Paradigmas de la Programación con Python

Carlos Zayas Guggiari

22 de marzo de 2019

Resumen

Este documento fue preparado para servir de apoyo a la asignatura "Paradigmas de la Programación" de la carrera de Licenciatura en Ciencias Informáticas de la Facultad Politécnica de la Universidad Nacional de Asunción. En él, se examinan los principales modelos de programación usando como ejemplo código escrito en Python, un lenguaje de programación multiparadigma.

Contenido

1. Introducción

1.1. ¿Qué es un paradigma?

La palabra "paradigma" proviene del griego *pará* (junto) y *deigma* (modelo), y significa algo así como "demostración de un modelo".

Un paradigma es un esquema formal de organización, un marco teórico, un conjunto de teorías alrededor de un tema determinado.

1.2. ¿Qué es un lenguaje?

Un lenguaje, en su sentido natural, es un método humano, no instintivo, que sirve para comunicar ideas, emociones y deseos mediante un sistema de símbolos producidos voluntariamente.

Un lenguaje, en su sentido formal, es aquel que tiene una gramática que consta de un vocabulario de símbolos, reglas sintácticas (combinaciones) y reglas semánticas (significado).

El lenguaje, en general, es entonces un conjunto de objetos (vocabulario) capaces de ser combinados de acuerdo a ciertas reglas (gramática) con el fin de comunicarse con un receptor.

1.3. ¿Qué es un programa?

Un programa, en un contexto informático, es una secuencia de instrucciones que permiten realizar una tarea específica con una computadora.

Dichas instrucciones primero son escritas por un programador en un lenguaje de programación para luego ser convertidas en un formato procesable por la computadora.

- En su forma legible para el ser humano, el programa se denomina **código fuente**.
- En su forma procesable por la computadora, el programa se denomina **código ejecutable**.

Para que un programa escrito en un lenguaje de programación "humano" (de alto nivel) pueda ser ejecutado, éste debe convertirse a **código máquina** mediante un programa traductor que funciona como intérprete (traduce y ejecuta cada instrucción) o compilador (traduce y guarda las instrucciones en código máquina en un archivo nuevo).

La conversión del código fuente al código ejecutable se realiza mediante uno de estos procesos:

- El **intérprete** es un programa que traduce y ejecuta secuencialmente cada una de las instrucciones del código fuente, sin conservar el resultado de dicha traducción.
- El **compilador** es un programa que traduce el código fuente y genera un nuevo programa escrito en un lenguaje de nivel inferior, típicamente lenguaje de máquina.

Un intérprete ofrece al programador más flexibilidad en su entorno de programación, y al programa un entorno de ejecución independiente de la máquina donde se ejecuta, lo que se conoce comúnmente como máquina virtual.

Un compilador, por su parte, permite que un programador pueda diseñar un programa en un lenguaje mucho más cercano a como piensa un ser humano, para luego convertirlo a un programa que pueda ser procesable directamente por una computadora.

1.4. Código binario

El código binario es el sistema de representación de instrucciones y datos que utiliza un dispositivo electrónico. Se basa en el sistema de numeración binario (de dos dígitos, "0" y "1") que representan los dos únicos niveles de tensión que usan los circuitos electrónicos.

La unidad básica de información del código binario es el **bit** (binary digit) que es una variable cuyo valor almacenado puede representar sólo a uno de dos posibles estados: encendido/apagado, verdadero/falso, sí/no, etc.

El código binario puede representar cualquier tipo de información, por ejemplo la letra minúscula "a" en la tabla ASCII (*American Standard Code for Information Interchange*) está representada por la cadena de bits "01100001" que tiene un valor decimal de 97.

1.5. Arquitectura del conjunto de instrucciones

Del inglés "Instruction Set Architecture" (ISA) es una especificación que detalla las instrucciones que una CPU en particular puede interpretar y ejecutar.

ISA es la parte de la arquitectura de hardware relacionada con la programación, e incluye los tipos de datos, instrucciones, registros, modos de direccionamiento, arquitectura de memoria, interrupciones y manejo de excepciones.

1.6. Lenguaje de programación

Un lenguaje de programación es un subconjunto de los lenguajes formales que le permite al ser humano comunicarse con una computadora (receptor) para indicarle a ésta qué es lo que debe hacer.

Se puede considerar a un lenguaje de programación como una **capa de abstracción** que le protege al ser humano (programador) de las complejidades o particularidades de la computadora. Esa capa le presenta al programador una **máquina virtual** que le resulta mucho más fácil de comprender que la máquina real.

1.7. Lenguaje de máquina

El lenguaje de máquina es un sistema de instrucciones que pueden ser ejecutadas directamente por una unidad central de procesamiento (microprocesador). Se lo considera un lenguaje de programación primitivo e incómodo, por estar basado directamente en la lógica digital y por no haber un único lenguaje de máquina, sino que cada arquitectura de hardware cuenta con el suyo.

El código máquina a veces se denomina código nativo, por haber partes que dependen de una plataforma de hardware en particular. No debe confundirse al código máquina con bytecode (código de bytes) ya que este último es un código intermedio que se ejecuta a través de un intérprete o máquina virtual, no directamente por la CPU.

1.8. Ensamblador

El término "ensamblador" puede referirse a un programa especial o a un lenguaje de programación (Assembly). El Assembly o Ensamblador es un tipo de lenguaje de bajo nivel (cercano al hardware) que permite una representación un poco más abstracta de las instrucciones en código máquina.

El programa ensamblador es capaz de leer un archivo con instrucciones escritas en lenguaje Assembly y traducirlas a código máquina, para posteriormente almacenarlas en un nuevo archivo, denominado objeto o ejecutable.

Los ensambladores fueron creados para facilitar la escritura de programas en código máquina, ya que hacerlo en código binario resulta muy complicado para el ser humano.

1.9. Máquina virtual

Una máquina virtual es una implementación en software de una máquina real o física (hardware). Dicha máquina virtual debe poder ejecutar órdenes diseñadas para la máquina real.

Lo que comúnmente se entiende por máquina virtual es lo que en términos técnicos más específicos se conoce como máquina virtual **de sistema**, que permite compartir los recursos físicos (hardware) entre distintas máquinas virtuales, cada una de ellas funcionando sobre su propio sistema operativo.

Los lenguajes de programación pueden considerarse máquinas virtuales de un tipo especial, denominado máquina virtual **de proceso** o **de aplicación**, debido a que funcionan sobre un sistema operativo y soportan un único proceso.

Un lenguaje de programación provee al programador de un entorno de desarrollo independiente de la plataforma (hardware + sistema operativo) lo que al menos en teoría permite que un mismo programa funcione en cualquiera de ellas.

1.10. Tipado de datos

Un sistema de tipos (o tipado de datos) define cómo un lenguaje de programación clasifica los valores y las expresiones en tipos, cómo se pueden manipular estos tipos y cómo interactúan.

Un tipo indica un conjunto de valores que tienen el mismo significado genérico o propósito.

Los sistemas de tipificación varían significativamente entre lenguajes, siendo quizás las más importantes variaciones las implementadas en tiempo de **compilación** y en tiempo de **ejecución**.

Los tipados, de acuerdo a criterios determinados, pueden categorizarse en:

- **Fuerte** y **débil** (según cómo se combinan los datos)
- **Estático** y **dinámico** (según cuándo se comprueban los tipos de datos)
- **Explícito** e **implícito** (según si se declaran o no los datos)

1.10.1. Strong typing

Un lenguaje de programación es **fuertemente tipado** si no se permiten violaciones de los tipos de datos, es decir, dada 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.

No hay una única definición de este término. Un lenguaje que no es fuertemente tipado se dice que no está tipado. La mayoría de los lenguajes imperativos son fuertemente tipados mientras que los lenguajes declarativos no suelen estar tipados.

Algunos de los lenguajes fuertemente tipados son: C++, C#, Java, Python.

Ejemplo (en seudocódigo):

```
a = 2
b = "2"
concatenar(a, b)      # Error de tipo
sumar(a, b)           # Error de tipo
concatenar(str(a), b) # Retorna "22"
sumar(a, int(b))      # Retorna 4
```

1.10.2. Weak typing

Los lenguajes de programación no tipados o **débilmente tipados** no controlan los tipos de las variables que declaran, de este modo, es posible usar variables de cualquier tipo en un mismo escenario. Por ejemplo, una función puede recibir como parámetro un valor entero, cadena de caracteres, flotante...

No hay que confundir el término con los lenguajes de tipado dinámico, en los que no se declaran los tipos de las variables sino se deciden en tiempo de ejecución, si bien es cierto que muchos lenguajes de este tipo son también no tipados.

Algunos de los lenguajes débilmente tipados son: BASIC, JavaScript, Perl, PHP.

Ejemplo (en seudocódigo):

```
a = 2
b = "2"
concatenar(a, b) # Retorna "22"
```

```
sumar(a, b)      # Retorna 4
```

1.10.3. Static typing

Se dice que un lenguaje de programación usa un **tipado estático** cuando el chequeo de tipificación se realiza durante el tiempo de **compilación**, opuesto al de ejecución.

Comparado con el tipado dinámico, el estático permite que los errores de programación sean detectados antes, y que la ejecución del programa sea más eficiente.

Como ejemplos de lenguajes que usan tipado estático tenemos a C, C++, Java y Haskell.

1.10.4. Dynamic typing

Un lenguaje de programación tiene **tipado dinámico** si una misma variable puede tomar valores de distinto tipo en distintos momentos durante la ejecución.

La separación entre tipado estático y dinámico se suele confundir con la diferencia entre lenguajes fuertemente tipados y lenguajes débilmente (o no) tipados.

La mayoría de los lenguajes de tipado dinámico son lenguajes interpretados, como Python o Ruby.

1.10.5. Duck typing

En los lenguajes de programación orientados a objetos, se conoce como *duck typing* o **tipado de pato** al estilo de tipificación dinámica de datos en que el conjunto actual de métodos y propiedades determina la validez semántica, en vez de que lo hagan la herencia de una clase en particular o la implementación de una interfaz específica.

Dicho de una manera más simple, el tipado de pato, característica común en los lenguajes dinámicos, hace posible que no importe de qué tipo sea el dato que estoy manejando, sino lo que sea capaz de hacer con él.

El nombre del concepto se refiere a la prueba del pato, una humorada de razonamiento inductivo atribuida a James Whitcomb Riley, que es como sigue: *"Cuando veo un ave que camina como un pato, nada como un pato y suena como un pato, a esa ave yo la llamo un pato."*

Python es un buen ejemplo de lenguaje de programación que implementa el *duck typing*. Por ejemplo, el operador de asignación de suma (+=) se puede usar con listas y cadenas de caracteres, además de números:

```
>>> a = 1
>>> a += 1
>>> a
2
>>> l = ["Hugo", "Paco"]
>>> l += ["Luis"]
>>> l
['Hugo', 'Paco', 'Luis']
>>> c = "Hola, "
>>> c += "Mundo"
>>> c
'Hola, Mundo'
```

2. El lenguaje de programación Python

2.1. ¿Qué es Python?

Python es un lenguaje de programación de alto nivel, interpretado, multiparadigma, dinámico y fuertemente tipado, creado en 1991 por el programador holandés Guido van Rossum, basado en un fuerte énfasis en la simplicidad y legibilidad del código, para así hacer posible el desarrollo rápido de aplicaciones.

La mayoría de las veces, un programador experimentado en Python es capaz de escribir una cantidad considerablemente menor de líneas de código para realizar la misma tarea comparada con otros lenguajes. Esto conlleva a menos errores de programación y reduce el tiempo necesario para el desarrollo de un proyecto.

Otra ventaja de Python la constituye su enorme biblioteca de módulos, distribuida entre los que comprenden la llamada **Biblioteca Estándar** (*Python Standard Library*) y los desarrollados por terceras partes, disponibles en el **Índice de Paquetes** (PyPI: Python Package Index) e instalables mediante el administrador de paquetes pip.

2.2. ¿Qué se necesita para programar en Python?

Python está disponible para prácticamente todos los sistemas operativos, y suele venir incluido en la mayoría de las distribuciones Linux, por lo tanto es probable que muchos usuarios ya tengan en su equipo todo lo necesario para poder empezar a programar en este lenguaje. De no ser así, puede descargarse el instalador correspondiente desde https://www.python.org/downloads/

En nuestra asignatura, usaremos la versión 3 de Python, a pesar de que la versión 2 sigue siendo muy popular al momento de escribir ésto, pero las diferencias entre ambas versiones no son muy notables, hasta el punto de que es posible escribir programas que sean completamente compatibles con cualquier versión.

Para que la experiencia con Python sea más cómoda, se recomienda usar un editor o IDE (*Integrated Development Environment*, entorno integrado de desarrollo) que tenga a Python entre los lenguajes de programación soportados.

2.3. El modo interactivo

Python cuenta con un **modo interactivo** donde el programador puede introducir directamente por teclado una instrucción para ser evaluada y ejecutada inmediatamente por el intérprete.

El modo interactivo constituye una valiosa herramienta para el programador, ya que le permite aprender nuevos conceptos y probar ideas antes de incorporarlas al código de la aplicación que está desarrollando.

```
Python 3.6.4 (default, Mar  5 2018, 12:14:58)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> "Hola, Mundo.".upper()
'HOLA, MUNDO.'
>>> 2+3
5
>>> for i, e in enumerate([1,2,3,4,5]): print("{}:{}".format(i, e))
...
0:1
1:2
2:3
3:4
4:5
>>>
```

Hay una gran cantidad de editores e IDEs que soportan Python, y la mayoría de ellos incluyen en su interfaz un panel con Python en modo interactivo. Probablemente el más conocido de todos sea PyCharm.

3. Los Paradigmas de la Programación

3.1. ¿Qué es un paradigma de programación?

Un paradigma de programación es un conjunto de métodos sistemáticos aplicables en el diseño de programas, basados en un modelo de definición y operación de la información.

Los paradigmas de programación se diferencian entre sí por los conceptos, abstracciones y pasos que usan para representar los elementos de un programa.

3.2. Paradigmas fundamentales

- **Imperativo**: Basado en una secuencia de instrucciones. Describe la programación en términos del estado del programa y las sentencias que cambian dicho estado. A estos cambios de estado se los considera *efectos secundarios*.
- **Declarativo**: Expresa la lógica de la computación, sin describir su flujo de control. Está basado en el desarrollo de programas especificando o declarando un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución. No permite *efectos secundarios*.

3.3. El paradigma imperativo

La implementación de hardware de la mayoría de las computadoras es imperativa. Prácticamente todo el hardware está diseñado para ejecutar código máquina, que es el lenguaje nativo de la computadora, y está escrito en forma imperativa.

Desde esta perspectiva de bajo nivel, el estilo del programa está definido por los contenidos de la memoria, y las sentencias son instrucciones en el lenguaje de máquina nativo de la computadora, o por sus equivalencias en lenguaje ensamblador.

A partir del paradigma imperativo, evolucionaron los siguientes paradigmas:

- **Estructurado**: Consiste en el uso de estructuras jerárquicas de flujo con el fin de mejorar la claridad de los programas y reducir el tiempo de desarrollo.
- **Modular**: Consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.

Actualmente, el paradigma de programación dominante en la industria del desarrollo de software es el **orientado a objetos**, que podría considerarse una evolución de los paradigmas anteriormente mencionados. En la orientación a objetos, los datos se incluyen con los métodos que los usan en entidades denominadas objetos, que son instancias de una clase, la cual está compuesta de datos y los métodos correspondientes para acceder a ellos y manipularlos (cambios de estado).

3.4. El paradigma declarativo

La programación declarativa es lo opuesto a la programación imperativa. Un programa declarativo es aquel que describe lo que se debe realizar y no la forma de realizarlo.

Los lenguajes declarativos no basan su funcionamiento en efectos secundarios, y tienen una clara correspondencia con la lógica matemática.

El término "programación declarativa" es en realidad un término general que engloba una serie de paradigmas de programación más conocidos:

- **Funcional**: Basado en la evaluación de funciones matemáticas.
- **Lógico**: Basado en el uso de sentencias lógicas para representar y evaluar programas.

4. El Paradigma Estructurado

4.1. Estructuras jerárquicas de flujo

En la programación estructurada se utilizan únicamente tres tipos de estructuras: **secuencia**, **selección** e **iteración**, siendo innecesario el uso de instrucciones de transferencia incondicional como GOTO, GOSUB o EXIT que actualmente están casi en desuso.

Estructura secuencial: Las instrucciones se ejecutan una tras otra, a modo de secuencia lineal. Una instrucción no se ejecuta hasta que finaliza la anterior, ni se bifurca el flujo del programa.

```
nombre = input('¿Cuál es tu nombre? ')
print('Hola,', nombre)
```

Estructura selectiva: La ejecución del programa se bifurca a una instrucción (o conjunto) u otra(s) según un criterio o condición lógica establecida. Sólo uno de los caminos en la bifurcación será el tomado para ejecutarse.

Palabras reservadas de Python: `if-else-elif`

```
nombre = input('¿Cuál es tu nombre? ')
if nombre.isdigit():
    print('Dije nombre, no edad...')
elif nombre:
    print('Hola,', nombre)
else:
    print('No seas timido...')
```

Estructura iterativa: Dada una secuencia de instrucciones, un bucle iterativo o iteración hace que se repita su ejecución mientras se cumpla una condición. El número de iteraciones normalmente está determinado por el cambio en la condición dentro del mismo bucle, aunque puede ser forzado o explícito por otra condición.

Palabras reservadas de Python: `for, while`

```
nombre = 'x'
while nombre:
    nombre = input('¿Cuál es tu nombre? ')
    if nombre.isdigit():
        print('Dije nombre, no edad...')
    elif nombre:
        print('Hola,', nombre)
        vocales = 0
        for letra in nombre:
            if letra in 'aeiou':
                vocales += 1
        print('Tu nombre tiene', vocales, 'vocales')
    else:
        print('No seas timido...')
```

5. El Paradigma Modular

5.1. ¿Qué es un módulo?

Un módulo es cada una de las partes de un programa que resuelve uno de los subproblemas en que se divide el problema complejo original. Cada uno de estos módulos tiene una tarea bien definida y algunos necesitan de otros para poder operar.

5.2. Entrada-Proceso-Salida

Cada módulo puede o no aceptar valores de entrada y retornar valores de salida. El proceso interno puede o no estar relacionado con dichos valores. Los valores de entrada se denominan indistintamente **parámetros**, **argumentos** o **atributos**.

5.3. Divide y vencerás (top-down)

Top-down es una técnica de refinamiento sucesivo o análisis descendente, donde un problema complejo debe ser dividido en varios subproblemas más simples, y éstos a su vez en otros subproblemas más simples. Esto debe hacerse hasta obtener subproblemas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación.

5.4. Procedimientos y funciones

Son subprogramas, también denominados submódulos, ya que en algunos lenguajes, como por ejemplo Python, se considera como módulo a un conjunto de subprogramas.

Un **procedimiento** es básicamente una función que no retorna ningún valor. Una **función** siempre retorna un valor al programa que la llamó. Tanto los procedimientos como las funciones pueden o no aceptar parámetros de entrada.

Palabras reservadas de Python: `import, def, return, yield`

```
import random
def azar(limite):
    return random.randint(1, limite)
def genazar(limite):
    yield random.randint(1, limite)
def par(numero):
    return numero % 2 == 0
def pares(limite):
    for numero in range(limite):
        if numero % 2 == 0:
            yield numero
```

6. El Paradigma Orientado a Objetos

6.1. Introducción

La programación orientada a objetos (POO) es un paradigma de programación que permite representar un concepto en una estructura denominada **clase**, que integra a las **propiedades** que describen al concepto, y a los **métodos** que son los procedimientos asociados a dichas propiedades.

Los objetos se crean a partir de las clases, y pueden interactuar entre sí mediante mensajes. Se puede considerar a un objeto como compuesto de sustantivos (variables) y verbos (funciones).

En la actualidad, la orientación a objetos constituye el paradigma más importante de la industria del desarrollo de software. Smalltalk, C++, Objective-C, C#, Java, Javascript y Python son sólo algunos ejemplos de lenguajes de programación disponibles que implementan la orientación a objetos.

Un programa orientado a objetos puede verse como una colección de objetos que interactúan entre sí, en oposición a la visión del modelo convencional, en la que se ve un programa como una lista de tareas (subrutinas) para llevar a cabo.

```
class MiClase:                          # Definición de una clase
    def MiMetodo(self):                 # Definición de un método ('self' es el objeto)
        self.MiPropiedad = "Mi Dato"    # Definición de una propiedad de objeto

MiObjeto = MiClase()                    # Instanciación de una clase
MiObjeto.MiMetodo()                     # Ejecución de un método
print(MiObjeto.MiPropiedad)             # Obtención de la propiedad de un objeto
```

Cada objeto es capaz de recibir mensajes, procesar datos, y enviar mensajes a otros objetos, como si se tratara de una "máquina" o **caja negra**, independiente, con un papel distinto o una responsabilidad única. Tanto los datos (llamados propiedades) como las acciones sobre esos datos (llamadas métodos) están estrechamente asociados con el objeto correspondiente.

6.2. Clase

Una clase es un modelo o plantilla que describe un tipo de objeto. El intérprete o compilador utiliza la clase cada vez que se tenga que crear un nuevo objeto. Este proceso se llama **instanciación**. Cada objeto es una instancia de alguna clase.

6.3. Objeto (Instancia)

Un objeto es una entidad de programación que contiene propiedades y métodos, de acuerdo a un modelo definido al que se denomina clase. Cada objeto tiene una **identidad** (nombre o identificador), un **estado** (datos o propiedades) y un **comportamiento** (acciones o métodos).

6.4. Propiedades

Las propiedades pueden ser de clase o de instancia:

```
class OtraClase:
    PropiedadClase = 0
    def __init__(self, n):          # Método constructor de la clase
        self.PropiedadObjeto = n    # Propiedad de instancia

Objeto1 = OtraClase(1)              # Instancia 1: Inicializa la propiedad de Objeto1
Objeto2 = OtraClase(2)              # Instancia 2: Inicializa la propiedad de Objeto2

print(Objeto1.PropiedadClase)       # Imprime: 0
print(Objeto1.PropiedadObjeto)      # Imprime: 1
```

```
print(Objeto2.PropiedadClase)     # Imprime: 0
print(Objeto2.PropiedadObjeto)    # Imprime: 2

OtraClase.PropiedadClase = 3      # Modifica la propiedad de la clase

print(Objeto1.PropiedadClase)     # Imprime: 3
print(Objeto2.PropiedadClase)     # Imprime: 3
```

- **de Clase**: Una propiedad de clase es aquella cuyo estado es compartido por todas las instancias de la clase.
- **de Instancia**: Una propiedad de instancia (o de objeto) es aquella cuyo estado puede ser invocado o modificado únicamente por la instancia a la que pertenece.

6.5. Métodos

Un método es una función, procedimiento o subrutina que forma parte de la definición de una clase. Los métodos, mediante sus parámetros de entrada, definen el comportamiento de las instancias de la clase asociada.

6.5.1. El parámetro "self"

La palabra *self* representa a la instancia de una clase, es decir, a un objeto. Se utiliza dentro de la clase para hacer referencia a las propiedades del objeto en sí y diferenciarlas de las de los métodos y la propia clase.

```
class Persona:

    cantidad = 0

    def __init__(self, nombre):
        self.nombre = nombre
        Persona.cantidad += 1

    def saludar(self):
        print("Hola, mi nombre es", self.nombre)
        print("Somos", Persona.cantidad)

    def __del__(self):
        print("{} dice adiós.".format(self.nombre))

sobrino_donald = dict()
for nombre in ["Hugo", "Paco", "Luis"]:
    sobrino_donald[nombre] = Persona(nombre)
    sobrino_donald[nombre].saludar()
```

En el ejemplo, `nombre` es un atributo o parámetro del método `__init__`, mientras que `self.nombre` es una propiedad de instancia y `cantidad` es una propiedad de clase.

6.5.2. Sobrecarga de operadores

Consiste en la posibilidad de contar con métodos del mismo nombre pero con comportamientos diferentes.

```
class Lista:
    def __init__(self):
        self.items = []
    def __str__(self):
        return '\n'.join(self.items)
    def __repr__(self):
        return repr(self.items)
```

```
    def agregar(self, item):
        self.items.append(str(item))
    def vaciar(self):
        self.items = []
    def vacio(self):
        return self.items == []
    def cantidad(self):
        return len(self.items)

class Archivo:
    def __init__(self, nombre='lista.txt'):
        self.items = open(nombre).readlines() if os.path.exists(nombre) else []
        self.items = [ item.strip() for item in self.items ] # Quita espacios sobrantes
        self.nombre = nombre
    def __del__(self):
        open(self.nombre,'w').write('\n'.join(self.items))

class ListArch(Archivo, Lista):
    pass

lista = ListArch()
lista.vaciar()
lista.agregar('Hola')
lista.agregar(2)
lista.agregar((3, 4))
lista.agregar('Chau')

print("Cantidad:", lista.cantidad(), '\n')
print(lista)
print('\nLista:',repr(lista))
```

En el código anterior, `__init__`, `__repr__`, `__str__` y `__del__` son ejemplos de **redefiniciones de métodos predefinidos** que modifican parte del comportamiento estándar de los objetos, en este caso la construcción y la impresión del objeto, respectivamente. A esta capacidad del lenguaje se le denomina **sobrecarga**.

- `__repr__` devuelve una representación del objeto en forma de cadena de caracteres, que puede volver a generar el objeto si es evaluada nuevamente, por ejemplo con la función `eval()`.
- `__str__` devuelve una cadena de caracteres cuando se imprime el objeto con la sentencia `print()`.

6.5.3. Constructor y Destructor

En Python, los métodos `__init__` y `__del__` son lo que se conoce como método **constructor** y **destructor** respectivamente. No es obligatorio incluirlos cuando definimos una clase, pero como se verá a continuación, pueden ser de mucha utilidad, y a menudo son necesarios.

- El método **constructor** de una clase se ejecuta en cada instanciación de la clase, es decir, durante el proceso de creación de un objeto.
- El método **destructor** de una clase se ejecuta cuando el objeto ya no es referenciado dentro del programa, o simplemente cuando el proceso termina.

Recordemos que un proceso es un programa en ejecución, y durante su finalización, se borran de la memoria todos los objetos creados, lo que activa el método destructor correspondiente a cada uno de ellos.

6.5.4. Getter / Setter / Deleter

Las propiedades, tanto las de clase como las de instancia, se pueden acceder directamente utilizando la sintaxis "punto" (objeto.propiedad). Sin embargo, se recomienda que cualquier operación relacionada con

propiedades se realice mediante métodos especiales, denominados **getters** y **setters** (obtenedores y colocadores). Opcionalmente, también puede definirse un **deleter** (borrador).

Estos métodos especiales son necesarios principalmente para el manejo de las propiedades más importantes del objeto, generalmente aquellas que necesitan ser accedidas desde otros objetos, no precisamente desde los métodos propios del objeto.

No es obligatorio definir getters y setters para todas las propiedades, sino sólo para aquellas en las que necesitemos algún tipo de validación previa antes de obtener o colocar el valor involucrado.

En Python contamos con dos maneras de definir estos métodos especiales:

```
# Ejemplo 1: Función 'property'

class Ejemplo1(object):
    def __init__(self):
        self._x = 1
    def getx(self):
        return self._x
    def setx(self, valor):
        self._x = valor
    def delx(self):
        del self._x
    x = property(getx, setx, delx, "Soy la propiedad 'x'.")

# Ejemplo 2: Decoradores ('@...')

class Ejemplo2(object):
    def __init__(self):
        self._x = 2
    @property
    def x(self):
        "Soy la propiedad 'x'."
        return self._x
    @x.setter
    def x(self, valor):
        self._x = valor
    @x.deleter
    def x(self):
        del self._x
```

Ambas sintaxis cumplen el mismo propósito y dan exactamente los mismos resultados, pero la segunda es la más nueva incorporación al lenguaje Python.

También contamos con los métodos `__getitem__` y `__setitem__` para manipular listas y/o diccionarios usando únicamente el nombre del objeto. De esta manera, en vez de escribir `objeto.lista[indice]` simplemente podemos escribir `objeto[indice]`.

```
class Dic:
    def __init__(self):
        self.dic = {}
    def __repr__(self):
        return repr(self.dic)
    def __getitem__(self, clave):
        return self.dic.get(clave, None)
    def __setitem__(self, clave, valor):
        self.dic[clave] = valor
```

De esta manera, una instancia de la clase `Dic` puede usarse como si se tratase de un diccionario:

```
d = Dic()
d['Nombre'] = 'Carlos'
```

```
print(d['Nombre'])
```

6.6. Conceptos Fundamentales

La programación orientada a objetos se basa en los siguientes conceptos fundamentales:

6.6.1. Encapsulamiento

Consiste en colocar al mismo nivel de abstracción a todos los elementos (estado y comportamiento) que pueden considerarse pertenecientes a una misma entidad (identidad). Esto permite aumentar la cohesión de los componentes del sistema.

```
class Persona(object):

    def __init__(self, nombre):
        self.setNombre(nombre)

    def getNombre(self):
        return ' '.join(self.__nombre)

    def setNombre(self, nombre):
        self.__nombre = nombre.split()

    nombre = property(getNombre, setNombre)
```

Las propiedades pertenecen al espacio de nombres del objeto (namespace) y pueden estar ocultas, es decir, sólo accesibles para el objeto. En Python, esto último se logra superficialmente anteponiendo dos guiones bajos (__) al nombre de la propiedad.

```
carlos = Persona("Carlos Zayas")
print(carlos.nombre.upper())

carlos.nombre = 'Carlos A. Zayas G.'
print(carlos.nombre.upper())
```

Decimos que en Python se logra sólo superficialmente el ocultamiento de las propiedades porque, si bien no podemos invocar la propiedad nombre de manera tradicional, sí podemos hacerlo de la siguiente manera:

```
print(carlos._Persona__nombre)
```

6.6.2. Herencia

Las clases se relacionan entre sí dentro de una jerarquía de clasificación que permite definir nuevas clases basadas en clases preexistentes, y así poder crear objetos especializados.

```
class Empleado(Persona):

    def __init__(self, nombre, cargo):
        Persona.__init__(self, nombre)
        self.cargo = cargo
        print("{} es {}".format(self.nombre, self.cargo))
```

La herencia es el mecanismo por excelencia de la programación orientada a objetos que permite lograr la reutilización de código. Mediante ella, podemos crear nuevas clases modificando clases ya existentes.

6.6.2.1. Herencia Múltiple

Cuando la herencia involucra a más de una clase, hay herencia múltiple. El orden en el que las clases son invocadas determina la prevalencia de propiedades y métodos de idéntica denominación en el "árbol genealógico".

6.6.2.2. Orden de Resolución de Métodos

El Orden de Resolución de Métodos o MRO (*Method Resolution Order*) es la manera en que un lenguaje de programación decide dónde buscar un método (o una propiedad) en una clase que hereda estos elementos de varias clases superiores.

La importancia del MRO se hace patente en presencia de la herencia múltiple donde, ante dos elementos con la misma denominación en clases distintas, es necesario definir qué método o propiedad prevalecerá en una instanciación.

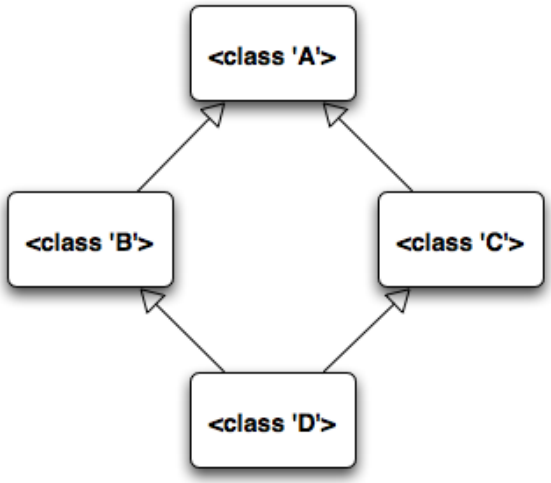

Figura 1: Herencia múltiple en forma de diamante.

En el caso de Python, la evolución del lenguaje dio como resultado dos algoritmos MRO distintos, uno simple para las clases de estilo antiguo y otro más sofisticado para las clases de estilo nuevo (las que heredan de `object`). Ambos algoritmos provienen de la teoría de grafos, y son los siguientes:

- **DFS – Depth First Search** (Búsqueda en profundidad): Recorre los nodos del grafo (árbol) de izquierda a derecha empezando de la raíz pero hasta el nodo más lejano. En el diagrama, empezando desde la clase D, el orden sería D, B, A, C.
- **BFS – Breadth First Search** (Búsqueda en anchura): Recorre los nodos del grafo efectuando barridos de izquierda a derecha. En el diagrama, empezando desde la clase D, el orden sería D, B, C, A.

```
# Este ejemplo es para Python 2

# Algoritmo: DFS - Depth First Search (Busqueda en profundidad)
class A: x = 'a'
class B(A): pass
class C(A): x = 'c'
class D(B, C): pass
print 'Viejo estilo: D.x = "%s"' % D.x

# Algoritmo: BFS - Breadth First Search (Busqueda en anchura)
class A(object): x = 'a'
class B(A): pass
class C(A): x = 'c'
class D(B, C): pass
print 'Nuevo estilo: D.x = "%s"' % D.x
```

Este es el resultado al ejecutar el programa en Python 2:

```
Viejo estilo: D.x = "a"
Nuevo estilo: D.x = "c"
```

El MRO puede obtenerse mediante mecanismos de **introspección**, como puede verse en la siguiente secuencia de sentencias ejecutadas durante una sesión con el intérprete interactivo de Python.

Empecemos creando dos clases vacías, únicamente a modo de ejemplo, haciendo uso de la sentencia comodín pass:

```
>>> class Animal(object): pass
...
>>> class Perro(Animal): pass
...
```

Acabamos de definir dos clases: `Animal` (de tipo `object`) y `Perro`, que hereda los métodos y propiedades de `Animal`. Ambas clases aparentemente son iguales, a ninguna de las dos se les definieron métodos o propiedades específicas, pero se diferencian en cuanto a la herencia – una deriva de la otra.

A continuación, creamos una instancia de la clase `Perro`, a la que llamamos `fido`.

```
>>> fido = Perro()
```

Mediante el método `__class__` y el operador `is` podemos ver que la clase del objeto `fido` es `Perro`, no `Animal`, aunque "desciende" de ella.

```
>>> fido.__class__ is animal
False
>>> fido.__class__ is perro
True
```

Sin embargo, con la función `isinstance`, vemos que `fido` es una instancia de la clase `Animal`, al igual que de la sub-clase `Perro`.

```
>>> isinstance(fido, Animal)
True
>>> isinstance(fido, Perro)
True
```

Por último, el método `__mro__` nos devuelve una **tupla** en la que podemos ver el orden de resolución de métodos:

```
>>> Animal.__mro__
(<class '__main__.Animal'>, <type 'object'>)
>>> Perro.__mro__
(<class '__main__.Perro'>, <class '__main__.Animal'>, <type 'object'>)
```

Los métodos y propiedades de una clase se superpondrán a los de la clase superior en el orden que aparecen en la tupla.

6.6.3. Polimorfismo

Consiste en definir comportamientos diferentes basados en una misma denominación, pero asociados a objetos distintos entre sí. Al llamarlos por ese nombre común, se utilizará el adecuado al objeto que se esté invocando.

```
a = Persona("Pablo")                # Nace Pablo
b = Persona("Juan")                 # Nace Juan
print(b)                            # Juan dice "Hola"
c = Empleado("Esteban", "Chofer")   # Nace Esteban
                                    # Esteban es Chofer
```

En el siguiente ejemplo, las dos clases derivadas de Animal comparten el método sonido, pero cada una le agrega su particularidad.

```
class Animal:

    cantidad = 0

    def __init__(self):
        print("Hola, soy un animal.")
        self.nombre = ""
        Animal.cantidad += 1
        print("Hay", Animal.cantidad, "animales.")

    def sonido(self):
        print("Este es mi sonido:")

    def __del__(self):
        print(self.nombre, "dice: Adios!")

class Perro(Animal):

    def __init__(self, nombre):
        Animal.__init__(self)
        print("Soy un perro.")
        self.nombre = nombre
        print("Me llamo", self.nombre)

    def sonido(self):
        Animal.sonido(self)
        print("Guau!")

class Gato(Animal):

    def __init__(self, nombre):
        Animal.__init__(self)
        print("Soy un gato.")
        self.nombre = nombre
        print("Me llamo", self.nombre)

    def sonido(self):
        Animal.sonido(self)
        print("Miau!")

fido = Perro("Fido")
fido.sonido()
tom = Gato("Tom")
tom.sonido()
```

Los objetos `fido` y `tom` descienden de `Animal` pero cada uno "emite su propio sonido".

Una subclase suele necesitar llamar al constructor de la superclase:

```
class Subclase(Superclase):
    def __init__(self):
        # Aquí la subclase hace sus cosas
        Superclase.__init__(self)
        # Aquí la subclase sigue haciendo sus cosas
```

Para no tener que nombrar a la superclase, puede usarse la función `super`:

```
super(Subclase, self).__init__()
```

En Python 3, es posible ahorrarse un poco de código escribiendo simplemente:

```
super().__init__()
```

Ejemplo en ambas versiones:

```
# Python 2.x

class A(object):
    def __init__(self):
        print "Mundo"

class B(A):
    def __init__(self):
        print "Hola"
        super(B, self).__init__()

# Python 3.x

class A:
    def __init__(self):
        print("Mundo")

class B(A):
    def __init__(self):
        print("Hola")
        super().__init__()
```

Al crear una instancia de la clase `B`:

```
b = B()
```

El resultado es:

```
Hola
Mundo
```

7. Referencias

- El Tutorial de Python 3 - PDF
- Aprenda a Pensar como un Programador con Python - PDF
- Inmersión en Python 3 - PDF

Favor enviar sugerencias y correcciones a czayas@pol.una.py