida de Datos (I)

Dada la siguiente clase:



Para un conjunto de 20 instancias de la clase Persona, mostrar el apellido, nombre y sueldo del empleado de mayor sueldo (suponer único).

a) Los datos de los empleados se ingresaron por pantalla, se almacenaron en una lista como referencias a objetos.

b) Calcular el sueldo promedio , y mostrar los datos de las personas que tienen sueldo mayor o igual al sueldo promedio.

Entrada Salida de Datos (II)

```
class Persona:
    __dni: int
    __nombre: str
    __apellido: str
    __sueldo: float
    def __init__(self, dni, nombre, apellido, sueldo = 35000.0):
        self.__dni = dni
        self.__nombre = nombre
        self.__apellido = apellido
        self.__sueldo = sueldo
    def getSueldo(self):
        return self.__sueldo
    def __str__(self):
        return 'Apellido y nombre: '+self.__apellido+', '+self.__nombre
    def mostrarDatos(self):
        print('Nombre {}, Apellido {}, DNI {}, sueldo {}'.format(self.__nombre, self.__apellido, self.__dni,
self.__sueldo))
```

Referencia: https://docs.python.org/3/reference/datamodel.html

Entrada Salida de Datos (III)

```
#### FUNCIONES PARA CARGA DE DATOS DE PERSONA Y LISTA
def crearPersona():
    print('Ingrese los siguientes datos:')
    dni = int(input('DNI:'))
    nombre = input('Nombre: ')
    apellido = input('Apellido: ')
    sueldo = float(input('Sueldo: '))
    persona = Persona(dni, nombre, apellido, sueldo)
    return persona
def cargarPersonas(lista):
    for i in range(5):
        p = crearPersona()
        print(p)
        lista.append(p)
def calcularMaximoSueldo(lista):
    maximo = 0.0
    indiceMaximo = -1
    for i in range(len(lista)):
        if(lista[i].getSueldo()>maximo):
            maximo = lista[i].getSueldo()
            indiceMaximo = i
    return indiceMaximo

if __name__ == '__main__':
    listaPersonas = []
    cargarPersonas(listaPersonas)
    for i in range(len(listaPersonas)):
        listaPersonas[i].mostrarDatos()
    indice = calcularMaximoSueldo(listaPersonas)
    print('La persona que posee el mayor sueldo es {}, con un sueldo de {}'.format(listaPersonas[indice],
listaPersonas[indice].getSueldo()))
```

Organización de los programas Python

- Los módulos en Python, son una forma de agrupar funciones, métodos, clases y datos, que se relacionan.
- Un módulo es una archivo conteniendo definiciones y declaraciones de Python.
- El nombre del archivo es el nombre del módulo con el sufijo .py agregado.
- Dentro de un módulo, el nombre del mismo (como una cadena) está disponible en el valor de la variable global __name__.
- Python viene con una biblioteca de módulos estándar, descripta en un documento separado, la Referencia de la Biblioteca de Python (de aquí en más, "Referencia de la Biblioteca").
- Algunos módulos se integran en el intérprete; estos proveen acceso a operaciones que no son parte del núcleo o core del lenguaje pero que sin embargo están integrados, tanto por eficiencia como para proveer acceso a primitivas del sistema operativo, como llamadas al sistema.
- El conjunto de tales módulos es una opción de configuración que depende de la plataforma subyacente.

(*3) Chapter 7 – Páginas 111 a 123

Módulos Python

Módulo	Descripción
abc	*Abstract base classes de acuerdo a: pep '3119'*
csv	Lectura y escritura de archivos delimitados por un caracter
datetime	Tipos básicos para fechas y horas
gc	Interface con funciones del Garbage Collector
io	Funciones importantes para el manejo de streams
json	Codificación y decodificación de archivos en formato JSON
os	Miscelánea de interfaces con el Sistema Operativo
random	Generador de números pseudo aletorios con varias distribuciones
sqlite3	Una implementación de la API DB 2.0 usando SQLite 3.x
sys	Acceso a funciones y parámetros específicos del sistema
tkinter	Interface a Tcl/Tk para interfaces gráficas
unittest	Framework para el testeo de unidad en Python

Entre otras…

Referencia: https://docs.python.org/3/py-modindex.html

Utilización de los módulos

Para poder utilizar un módulo son necesarias dos cosas:

1) Debe estar instalado.
2) Incorporarlo al programa, para ello se usa la palabra reservada **import**.

```
import sys

if __name__ == '__main__':
      print(dir(sys))
      print (sys.path)
      print(sys.winver)
      print(sys.version)
      print(sys.version_info)

print(sys.getrefcount(100))
```

Consola Python

```
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
'__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
'__stderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames',
'_debugmallocstats', '_enablelegacywindowsfsencoding', '_framework', '_getframe',
'_git', '_home', '_xoptions', 'api_version', 'argv', 'base_exec_prefix', 'base_prefix',
'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
'copyright', 'displayhook', 'dllhandle', 'dont_write_bytecode', 'exc_info', 'excepthook',
'exec_prefix', 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
'get_coroutine_wrapper', 'getallocatedblocks', 'getcheckinterval', 'getdefaultencoding',
'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile', 'getrecursionlimit',
'getrefcount', 'getsizeof', 'getswitchinterval', 'gettrace', 'getwindowsversion',
'hash_info', 'hexversion', 'implementation', 'int_info', 'intern', 'is_finalizing', 'maxsize',
'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
'platform', 'prefix', 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth',
'set_coroutine_wrapper', 'setcheckinterval', 'setprofile', 'setrecursionlimit',
'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout', 'thread_info', 'version',
'version_info', 'warnoptions', 'winver']
3.7
3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)]
sys.version_info(major=3, minor=7, micro=0, releaselevel='final', serial=0)
8
```

Creación de Módulos

Para crear un módulo, se necesita:

1. Proveer un nombre al módulo
2. Guardar un archivo con es nombre y extensión .py
3. La importación del módulo se hace con la palabra reservada **import**

```
import clasePunto

if __name__ == '__main__':
    print(dir(clasePunto))
    print(dir(clasePunto.Punto))
```

Consola Python

```
['Punto', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__']
['_Punto__x', '_Punto__y', '__class__', '__delattr__',
'__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'getX', 'getY',
'mostrarDatos', 'mover', 'setX', 'setY']
```

Otras formas de importación de Módulos

Cuando se conocen los elementos que hay dentro de un módulo, se pueden importar los que se utilizarán, haciendo uso de la sentencia **from**:

```
from sys import path, winver, version, version_info, getrefcount

if __name__ == '__main__':
     print (path)
     print(winver)
     print(version)
     print(version_info)
     print(getrefcount(100))
```

Con la misma sentencia **from**, puede asignarse un alias a los elementos importados:

```
from clasePunto import Punto as p

if __name__ == '__main__':
    print(dir(p))
    unPunto = p(2,3)
    unPunto.mostrarDatos()
```

Paquetes en Python

En Python, cada uno los archivos .py se denominan **módulos**. Estos módulos, a la vez, pueden formar parte de **paquetes**.
Un paquete, es una carpeta que contiene archivos .py.
Para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado __init__.py.
Este archivo, no necesita contener ninguna instrucción. De hecho, puede estar completamente vacío.

Estructura de un paquete

```
└── paquete
    ├── __init__.py
    ├── modulo1.py
    ├── modulo2.py
    └── modulo3.py
```

Paquete con un subpaquete

```
. └── paquete
    ├── __init__.py
    ├── modulo1.py
    └── subpaquete
        ├── __init__.py
        ├── modulo1.py
        └── modulo2.py
```

```
├── modulo1.py
└── paquete
    ├── __init__.py
    ├── modulo1.py
    └── subpaquete
        ├── __init__.py
        ├── modulo1.py
        └── modulo2.py
```

Los módulos no necesariamente tienen que formar parte de un paquete

Uso

```
import paquete.modulo1
import paquete.subpaquete.modulo1
```

(*5) – Chapter 19 – Páginas 897 a 917

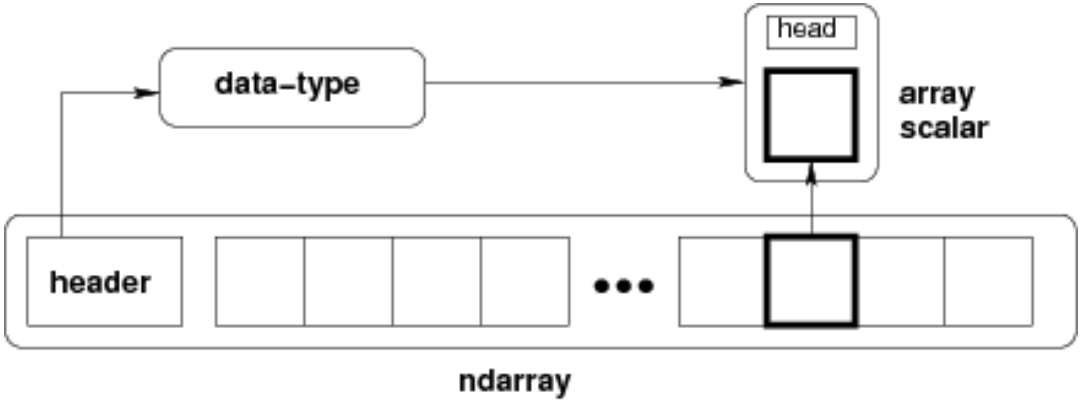
PEP 8: Importación

- La importación de módulos debe realizarse al comienzo del documento, en orden alfabético de paquetes y módulos.
- Primero deben importarse los módulos propios de Python. Luego, los módulos de terceros y finalmente, los módulos propios de la aplicación.
- Entre cada bloque de imports, debe dejarse una línea en blanco.

PEP: Una Propuesta de Mejora de Python(PEP, Python Enhancement Proposal)

Arreglos NumPy (I)

Una **tabla n-dimensional** es un tipo especial de variable capaz de almacenar en su interior y de manera ordenada uno o varios datos de un determinado tipo.
El paquete NumPy, provee una estructura de datos arreglo n-dimensional. NumPy es un paquete externo de Python, se debe agregar a los paquetes y módulos provistos por el core del lenguaje.
La clase más importante definida en NumPy es un arreglo n-dimensional, la clase a la que pertenecen los objetos es **ndarray**, debe quedar claro que los objetos de esta clase, no son arreglos propiamente dicho.
Este objeto arreglo es una **colección de items todos del mismo tipo**, que puede accederse usando un subíndice desde la posición 0 hasta la dimensión menos uno.
Cada item del arreglo ndarray tiene el mismo tamaño en memoria, todos los ítems son del tipo descripto por el objeto tipo de dato (data-type), al objeto tipo de datos que determina el tipo escalar almacenado en el arreglo, se lo denomina dtype.
Se puede acceder a cualquier item del arreglo usando un subíndice, el elemento accedido es un tipo escalar de Python.



Arreglos NumPy (II)

Una de las formas de creación de un arreglo del tipo **ndarray**, es utilizando la función **empty**, para crear un arreglo de referencias nulas. Para crear el arreglo de un tipo de datos específico, el tipo de datos debe establecerse en el parámetro **dtype**.
Ejemplo:

```
import numpy as np
puntos = np.empty(3, dtype=Punto)
print(puntos)
```

Consola Python
[None None None]

Estas sentencias crean un arreglo del tipo **ndarray** de tres referencias vacías. El arreglo tendrá referencias a objetos de la clase Punto (**dtype=Punto**).

Dada la clase Punto, definida anteriormente, generar una clase arreglo, que permita:
1) Almacenar instancias de la clase punto
2) Calcular la distancia euclídea entre el primer punto y el resto de los puntos.
3) Mostrar los datos de la colección de puntos

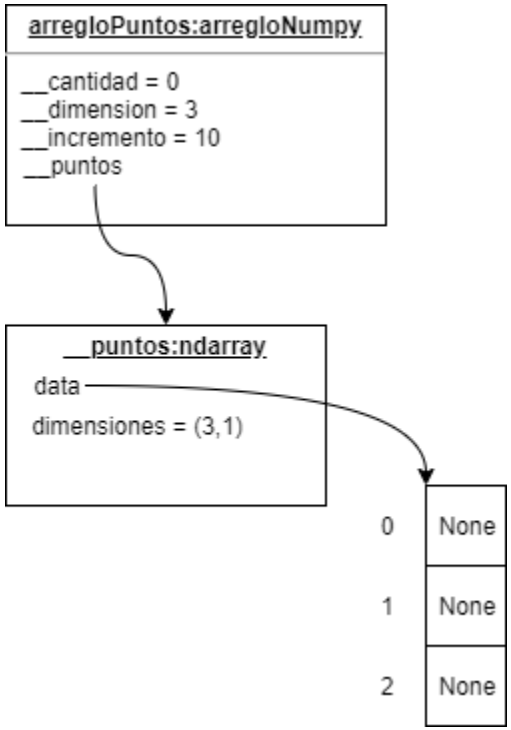
Arreglo de Puntos (I)

```
import numpy as np
from clasePunto import Punto
class arregloNumPy:
    __cantidad: int
    __dimension: int
    __incremento = 5
    __puntos: n.ndarray
    def __init__(self, dimension, incremento=5):
        self.__puntos = np.empty(dimension, dtype=Punto)
        self.__cantidad = 0
        self.__dimension = dimension
    def agregarPunto(self, unPunto):
        if self.__cantidad==self.__dimension:
            self.__dimension+=self.__incremento
            self.__puntos.resize(self.__dimension)
        self.__puntos[self.__cantidad]=unPunto
        self.__cantidad += 1
    def getUnPunto(self, indice):
        return self.__puntos[indice]
    def mostrarPuntos(self):
        for i in range(self.__cantidad):
            self.__puntos[i].mostrarDatos()
    def calcularDistanciasP0(self, unPunto):
        for i in range(self.__cantidad):
            distancia = unPunto.distanciaEuclidea(self.__puntos[i])
            print('Distancia del punto {} al punto {} es {}'.format(unPunto, self.__puntos[i], distancia))
    def testArregloPuntos(self):
        p1 = Punto(4, 5)
        p2 = Punto(3, 4)
        p3 = Punto(-9, 5)
        p4 = Punto(3, 2)
        p5 = Punto(1, 7)
        self.agregarPunto(p1)
        self.agregarPunto(p2)
        self.agregarPunto(p3)
        self.agregarPunto(p4)
        self.agregarPunto(p5)
```

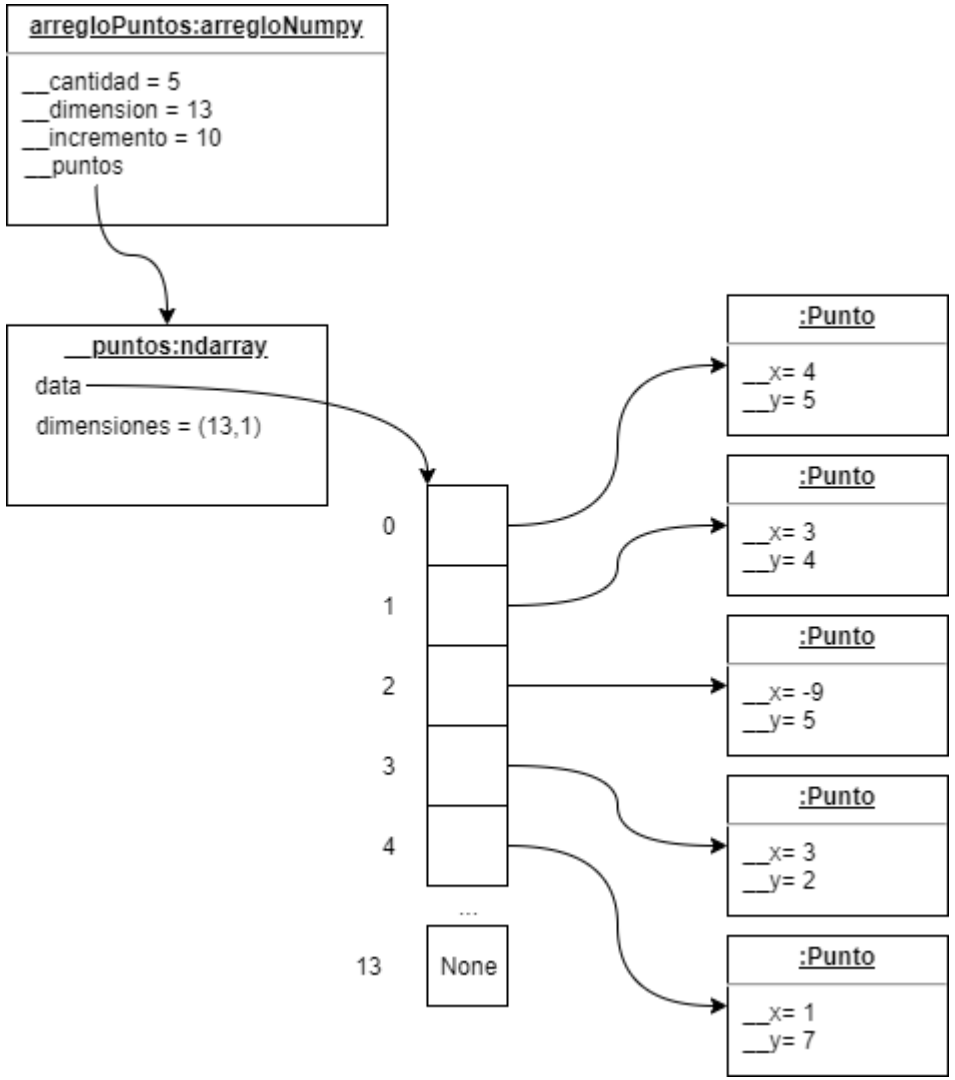
```
if __name__ == '__main__':
    arregloPuntos = arregloNumPy(3,10)
    arregloPuntos.testArregloPuntos()
    punto0 = arregloPuntos.getUnPunto(0)
    arregloPuntos.calcularDistanciasP0(punto0)
    arregloPuntos.mostrarPuntos()
```

Diagrama de Objetos – Arreglo de Puntos

Objetos creados al ejecutar el constructor



Objetos luego de ejecutar el programa



Tarea de investigación I

Analice el siguiente código:

```
class Producto:
    __descripcion: str
    __cantidad: int
    __precioUnitario: float
    def __init__(self, descripcion, cantidad, precioUnitario):
        self.__descripcion=descripcion
        self.__cantidad=cantidad
        self.__precioUnitario=precioUnitario
    def __add__(self, incremento):
        self.__precioUnitario=self.__precioUnitario*(1+incremento/100)
    def __str__(self):
        return f"descripción: {self.__descripcion}, precio: {self.__precioUnitario}"
```

```
from claseProducto import Producto
import numpy as np
class ManejadorArreglo:
    __datos: None
    __cantidad: int
    def __init__(self, cantidad):
        self.__datos=np.empty(cantidad, dtype=Producto)
        self.__cantidad=cantidad
    def test(self):
        unProducto=Producto("Azúcar", 400, 867.50)
        self.__datos[0]=unProducto
        otroProducto=Producto("Harina", 100, 1250)
        self.__datos[1]=otroProducto
        ultimoProducto=Producto("Arroz", 120, 2130)
        self.__datos[2]=ultimoProducto
    def incremento(self,indiceInflacion):
        self.__datos+indiceInflacion
    def mostrarElementos(self):
        for indice in range(self.__cantidad):
            print(self.__datos[indice])
```

```
from claseProducto import Producto
from arregloPoductos import ManejadorArreglo
def test():
    unManejador=ManejadorArreglo(3)
    unManejador.test()
    unManejador.incremento(2.4)
    unManejador.mostrarElementos()
if __name__=='__main__':
    test()
```

Tarea de investigación II

Teniendo en cuenta el ejemplo anterior, la presente investigación debe hacer una exhaustiva comparación entre las listas python y el arreglo numpy, debe incluir una tabla comparativa con ventajas y desventajas del uso de uno respecto del otro. **Los ejemplos que aparezcan en el texto, deberán aplicarse sobre objetos que pertenezcan a clases definidas por el programador**, **NO SE PERMITEN** ejemplos que manipulen enteros, flotantes, strings, etc.

Se deberá utilizar: libros (referenciados correctamente), buscadores académicos y el sitio oficial de Python.

Fecha límite para la presentación: Lunes 28/04/2025 a las 0:59h.

Pautas para el trabajo

Trabajo grupal (no más de dos personas)

El trabajo se escribe en tercera persona, no se admiten frases como: hicimos, tomamos, probamos, etc., se reemplazan por frases como: se hizo, se tomaron, se probaron, se hicieron pruebas con... etc.

Tipografía a usar: Arial 10pts, los títulos deben ir en negrita y el código en cursiva.

Se generará un informe en formato de texto (no más de cinco hojas y no menos de tres), con los métodos más destacados de las Listas y los arreglos Numpy de Python, debiendo contar con los siguientes elementos **(UTILICE LA PLANTILLA PROPUESTA POR LA CÁTEDRA, QUE ESTÁ EN EL CAMPUS):**

Portada
Introducción
Palabras clave
Desarrollo
Conclusiones
Bibliografía

El apartado Desarrollo debe incluir teoría y ejemplos con el código debidamente chequeado y comentado.

Las referencias bibliográficas deben estar en una sola norma, puede elegir entre normas APA o IEEE. Deberá contener al menos dos referencias bibliográficas.

Procesamiento de archivos en formato libre (I)

Archivos CSV

Los archivos **CSV** (del inglés *comma-separated values*) son un tipo de documento en formato libre, sencillo para representar datos en forma de tabla, en las que las columnas se separan por comas o punto y coma, o cualquier otro delimitador, ya que la coma es el separador decimal: 1,0,0,0, y las filas por saltos de línea.

Ejemplo: archivo empleados (archivo de texto)

Nombre, sueldo ← Primera fila, cabecera del archivo

Luis, 3000

Carlos, 4577

Sofía, 3445

Ernesto, 7899

Carla, 6777

Desde la segunda fila, nombre del empleado, separador (en este caso la coma) y luego el importe del sueldo.

(* 4) Chapter 11 – Páginas 256 , 257

Procesamiento de archivos en formato libre (II)

Para trabajar con archivos almacenados en disco, se deben utilizar las clases especializadas que Python posee para dichas operaciones.

Apertura del flujo de datos

```
archivo = open('librosPOO.csv')
reader = csv.reader(archivo,delimiter=';')
```

Lectura de líneas

```
for fila in reader:
        ...
```

Cierre del flujo de datos

```
archivo.close()
```

Procesamiento de archivos en formato libre (III)

Cada línea leída desde el flujo de datos, es una lista, por lo que las columnas del archivo CSV, se acceden con un índice desde 0.

Por ejemplo, se tiene la siguiente lista de items comprados (descripción, cantidad, precio unitario), almacenados en un archivo separado por «;» denominado compras.csv:

```
Azúcar;5;44,57
Aceite;2;149,91
Leche Larga Vida;10;45,65
Arroz;3;67,99
```

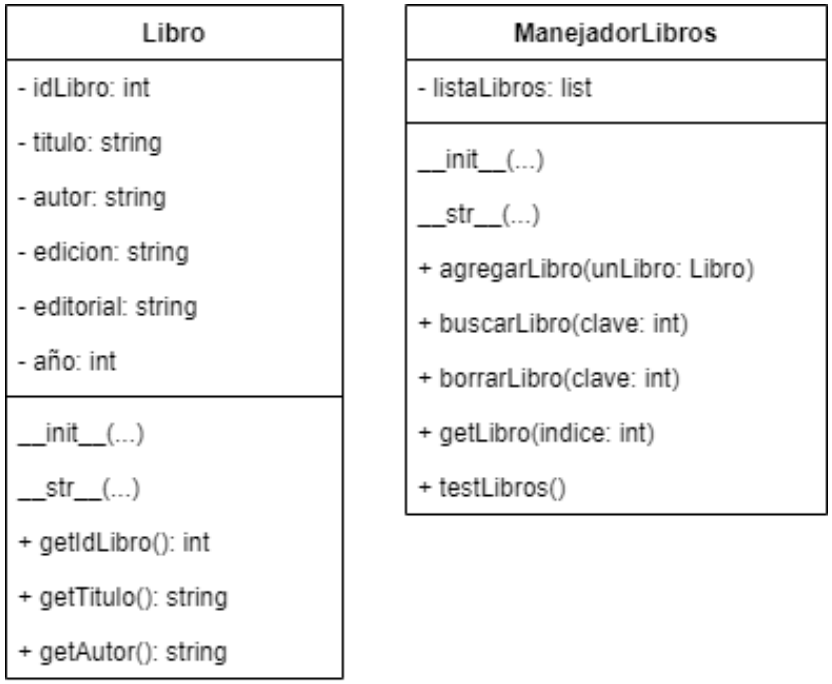
```
import csv
total = 0
lineaCompleta = []
archivo = open('compras.csv')
reader = csv.reader(archivo,delimiter=';')
for fila in reader:
    '''
    fila[0] es la descripción, fila[1] es la cantidad, fila[2] es el precio unitario
    el precio unitario viene con coma decimal, Python trabaja con punto decimal,
    por eso se reemplaza
    '''
    costo = float(fila[1]) * float(fila[2].replace(",","."))
    total += costo
    lineaCompleta = fila + [round(costo,2)]
    print(lineaCompleta)
print('Total de compra: {0:.2f}'.format(total))
archivo.close()
```

Consola Python

```
['Azúcar', '5', '44,57', 222.85]
['Aceite', '2', '149,91', 299.82]
['Leche Larga Vida', '10', '45,65', 456.5]
['Arroz', '3', '67,99', 203.97]
Total de compra: 1183.14
```

Procesamiento de archivos en formato libre (IV)

Dadas las clases Libro y Manejador de Libros, cuyo diseño corresponde a las clases especificadas en UML que se muestran a continuación:



Los métodos __str__ de ambas clases, devuelven una representación string del estado de los objetos.
El método buscarLibro(clave: int), recibe un entero con el idLibro a buscar, devuelve el índice si lo encuentra o None en caso contrario.
El método borrarLibro(clave: int), recibe un entero con el idLibro a eliminar, utiliza el método buscarLibro(clave) para localizar el índice del libro a borrar, y hace uso del método pop de las listas Python para eliminar de la colección el libro.
El método testLibros(), lee los datos de los libros desde el archivo CSV, separado por «;», denominado librosPOO.csv
La biblioteca le solicita a usted que resuelva la siguiente situación problemática:
Leer los datos de los libros, agregándolos a la colección.
Mostrar los datos de los libros que hay en la colección
Leer desde teclado un idLibro, para buscarlo y mostrar título y autor (si lo encuentra).
Leer desde teclado un idLibro y eliminarlo de la colección.

Archivo librosPOO.csv

```
idLibro;Titulo;Autor;Edicion;Editorial;Anio
1;Mastering Oriented-Object  Python;Steven F. Lott ;Second Edition;Packt Publishing;2019
2;Learning Python ;Mark Lutz;Fifth Edition;O'Reilly Media;2013
3;Python 3 Object-oriented Programming;Dusty Phillips;Second Edition;Packt Publishing;2015
4;Python for absolute Beginners;Tim Hall;First Edition;Apress;2009
5;Programming in Python 3;Mark Summerfield;Second Edition;Addison Wesley;2009
6;Beginning Python ;James Payne;First Edition;Wiley Publishing;2010
```

Una posible solución

```
class Libro:
    … #acá van declarados los atributos que son privados
    def __init__(self, idLibro, titulo, autor, edicion,  editorial, anio):
        self.__idLibro = idLibro
        self.__titulo = titulo
        self.__autor = autor
        self.__edicion = edicion
        self.__editorial = editorial
        self.__anio = anio
    def __str__(self):
        return "%s %s %s %s %s %s" % (self.__idLibro, self.__titulo, self.__autor, self.__edicion, self.__editorial, self.__anio)
    def getIdLibro(self):
        return self.__idLibro
    def getTitulo(self):
        return self.__titulo
    def getAutor(self):
        return self.__autor
    def getEdicion(self):
        return self.__edicion
    def getEditorial(self):
        return self.__editorial
```

Clase ManejadorLibros

```
class ManejadorLibros:
    def __init__(self):
        self.__listaLibros = []
    def agregarLibro(self,unLibro):
        self.__listaLibros.append(unLibro)
    def buscarLibro(self, clave):
        indice=0
        valorDeRetorno = None
        bandera=False
        while not bandera and indice < len(self.__listaLibros):
            if self.__listaLibros[indice].getIdLibro()==clave:
                bandera=True
                valorDeRetorno=indice
            else:
                indice+=1
        return valorDeRetorno
    def borrarLibro(self, clave):
        indice = self.buscarLibro(clave)
        if indice != None:
            self.__listaLibros.pop(indice)
            return indice
    def __str__(self):
        s = ""
        for libro in self.__listaLibros:
            s += str(libro) + '\n'
        return s
```

```
def testLibros(self):
        archivo = open('librosPOO.csv')
        reader = csv.reader(archivo,delimiter=';')
        bandera = True
        for fila in reader:
            if bandera:
                '''saltear cabecera '''
                bandera = not bandera
            else:
              idLibro = int(fila[0])
              titulo = fila[1]
              autor = fila[2]
              edicion = fila[3]
              editorial = fila[4]
              anio = int(fila[5])
              unLibro = Libro(idLibro,titulo,autor,edicion, editorial,anio)
              self.agregarLibro(unLibro)
        archivo.close()
```

Programa principal

```
if __name__=='__main__':
    ml = ManejadorLibros()
    ml.testLibros()
    print('Colección de libros inicial')
    print(ml)
    idLibro = int(input('Ingrese idLibro a Buscar'))
    indice=ml.buscarLibro(idLibro)
    if indice == None:
        print('El idLibro {} no corresponde a un libro de la colección'.format(idLibro))
    else:
        libro = ml.getLibro(indice)
        print ('idLibro: {}, título: {}, autor: {}'.format(idLibro, libro.getTitulo(),libro.getAutor()))
    idLibro = int(input('Ingrese idLibro a Borrar'))
    indice = ml.borrarLibro(idLibro)
    if indice != None:
        libro = ml.getLibro(indice)
        print('El libro {} se borró de la colección'.format(libro.getTitulo()))
    else:
        print('El idLibro {}, no corresponde a un libro de la colección'.format(idLibro))
    print('Colección de libros final')
    print (ml)
```

Salida generada

Se corre con la con los siguientes datos:
Buscar idLibro = 1
Borrar idLibro = 4

Consola Python

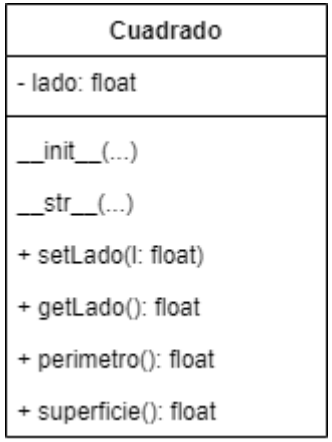
```
Colección de libros inicial
1 Mastering Oriented-Object  Python Steven F. Lott  Second Edition Packt Publishing 2019
2 Learning Python  Mark Lutz Fifth Edition O'Reilly Media 2013
3 Python 3 Object-oriented Programming Dusty Phillips Second Edition Packt Publishing 2015
4 Python for absolute Beginners Tim Hall First Edition Apress 2009
5 Programming in Python 3 Mark Summerfield Second Edition Addison Wesley 2009
6 Beginning Python  James Payne First Edition Wiley Publishing 2010


Ingrese idLibro a Buscar: 1
idLibro: 1, título: Mastering Oriented-Object  Python, autor: Steven F. Lott

Ingrese idLibro a Borrar: 4
El libro Programming in Python 3 se borró de la colección
Colección de libros final
1 Mastering Oriented-Object  Python Steven F. Lott  Second Edition Packt Publishing 2019
2 Learning Python  Mark Lutz Fifth Edition O'Reilly Media 2013
3 Python 3 Object-oriented Programming Dusty Phillips Second Edition Packt Publishing 2015
5 Programming in Python 3 Mark Summerfield Second Edition Addison Wesley 2009
6 Beginning Python  James Payne First Edition Wiley Publishing 2010
```

Actividad Conjunta

Dada la clase Cuadrado, especificada por el diagrama de UML, que se muestra a continuación:



La empresa el «El Graficador SRL» le pide a usted que es su pogramador un programa que permita operar con al menos 20 objetos de la clase cuadrado. El programa deberá permitir llevar a cabo las siguientes acciones:

1. Mostrar el perímetro promedio de todos los cuadrados.
2. Indicar el total de cuadrados que supera el perímetro promedio.
3. Obtener un listado con el perímetro y superficie de cada uno de los cuadrados, que deberá ajustarse a la siguiente forma:

Cuadrado	lado	perímetro	superficie
1	4	16.00	16.00
2	7	30.00	56.25

Una solución

```
import numpy as np
class Cuadrado:
    __lado: int
    def __init__(self, lado):
        self.__lado = lado
    def __str__(self):
        return  "%4d    %6.2f    %8.2f" % (self.__lado, self.perimetro(), self.superficie())
    def setLado(self, lado):
        self.__lado = lado
    def getLado(self):
        return self.__lado
    def perimetro(self):
        return self.__lado * 4
    def superficie(self):
        return self.__lado**2
```

```
class ManejadorCuadrados:
    __cantidad = 0
    __dimension = 0
    __incremento = 5
    def __init__(self, dimension, incremento=5):
        self.__cuadrados = np.empty(dimension, dtype=Cuadrado)
        self.__cantidad = 0
        self.__dimension = dimension
    def __str__(self):
        s = "Cuadrado lado perímetro superficie\n"
        for i in range(self.__cantidad):
            unCuadrado = self.__cuadrados[i]
            s += '{0:8}'.format(i+1)+str(unCuadrado) + '\n'
        return s
    def agregarCuadrado(self, unCuadrado):
        if self.__cantidad==self.__dimension:
            self.__dimension+=self.__incremento
            self.__cuadrados.resize(self.__dimension)
        self.__cuadrados[self.__cantidad]=unCuadrado
        self.__cantidad += 1
    def getCuadrado(self, indice):
        return self.__cuadrados[indice]
)
```

Clase ManejadorCuadrados – Cont.

```
def perimetroPromedio(self):
        promedio = 0.0
        for i in range(self.__cantidad):
            promedio += self.getCuadrado(i).perimetro()
        return promedio/self.__cantidad
    def cantidadCuadrados(self):
        cantidad = 0
        promedio = self.perimetroPromedio()
        for i in range(self.__cantidad):
            if self.getCuadrado(i).perimetro()>promedio:
                cantidad+=1
        return cantidad
    def testManejadorCuadrados(self):
        unCuadrado = Cuadrado(4)
        self.agregarCuadrado(unCuadrado)
        otroCuadrado = Cuadrado(7.5)
        self.agregarCuadrado(otroCuadrado)
        unCuadrado = Cuadrado(9.97)
        self.agregarCuadrado(unCuadrado
```

```
if __name__ == '__main__':
    mCuadrados = ManejadorCuadrados(3,5)
    mCuadrados.testManejadorCuadrados()
    print(mCuadrados)
    print('Cantidad de Cuadrados con perímetro > que perímetro
promedio: {}'.format(mCuadrados.cantidadCuadrados()))
```

Consola Python

```
Cuadrado lado perímetro superficie
      1     4    16.00     16.00
      2     7    30.00     56.25
      3     9    39.88     99.40

Cantidad de Cuadrados con perímetro > que perímetro promedio: 2
```

Referencia self (I)

- En Python todo método de una clase (que no sea método de clase), recibe un primer parámetro formal que es una referencia al objeto que recibe el mensaje, luego si existen irán el resto de los parámetros formales.
- Por convención, los programadores Python, decidieron llamar a dicho parámetro como ***self***.
- Python hace obligatorio el uso de la referencia implícita en el interior de los métodos de la clase para acceder a los atributos y otros métodos.

Dada la clase Punto vista anteriormente:

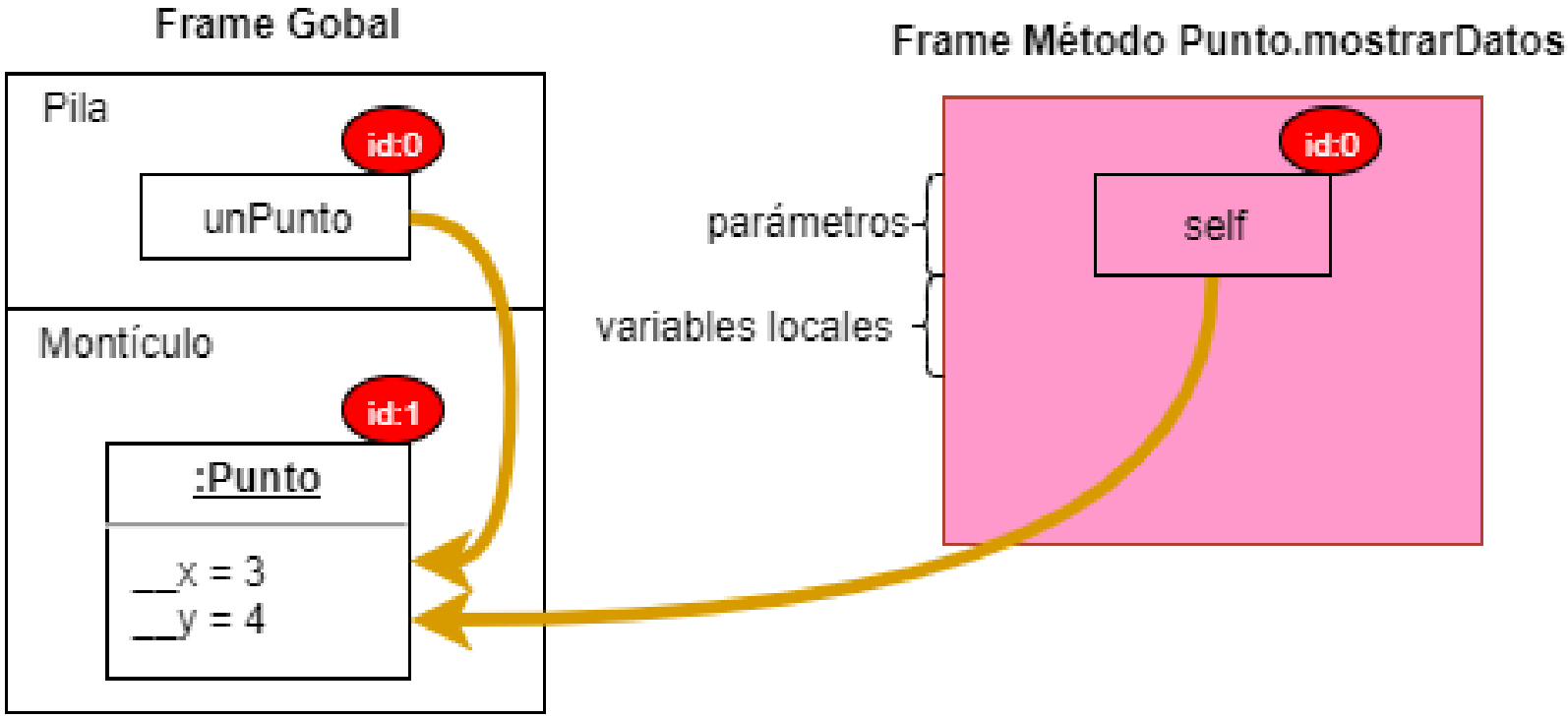
```
class Punto:
    __x = -1
    __y = 0
    def __init__(self, x = 0, y = 0):
        self.__x = x
        self.__y = y
    def __str__(self):
        return '({}, {})'.format(self.__x, self.__y)
    def mostrarDatos(self):
        print("(x,y) = (",self.__x,',', self.__y,")")
        print('Dirección de memoria de self en mostrarDatos:{}'.format(hex(id(self))))
```

```
if __name__=='__main__':
    unPunto = Punto(3,4)
    print('Direccion de memoria de instancia unPunto: {}'.format(hex(id(unPunto))))
    unPunto.mostrarDatos()
```

Consola Python

Dirección de memoria de instancia unPunto: **0x2119c077988**
(x,y) = (3 , 4)
Dirección de memoria de self en mostrarDatos: **0x2119c077988**
(x,y) = (3 , 4)

Referencia self (II)



Funciones y Métodos con parámetros por defecto

- En Python no existe la sobrecarga de funciones y métodos como se la conoce en otros lenguajes como C# y Java.

- Existen los parámetros por defecto, que hacen que a una función se la pueda invocar de distintas maneras.

- El intérprete es el encargado de identificar qué parámetros formales toman valor por defecto y cuales no, en función de la cantidad de parámetros actuales en la invocación de la función o método.

```
def suma(x=1, y=3, z=11):
    return x + y + z

if __name__=='__main__':
    print(suma())
    print(suma(15))
    print(suma(12,22))
    print(suma(11,12,13))
```

La invocación **suma()**, toma todos los parámetros por defecto, los parámetros formales toman los valores x=1, y=3 y z= 11, resultado: 15

La invocación **suma(15)**, hace que x tenga el valor 15, los parámetros formales restantes tomarán los valores por defecto, esto es y=3, z=11, resultado: 29

La invocación **suma(12, 22)**, hace que x tenga el valor 12, y tenga el valor 22, y z tenga el valor por defecto, esto es z=11, el resultado será: 45

La invocación **suma(11, 12, 13)**, no utiliza los valores por defecto, x toma el valor 11, y toma el valor 12, y z toma el valor 13.

Métodos con valores por defecto

```
class Fecha:
    __dia: int
    __mes: int
    __anio: int
    def __init__(self, d=1,m=1,a=1900):
        self.__dia=d
        self.__mes=m
        self.__anio=a
    def formato(self, tipo='es'):
        '''por defecto genera  la fecha en formato español, dd/mm/aaaa,
        si se pasa el parámetro tipo con el valor 'en', lo hará en formato
        inglés, aaaa/mm/dd'''
        formato = None
        if tipo=='en':
            formato =  '{}/{}/{}'.format(self.__anio, self.__mes, self.__dia)
        else:
            formato = '{}/{}/{}'.format(self.__dia, self.__mes, self.__anio)
        return formato
```

```
if __name__=='__main__':
    fecha = Fecha(21,1,2020)
    print('Fecha en formato español {}'.format(fecha.formato()))
    print('Fecha en formato inglés {}'.format(fecha.formato('en')))
```

Consola Python

```
Fecha en formato español 21/1/2020
Fecha en formato inglés 2020/1/21
```

Valores por defecto en constructores

✓ Como ya se pudo observar en los ejemplos anteriores, los constructores pueden generarse con valores por defecto.

✓ Ésta forma de trabajo permite asegurarse que los objetos de una clase, tengan una inicialización en todos sus atributos, aunque el programador no pase los valores iniciales al instanciar un objeto.

```
class Fecha:
    __dia: int
    __mes: int
    __anio: int
    def __init__(self, d=1,m=1,a=1900):
        self.__dia=d
        self.__mes=m
        self.__anio=a
    def formato(self, tipo='es'):
        '''por defecto genera  la fecha en formato español, dd/mm/aaaa,
        si se pasa el parámetro tipo con el valor 'en', lo hará en formato
        inglés, aaaa/mm/dd'''
        formato = None
        if tipo=='en':
            formato =  '{}/{}/{}'.format(self.__anio, self.__mes, self.__dia)
        else:
            formato = '{}/{}/{}'.format(self.__dia, self.__mes, self.__anio)
        return formato
```

```
if __name__=='__main__':
    fecha = Fecha()
    print('Fecha en formato español {}'.format(fecha.formato()))
    print('Fecha en formato inglés {}'.format(fecha.formato('en')))
```

Consola Python

```
Fecha en formato español 1/1/1900
Fecha en formato inglés 1900/1/1
```

Datos miembro estáticos y Funciones miembro estáticas (I)

✓ Cada objeto de una clase tiene su propia copia de todos los datos miembros de la clase a la que pertenece (datos miembro no estáticos).

✓ Un dato miembro de una clase se puede declarar estático, se declara sin decoradores y con el valor inicial que tendrá para todos los miembros de la clase. El programador es responsable de accederlo exclusivamente con funciones miembro estáticas, o a través del nombre de la clase.

✓ A un miembro estático se le asigna una zona fija de almacenamiento, al igual que una variable global, pero de clase.

✓ Un dato miembro estático es, por lo tanto, una **variable de clase** según el Paradigma de la Orientación a Objetos. Una variable de clase representa información "propia de la clase" (es decir, un atributo de la clase, no un atributo de un objeto específico).

✓ Un dato miembro estático es compartido por todas las instancias (objetos) de una clase y existe, incluso cuando ningún objeto de esta clase existe.

Datos miembro estáticos y Funciones miembro estáticas (II)

✓ Las **funciones miembros estáticas** sólo pueden acceder a otras **funciones y datos miembros estáticos.**

✓ Las funciones miembros estáticas representan los **métodos de clase** propios de los lenguajes puros orientados a objetos, por cuanto los métodos de clase manipulan solamente las **variables de clase** .

✓ Los métodos estáticos o métodos de clases se pueden ejecutar, simplemente haciendo referencia a la clase que lo contienen, no es necesario instanciar la clase para poder hacer uso del dato miembro o de la función miembro de clase.

✓ En Python, para declarar una función como método de clase, se utiliza el decorador @classmethod.

✓ Un método de clase recibe **la clase (*cls*) como primer argumento implícito**, al igual que un método de instancia recibe la instancia a través de ***self***.

Ejemplo de variables de clase y métodos de clase

Spaniel tibetano

Descripción

El Spaniel del Tibet es una raza de perro asertiva, pequeña e inteligente originaria de los montes Himalaya en el Tibet. Comparten ancestros con el Pequinés, el Spaniel japonés, Shih Tzu, Lhasa Apso y Pug.

Nombre científico: Canis lupus familiaris
Esperanza de vida: de 12 a 15 años
Masa Corporal: 4.1 – 6.8 kg (Adulto)
Temperamento: Voluntarioso, Reservado, Inteligente, Asertivo, Independiente, Juguetón, Alegre
Colores: Negro, Blanco, Negro y canela, Crema, Sable, Dorado, Rojo
Familia: Companion, herding
Área de Origen: Tibet
Tasa de aprendizaje: 9
Obediencia: 3
Resolución de problemas: 8

Se lo contrata a usted como programador de un entrenador de perros, deberá identificar atributos de clase y atributos de cada uno de los objetos, y crear la clase que modele a los perros de raza Spaniel tibetano Programar y testear una clase para representar a los perros de raza Spaniel Tibetano.

Una Solución

```
class TibetanSpaniel:
    # variables de clase
    '''todos los perros de la raza Spaniel Tibetano
    tienen las mismas características
    '''
    nombreCientifico = "Canis lupus familiaris"
    familia = "Companion, herding"
    areaDeOrigen = "Tibet"
    tasaDeAprendizaje = 9
    obediencia = 3
    resolucionDeProblemas = 8
    def __init__(self, nombre, jugueteFavorito, habilidad):
        self.__nombre = nombre
        self.__habilidad = habilidad
        self.__jugueteFavorito = jugueteFavorito
    def __str__(self):
        return 'Nombre %s, habilidad %d, juguete favorito %s,
obediencia %d' % (self.__nombre, self.__habilidad,
self.__jugueteFavorito, self.getObediencia())
    #Métodos de instancia
    def getNombre(self):
        return self.__nombre
    def getHabilidad(self):
        return self.__habilidad
    def getJugueteFavorito(self):
        return self.__jugueteFavorito
```

```
    #Métodos de clase
    @classmethod
    def getNombreCientifico(cls):
        return cls.nombreCientifico
    @classmethod
    def getFamilia(cls):
        return cls.familia
    @classmethod
    def getAreaDeOrigen(cls):
        return cls.areaDeOrigen
    @classmethod
    def getTasaDeAprendizaje(cls):
        return cls.tasaDeAprendizaje
    @classmethod
    def getObediencia(cls):
        return cls.obediencia
    @classmethod
    def getResolucionProblemas(cls):
        return cls.resolucionDeProblemas
    @classmethod
    def verRaza(cls):
        print('Características de la Raza')
        print('Nombre Científico: '+cls.getNombreCientifico())
        print('Familia: '+cls.getFamilia())
        print('Área de Origen: '+cls.getAreaDeOrigen())
        print('Tasa de
Aprendizaje:'+str(cls.getTasaDeAprendizaje()))
        print('Obediencia: '+str(cls.getObediencia()))
        print('Tasa de Resolución de problemas:
'+str(cls.getResolucionProblemas()))
```

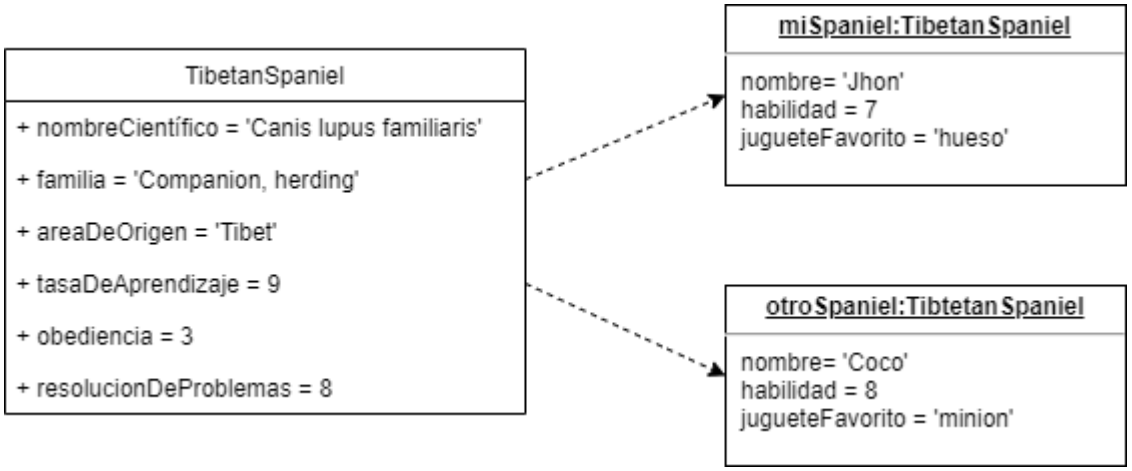
Programa Principal

```
if __name__ == '__main__':
    TibetanSpaniel.verRaza()
    miSpaniel = TibetanSpaniel('Jhon', 'hueso', 7)
    otroSpaniel = TibetanSpaniel('Coco','minion',8)
    print(miSpaniel)
    print('Cambio de una variable de clase')
    TibetanSpaniel.obediencia = 9
    print(miSpaniel)
    print(otroSpaniel)
```

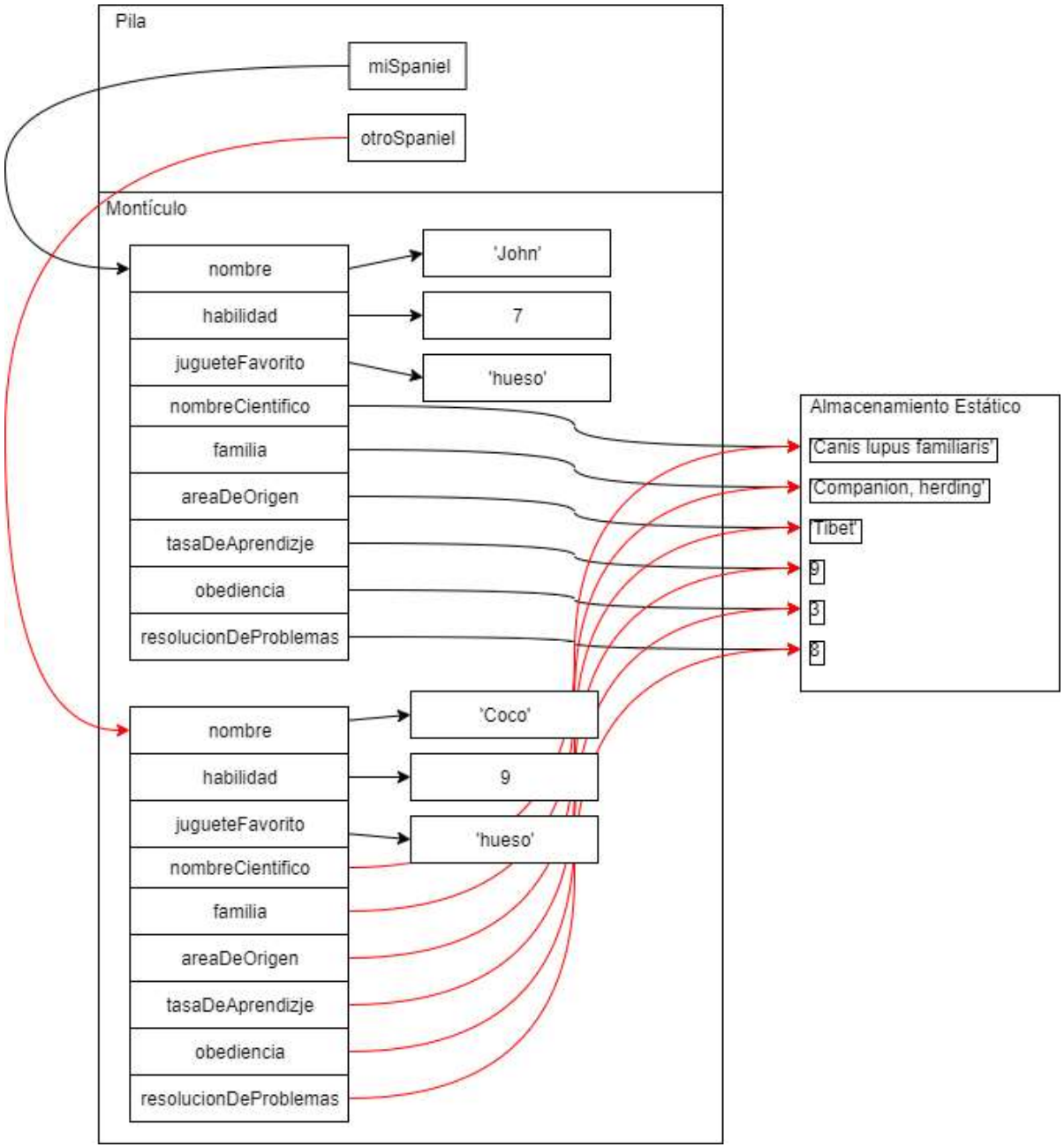
Consola Python

```
Características de la Raza
Nombre Científico: Canis lupus familiaris
Familia: Companion, herding
Área de Origen: Tibet
Tasa de Aprendizaje:9
Obediencia: 3
Tasa de Resolución de problemas: 8
Nombre Jhon, habilidad 7, juguete favorito hueso, obediencia 3
Cambio de una variable de clase
Nombre Jhon, habilidad 7, juguete favorito hueso, obediencia 9
Nombre Coco, habilidad 8, juguete favorito minion, obediencia 9
```

Diagrama de Objetos



Variables de clase y memoria



Ejemplo a desarrollar en clase

El administrador del Consorcio Asunción solicita a una empresa de desarrollo de software una aplicación que almacene los datos de los propietarios e inquilinos de los departamentos que él administra.
Mensualmente el administrador recibe las boletas de luz, agua, impuesto inmobiliario de cada uno de los 25 departamentos del Consorcio, que carga a cada uno de los departamentos.
Además de estos gastos, hay un gasto de administración y mantenimiento que refleja el costo de administración (que se fija anualmente, para el año 2022 se ha fijado en $ 65000), gastos de mantenimiento de espacios verdes del Consorcio y de mantenimiento de luminarias, éstos dos últimos gastos son variables y se ingresan por teclado. Los gastos de administración y mantenimiento se prorratean entre todos los departamentos del consorcio, por lo que a cada departamento le toca una suma fija igual, para un mes determinado.
El analista de la empresa de software ha determinado que los datos a registrar por cada departamento corresponden a número de departamento, bloque y piso, nombre y apellido del propietario o inquilino, importe de luz, importe de gas, valor de impuesto inmobiliario, valor de la suma fija del gastos de administración (65000/12) y mantenimiento del mes actual, estado (que indicará si los gastos están pagos o no).
Usted es uno de los programadores de la empresa de desarrollo de software, que deberá programar la clase Departamento para registrar los datos, y una colección para administrar los 25 departamentos.
La aplicación deberá permitir:

1) Obtener la fecha actual y para el mes en curso solicitar los gastos de mantenimiento.
2) Procesar los datos de los departamentos que vienen en un archivo denominado gastosMMAAAA.csv (donde MM es el mes actual, AAAA es el año actual).
3) Leer desde teclado el número de departamento, bloque y piso, y mostrar el nombre y apellido del propietario o inquilino, y el total de gastos a pagar, incluye los gastos de administración y mantenimiento, luz, gas e impuesto inmobiliario.
4) Leer desde teclado el nombre de un propietario o inquilino, y registrar su pago de todos los gastos.
5) Obtener número de departamento, nombre y apellido del propietario e inquilino, importe total de gastos, de los departamentos que deben los gastos.
6) Guardar en un archivo separado por comas, los datos actualizados de los departamentos.

Sobrecarga de operadores (I)

La sobrecarga de operadores, significa simplemente, capturar las operaciones básicas incluidas en el lenguaje estándar de Python, como lo son + (suma), - (resta), * (multiplicación), / (división), operadores de comparación > (mayor que), < (menor que), == (igual), etc., en la clases definidas por el programador, de modo que tengan el mismo significado que lo tienen para los tipos predefinidos del lenguaje.

Ideas clave en la sobrecarga de operadores en Python:

✓La sobrecarga de operadores, permite que las clases intercepten el comportamiento de las operaciones normales de Python.

✓La sobrecarga de operadores se implementa al proporcionar métodos especialmente nombrados en una clase.

✓La sobrecarga de operadores, hace que las instancias de las clases definidas por el programador, se comporten como los tipos integrados del lenguaje.

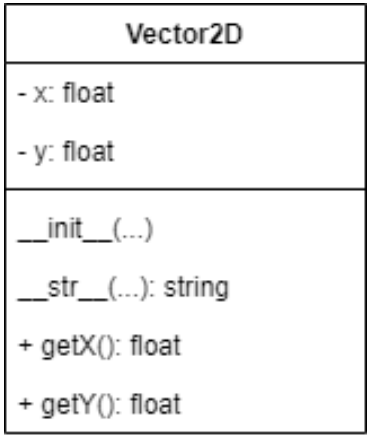
(*1) – Chapter 30 – Páginas 887, 889, 917 a 921
(*2) – Chapter 4 - Páginas 88 a 90

Sobrecarga de operadores (II)

Operadores Sobrecargables (nómina reducida)		
Operador	**Método**	**Expresión**
+ Suma	__add__(self, otro)	a1+a2
- Resta	__sub__(self, otro)	a1-a2
*** Multiplicación**	__mul__(self, otro)	a1*a2
/ División	__div__(self, otro)	a1/a2
% Módulo	__mod__(self, otro)	a1%a2
> Mayor que	__gt__(self, otro)	a1>a2
>= Mayor o igual que	__ge__(self, otro)	a1>=a2
< Menor que	__lt__(self, otro)	a1<a2
<= Menor o igual que	__le__(self, otro)	a1<=a2
== Igual que	__eq__(self, otro)	a1==a2
!= Distinto	__ne__(self, otro)	a1!=a2

Ejemplo Sobrecarga de Operadores (I)

Usted ha sido contratado como programador por una empresa que desarrolla software matemático. En el módulo de álgebra lineal, se necesita generar una clase que represente un vector en el plano bidimensional.
El analista funcional de la empresa ha capturado los requisitos del programa a desarrollar. Para el caso de clase Vector2D, ha generado el siguiente diagrama UML.



Se necesita crear un programa que implemente la clase Vector2D, y que provea el comportamiento a los siguientes operadores:

- a+b (siendo a y b dos instancias de la clase Vector2D
- a-b (siendo a y b instancias de la clase Vector2D
- a * 3.5 (siendo a una instancia de la clase Vector2D, y 3.5 un escalar

La implementación deberá mostrar un vector de la forma: <x, y>=<2, 4.5>

Una solución

```
class Vector2D:
    __x: float
    __y: float
    def __init__(self, x=-1, y=-1):
        self.__x=x
        self.__y=y
    def __str__(self):
        return '<x, y>=<{},{}>'.format(self.__x, self.__y)
    def getX(self):
        return self.__x
    def getY(self):
        return self.__y
    def __add__(self, otroVector):
        return Vector2D(self.__x+otroVector.getX(), self.__y+otroVector.getY())
    def __sub__(self, otroVector):
        return Vector2D(self.__x-otroVector.getX(), self.__y-otroVector.getY())
    def __mul__(self, escalar):
        return Vector2D(self.__x*escalar, self.__y*escalar)
```

Qué pasa si intento ejecutar la sentencia:

productoEscalar = 1.5 * a

TypeError: unsupported operand type(s) for *: 'float' and 'Vector2D'

```
if __name__=='__main__':
    a = Vector2D(4, 5)
    b = Vector2D(2, 7)
    suma = a + b
    resta = a - b
    productoEscalar = a * 1.5
    print(a)
    print(b)
    print('Suma a+b:',suma)
    print('Resta a-b',resta)
    print('Producto por un escalar a * 3',productoEscalar)
```

Consola Python

```
<x, y>=<4,5>
<x, y>=<2,7>
Suma a+b: <x, y>=<6,12>
Resta a-b <x, y>=<2,-2>
Producto por un escalar a * 1.5 <x, y>=<6.0,7.5>
```

Sobrecarga por derecha

Tal y como está definida la clase Vector2D, hace que la sentencia: **productoEscalar = 1.5 * a**

Produce el siguiente error:
TypeError: unsupported operand type(s) for *: 'float' and 'Vector2D'

Lo que significa que la operación producto escalar definida, no es conmutativa.
Python, ofrece la sobrecarga por derecha (o reversa), para los operadores como + (suma), - (resta), * (multiplicación), / (división), % (módulo), etc., a los que denomina __radd__, __rsub__, __rmul__, __rdiv__, __rmod__, respectivamente.

Estos operadores permiten que las operaciones que son conmutativas, lo sigan siendo, y que las operaciones que involucren valores de distintas clases se puedan llevar a cabo.

En el ejemplo debería sobrecargarse el operador __rmul__ en la clase Vector2D, de ese modo, Python, ejecutará el operador sobrecargado __rmul__ sólo cuando el operador de la derecha pertenezca a la clase Vector2D, y el primero no pertenezca a dicha clase

Nueva solución

```
class Vector2D:
    __x: float
    __y: float
    def __init__(self, x=-1, y=-1):
        self.__x=x
        self.__y=y
    def __str__(self):
        return '<x, y>=<{},{}>'.format(self.__x, self.__y)
    def getX(self):
        return self.__x
    def getY(self):
        return self.__y
    def __add__(self, otroVector):
        return Vector2D(self.__x+otroVector.getX(), self.__y+otroVector.getY())
    def __sub__(self, otroVector):
        return Vector2D(self.__x-otroVector.getX(), self.__y-otroVector.getY())
    def __mul__(self, escalar):
        return Vector2D(self.__x*escalar, self.__y*escalar)
    def __rmul__(self, escalar):
        return Vector2D(self.__x*escalar, self.__y*escalar)
```

```
if __name__=='__main__':
    a = Vector2D(4, 5)
    b = Vector2D(2, 7)
    suma = a + b
    resta = a - b
    productoEscalar =  1.5 * a
    print(a)
    print(b)
    print('Suma a+b:',suma)
    print('Resta a-b',resta)
    print('Producto por un escalar  1.5 * a',productoEscalar)
    pe = b * 3
    print('Producto escalar de a * 3', pe)
```

Consola Python

```
<x, y>=<4,5>
<x, y>=<2,7>
Suma a+b: <x, y>=<6,12>
Resta a-b <x, y>=<2,-2>
Producto por un escalar  1.5 * a <x, y>=<6.0,7.5>
Producto escalar de a * 3 <x, y>=<6,21>
```

Validar los datos recibidos

Qué pasa si ejecuto la instrucción: **s = a + 5**
AttributeError: 'int' object has no attribute 'getX'

```
def __add__(self, otroVector):
    if type(self) == type(otroVector):
        return Vector2D(self.__x+otroVector.getX(), self.__y+otroVector.getY())
    else:
        if type(otroVector)==int or type(otroVector)==float:
            return Vector2D(self.__x+otroVector, self.__y)
```

Luego de la exitosa programación de la clase Vector2D, se le encarga la programación de la clase Fracción, que corresponde al diseño elaborado por el analista funcional de la empresa, y que corresponde al diagrama UML:

Fraccion
- numerador: int - denominador: int
__init__(...) __str__(...): string + getNumerador(): int + getDenominador(): int

Según las especificaciones elaboradas por el analista funcional, la case tendrá que permitir:

- La suma, resta, multiplicación y división de dos fracciones.
- Se deberá permitir realizar operaciones:
  - a+5, 5+a, siendo a un objeto de la clase Fracción
  - a-5, 5-a, siendo a un objeto de la clase Fracción
  - a*3, 3*a, siendo a un objeto de la clase Fracción
  - a/4, 4/a, siendo a un objeto de la clase Fracción
- Se deben poder comparar fracciones, para saber si una es mayor o igual que otra, si son iguales o distintas.

Destrucción de Objetos de una Clase (I)

Destructores y Recolección de Basura

El lenguaje de programación Python, utiliza el mecanismo de recolección de basura (Garbage Collector), para obtener el espacio de objetos que ya no están referenciados.

El principio fundamental que utiliza, se basa en el ciclo de vida de los objetos, si un objeto deja de estar referenciado, es candidato a la recolección de basura.

Las clases pueden proveer un método para la liberación de recursos que ya no se ocupen, a este método se los denomina destructor.

Si el programador no provee el código de un destructor, la clase tendrá un **destructor por defecto u omisión**.

El destructor es invocado automáticamente por el Garbage Collector (recolector de basura) para liberar los recursos utilizados por el objeto que se va a destruir.

El programador puede escribir código específico para la liberación de recursos, este código se ubicará en el destructor de la clase, que tiene el nombre reservado __del__.

(*1) Chapter 3 - Páginas 929 y 930
(*2) Chapter 2 – Páginas 14 y 15

https://docs.python.org/3/library/gc.html

Destrucción de Objetos de una Clase (II)

Destructores y Recolección de Basura

```
class CicloDeVida:
    __nombre: str
    def __init__(self, nombre):
        print('Hola: ',nombre)
        self.__nombre = nombre
    def vida(self):
        print(self.__nombre)
    def __del__(self):
        print('Chau... ',self.nombre)

def funcion():
    o = CicloDeVida('Violeta')

if __name__=='__main__':
    objeto = CicloDeVida('Carlos')
    objeto.vida()
    del objeto
    funcion()
```

Consola Python

```
Hola:  Carlos
Carlos
Chau...  Carlos
Hola:  Violeta
Chau...  Violeta
```

Referencias circulares entre clases

```
class A:
    __b: None
    def __init__(self, unObjetoB):
        print(type(unObjetoB))
        self.__b=unObjetoB
    def __del__(self):
        print('Chau objeto A')

class B:
    __a: None
    def __init__(self):
        self.__a=A(self)
    def __del__(self):
        print('Chau Objeto B')

def funcionCreaB():
    b = B()
    del b
```

```
if __name__=='__main__':
    funcionCreaB()
```

Consola Python

En este ejemplo, cuando se llama a la función funcionCreaB(), crea una instancia de clase B que se pasa a la clase A, que luego establece una referencia a la clase B y da como resultado una referencia circular.

En general, el recolector de basura de Python que se usa para detectar este tipo de referencias cíclicas lo eliminaría, pero en este ejemplo el uso de destructor personalizado marca este elemento como «**uncollectable**» (imposible de recolectar).

Simplemente, no sabe el orden en el que destruir los objetos, por lo que los deja. Por lo tanto, si sus instancias están involucradas en referencias circulares, vivirán en la memoria mientras se ejecute la aplicación.

Garbage Collector – Reference Counting (I)

A diferencia de muchos otros lenguajes, Python no necesariamente libera la adminsitración de memoria al sistema operativo
Tiene un asignador de objetos dedicado para objetos de menos de 512 bytes, que mantiene algunos fragmentos de memoria ya asignada para su uso posterior.
El garbage collector estándar de Python tiene dos componentes:

- El recolector de basura por conteo de referencias
- El recolector de basura generacional, conocido como el módulo gc

El algoritmo de conteo de referencias es increíblemente eficiente, pero no puede detectar referencias cíclicas.
Es por eso que Python tiene un algoritmo suplementario llamado GC generacional cíclico, que trata específicamente con las referencias cíclicas.

El módulo de conteo de referencias es fundamental para Python y no se puede deshabilitar, mientras que el GC cíclico es opcional y se puede invocar manualmente.
Si el contador de referencias llega a cero, automáticamente se invoca a la función de liberación de memoria para el objeto específico.

Para **eliminar algo de la memoria, se debe asignar un nuevo valor a una variable o salir de un bloque de código.**
En Python, **el bloque de código más popular es una función**.
Aquí es donde ocurre la mayor parte de la recolección de basura. Esa es otra razón para **mantener las funciones pequeñas y simples**.

https://docs.python.org/3.6/c-api/intro.html#objects-types-and-reference-counts

Garbage Collector – Reference Counting (II)

Se puede conocer la cantidad de referencias de un objeto usando la función sys.getrefecount

```
import sys
def funcion(a):
    # 4 referencias
    # variable comidas, argumento formal a, getrefcount y pila de funciones de Python
    print('Cantidad de referencias de a: ',sys.getrefcount(a))
    print(sys.getrefcount(a))

comidas = []

# 2 referencias,  1 por la variable comida y una por getrefcount
print('Antes de llamar a la función')
print('Cantidad de referencias de comidas: ',sys.getrefcount(comidas))
funcion(comidas)

# 2 referencias, las propias del alcance de la función se destruyen
print('Después de llamar a la función')
print('Cantidad de referencias de comidas: ',sys.getrefcount(comidas))
```

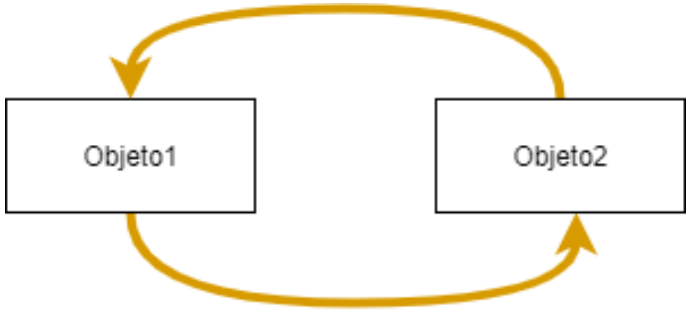
Consola Python

```
Antes de llamar a la función
Cantidad de referencias de comidas:  2
Cantidad de referencias de a:  4
Después de llamar a la función
Cantidad de referencias de comidas:  2
```

Garbage Collector Generacional (I)

¿Por qué se necesita un recolector de basura adicional cuando se tiene el conteo de referencias?

Desafortunadamente, el conteo clásico de referencias tiene un problema fundamental: no puede detectar referencias cíclicas.
Una referencia cíclica ocurre cuando uno o más objetos están haciendo referencia entre sí.



Objeto1 referencia a Objeto2, y Objeto2 referencia a Objeto1

Objeto3, se referencia a si mismo

Garbage Collector Generacional (II)

```
import ctypes
import gc
# We use ctypes module  to access our unreachable objects by memory address.
class PyObject(ctypes.Structure):
    _fields_ = [("refcnt", ctypes.c_long)]

gc.disable()  # Desactiva el Garbage Collector generacional (gc)

lst = []
lst.append(lst)

# Guarda la dirección de lista
lst_address = id(lst)

# Destruye la refencia lst
del lst

object_1 = {}
object_2 = {}
object_1['obj2'] = object_2
object_2['obj1'] = object_1

obj_address = id(object_1)

del object_1, object_2 # destruye las referencias
# Quitar comentario para ver resultados de llamar manualmente al recolector de basura
#gc.collect()

# Ver reference count de las direcciones de objetos
print('Dirección: ',hex(obj_address), 'Referencias: ',PyObject.from_address(obj_address).refcnt)
print('Dirección: ',hex(lst_address), 'Referencias: ',PyObject.from_address(lst_address).refcnt)
```

Consola Python

```
Dirección:  0x24a2ede6e58 Referencias:  1
Dirección:  0x24a2edefe88 Referencias:  1
```

Garbage Collector Generacional (III)

```
import ctypes
import gc
# We use ctypes moule  to access our unreachable objects by memory address.
class PyObject(ctypes.Structure):
    _fields_ = [("refcnt", ctypes.c_long)]

gc.disable()  # Desactiva el Garbage Collector generacional (gc)

lst = []
lst.append(lst)

# Guarda la dirección de lista
lst_address = id(lst)

# Destruye la refencia lst
del lst

object_1 = {}
object_2 = {}
object_1['obj2'] = object_2
object_2['obj1'] = object_1

obj_address = id(object_1)

# destruye las referencias
del object_1, object_2

# Llama manualmente al recolector de basura
gc.collect()

# Ver reference count de las direcciones de objetos
print('Dirección: ',hex(obj_address), 'Referencias: ',PyObject.from_address(obj_address).refcnt)
print('Dirección: ',hex(lst_address), 'Referencias: ',PyObject.from_address(lst_address).refcnt)
```

Consola Python

```
Dirección:  0x24a2edead18 Referencias:  0
Dirección:  0x24a2edf32c8 Referencias:  0
```

Fin Unidad 2