Talento Digital para Chile:

Módulo 3
Programación Avanzada en Python

MÓDULO 3 – Programación en Python

3.1.- Contenido 1: Describe los conceptos fundamentales del paradigma de orientación a objetos haciendo la distinción entre el concepto de Clase y de Objeto

Objetivo de la jornada

1. Distingue las características de la programación orientada a objetos respecto a la programación estructurada
2. Describe el concepto de clase haciendo la distinción con el concepto de objeto
3. Describe el concepto de atributo de una clase haciendo la distinción con el concepto de estado de un objeto
4. Describe el concepto de método de una clase haciendo la distinción con el concepto de comportamiento de un objeto
5. Describe los principios de abstracción y encapsulamiento de acuerdo al paradigma de orientación a objetos

3.1.1.- Paradigma de Orientación a Objetos

3.1.1.1. La Orientación a Objetos

Qué es la orientación a objetos

Todos saben qué es un objeto: un "algo" tangible que podemos sentir, sentir y manipular. Los primeros objetos con los que interactuamos suelen ser juguetes para bebés. Los bloques de madera, las formas plásticas y las piezas de rompecabezas de gran tamaño son los primeros objetos comunes. Los bebés aprenden rápidamente que ciertos objetos hacen ciertas cosas. Los triángulos encajan en agujeros en forma de triángulo. Suenan las campanas, se presionan los botones y se tiran de las palancas.

La definición de un objeto en el desarrollo de software no es tan diferente. Los objetos no son típicamente cosas tangibles que puedas captar, sentir o sentir, pero son modelos de cosas que pueden hacer ciertas cosas y que se les hagan ciertas

cosas. Formalmente, un objeto es una colección de datos y comportamientos asociados.

Entonces, sabiendo qué es un objeto, ¿qué significa orientado a objetos? Orientado simplemente significa dirigido hacia. Por lo tanto, orientado a objetos simplemente significa "funcionalmente dirigido a modelar objetos". Es una de las muchas técnicas utilizadas para modelar sistemas complejos al describir una colección de objetos que interactúan a través de sus datos y comportamiento.

De hecho, el análisis, el diseño y la programación son todas etapas del desarrollo de software. Llamarlos orientados a objetos simplemente especifica qué estilo de desarrollo de software se está buscando.

Importancia de la orientación a objetos en la programación

La programación orientada a objetos es uno de los enfoques más efectivos para escribir código eficiente. La programación orientada a objetos escribe clases que representan cosas del mundo real y situaciones, y crea objetos basados en estas clases. Cuando escribe una clase, define el comportamiento general que puede tener toda una categoría de objetos.

Cuando se crean objetos individuales de la clase, cada objeto se equipa automáticamente con el comportamiento general; A continuación, puede darle a cada objeto los rasgos únicos que desee. Es sorprendente lo bien que se pueden modelar situaciones del mundo real con programación orientada a objetos.

Diferencias con programación estructurada

Hasta ahora hemos visto ejemplos de programación estructurada, es decir: "Los programas (grandes) se dividen en programas pequeños llamados funciones".
En el siguiente ejemplo tenemos dos funciones, la clave es notar que una llama a la otra para que obtengamos el resultado deseado:

La función de abajo cuenta el número de caracteres de cada palabra del arreglo palabras y crea un arreglo con tales números.

```
def extraer_propiedad(prop):
    palabras = ['El', 'perro', 'llevó', 'el', 'diario', 'a', 'juan']
    return [prop(palabra) for palabra in palabras]
extraer_propiedad(len)
```

Resultado:

```
[2, 5, 5, 2, 6, 1, 4]
```

Por otro lado, la siguiente función imprime la última letra de una palabra dada:

```
def letra_final(palabra):
    return palabra[-1]
palabra = "muffin"
print(letra_final(palabra))
```

Resultado (la última letra de la palabra muffin, 'n' en este caso)

```
n
```

Ahora bien, si estructuramos nuestro código, podemos combinar ambas funciones:

```
extraer_propiedad(last_letter)
```

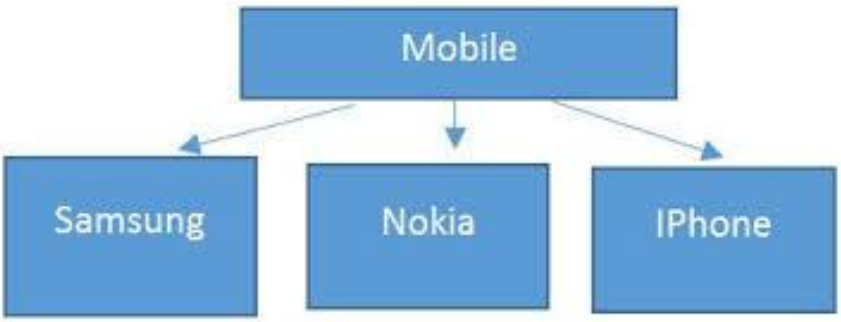
Resultado:

```
['l', 'o', 'ó', 'l', 'o', 'a', 'n']
```

Naturalmente podríamos utilizar sólo una función para ambas tareas, pero la idea es separar el código en pequeños fragmentos que luego pueden ser utilizados incluso por otras funciones.

Orientación a objetos aplicada a la vida cotidiana

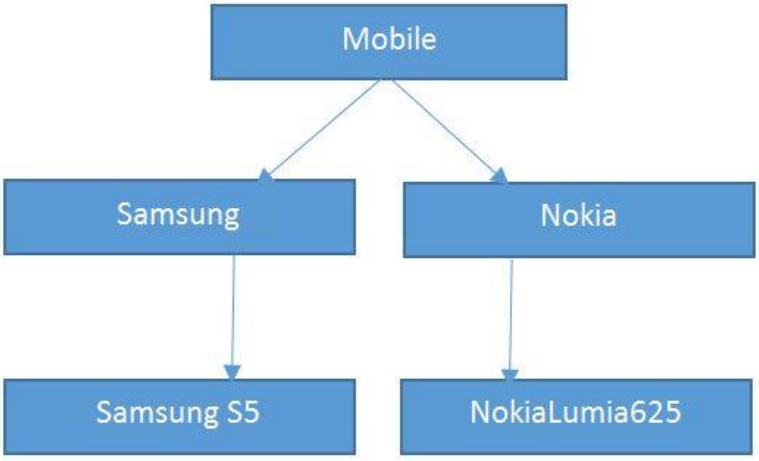
Con un simple ejemplo podemos ver como la orientación a objetos trabaja a un nivel abstracto. Cuando pensamos en un teléfono móvil como un objeto, su funcionalidad básica para la que fue inventado era llamar y recibir una llamada y mensajería. Pero hoy en día se agregaron miles de nuevas características y modelos y las características y la cantidad de modelos siguen creciendo.



En el diagrama

anterior, cada marca (Samsung, Nokia, IPhone) tiene su propia lista de funciones junto con la funcionalidad básica de marcación, recepción de llamadas y mensajería. No obstante, todos son objetos Mobile.

De la misma forma podríamos extender la abstracción un nivel más y/o incorporar nuevos modelos/marcas, etc.



3.1.1.2. Las Clases

Qué es una Clase

Una clase en OOP es un "plano" que describe el objeto. Lo llamamos un plano de cómo se debe representar el objeto. Básicamente, una clase constaría de un nombre, atributos y operaciones. Teniendo en cuenta el ejemplo anterior, Mobile puede ser una clase, que tiene algunos atributos como Tipo de perfil, Número IMEI, Procesador y algunos más. Puede tener operaciones como Marcar, Recibir y Enviar Mensaje.

Qué es un Objeto

Un objeto, en la programación orientada a objetos (OOP), es un tipo de datos abstracto creado por un desarrollador. Puede incluir múltiples propiedades y métodos e incluso puede contener otros objetos. En la mayoría de los lenguajes de programación, los objetos se definen como clases. Los objetos proporcionan un enfoque estructurado de la programación. Lo importante es tener claro que un **objeto se refiere a una** *instancia* **particular de una clase**, donde el objeto puede ser una combinación de variables, funciones y estructuras de datos.

Diferencia entre clase, instancia y objeto

Una clase es una plantilla de código para crear objetos con ciertas propiedades y métodos (en el contexto de programación orientada a objetos se utiliza el término método en lugar de función). Dicha plantilla se utiliza para crear uno o muchos objetos. Cada uno de los objetos creados de denomina instancia.

Para ilustrar estos conceptos vamos a utilizar un poco de código, quizás a esta altura le parezca un poco extraño pero los detalles de la estructura de este código se explicarán más adelante con mayor profundidad, por ahora de a poco vamos aprendiendo la implementación práctica de estos conceptos.

Supongamos que declaramos una clase Perro:

```
class Perro:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
```

Sin hurgar mayormente en los detalles vemos que tenemos una clase Perro (la plantilla) y dicha plantilla tiene 2 atributos: nombre y edad. Es decir, todo objeto de la clase Perro necesitará tener un valor para nombre y edad (pese a que se podrían omitir o predefinir en circunstancias especiales, pero eso no lo veremos aún).

Ahora usaremos la plantilla `Perro` para crear dos objetos o instancias `perro`.

```
boby = Perro('boby', 10)
duque= Perro('duque', 3)
```

boby y duque son dos perros (objetos) distintos, es decir, dos instancias de la clase Perro y por ende comparten atributos en su calidad de perros.

Inspeccionemos estas instancias/objetos utilizando el operador punto (.) para acceder a los atributos:

```
boby.nombre
```

Resulta en

```
'boby'
```

```
duque.edad
```

Resulta en

```
3
```

Puede resultar que duque haya envejecido respecto a nuestros registros, pero lo podemos actualizar fácilmente.

```
duque.edad = 12
```

Ahora

```
duque.edad
```

Resulta en

```
12
```

3.1.1.3. Los Atributos

Ahora tenemos una comprensión de alguna terminología básica orientada a objetos. Los objetos son instancias de clases que se pueden asociar entre sí. Una instancia de objeto es un objeto específico con su propio conjunto de datos y comportamientos; se dice que una naranja específica en la mesa frente a nosotros es una instancia de la clase general de naranjas. Eso es bastante simple, pero ¿cuáles son estos datos y comportamientos que están asociados con cada objeto?

Atributos de una clase

Empecemos por los datos. Los datos generalmente representan las características individuales de un determinado objeto. Una clase de objetos puede definir características específicas que son compartidas por todas las instancias de esa clase. Cada instancia puede tener diferentes valores de datos para las características dadas. Por ejemplo, si tenemos tres naranjas en la mesa (si no hemos comido ninguna) podrían tener un peso diferente cada una. La clase naranja podría tener un atributo de peso. Todas las instancias de la clase naranja tienen un atributo de peso, pero cada naranja puede tener un valor diferente para este peso. Sin embargo, los atributos no tienen que ser únicos; dos naranjas cualesquiera pueden pesar la misma cantidad. Como ejemplo más realista, dos objetos que representan a diferentes clientes pueden tener el mismo valor para su primer nombre.

Los atributos se denominan con frecuencia propiedades. Algunos autores sugieren que los dos términos tienen significados diferentes, generalmente los atributos se

pueden configurar, mientras que las propiedades son de solo lectura. En Python, el concepto de "solo lectura" no se usa realmente, por lo que en este curso veremos los dos términos usados indistintamente.

Estado de un objeto

El estado de un objeto puede verse como el conjunto de todos los atributos mutables. Considere una lampara simple. Puede estar encendida o apagada. Puede verlo como su estado. Pero un estado puede ser infinitamente complejo y estar formado por millones de atributos.

Diferencia entre atributo y estado

Los atributos son inmutables. Cuando crea un objeto, establece atributos. Y esos valores de atributo probablemente no cambien durante la vida útil del objeto.

Por ejemplo, considere el pseudocódigo siguiente.

```
class Perro {
  // Attributes
  color;
  edad;

  // Estado
  tieneSed;

 // Comportamiento
  correr();
}
```

El perro objeto aquí tiene dos atributos: color y edad. Cuando crea un objeto perro, define la edad y el color. En realidad, puede crear un objeto con el atributo de color establecido en negro. Entonces, lógicamente, está creando un perro negro. Y el perro permanece negro durante toda la vida del objeto. Si desea crear un perro rojo, es posible que deba crear otro objeto de perro con color rojo.

La forma de crear objetos con atributos inmutables varía de un lenguaje a lenguaje. El método típico es crear propiedades con función de conjunto privado.

En el ejemplo anterior, el color es un atributo. Eso no significa que el color deba ser un atributo en todos los casos. Cambiará según el contexto. Por ejemplo, considere el semáforo como un objeto, donde el color no es un atributo sino un estado.

Por otro lado, los estados son mutables. El estado de un objeto cambia varias veces a lo largo de su vida. El estado experimenta cambios ya sea por alguna función

aplicada en el objeto o por un evento fuera del objeto. Hay otros escenarios, pero estos son los casos típicos. Por ejemplo, si la función de correr() se llama con frecuencia en un objeto perro, el estado de sed puede cambiar a verdadero.

A veces, en programación, es posible que deba crear un objeto sin estado, sino que solo con atributos. Esos objetos son en realidad pasivos y no tienen ningún comportamiento. Literalmente representa un grupo de atributos relacionados lógicamente.

3.1.1.4. Los Métodos

Métodos de una clase

A nivel de programación, los métodos son como funciones en la programación estructurada, pero mágicamente tienen acceso a todos los datos asociados con ese objeto. Al igual que las funciones, los métodos también pueden aceptar parámetros y devolver valores.

Comencemos escribiendo una clase (Python 3) que contenga ejemplos simples para tres tipos de métodos (de clase, estáticos y regulares de instancia):

```
class MiClase:
    def metodo(self):
        return 'metodo de instancia llamado', self

    @classmethod
    def metodo_de_clase(cls):
        return 'metodo de clase llamado', cls

    @staticmethod
    Def metodo_estatico():
        return 'metodo estatico llamado'
```

El primer método en MiClase, llamado metodo, es un método de instancia regular. Ese es el tipo de método básico y sencillo que utilizará la mayor parte del tiempo. Puede ver que el método toma un parámetro, self, que apunta a una instancia de MiClase cuando se llama al método (pero, por supuesto, los métodos de instancia pueden aceptar más de un parámetro).

A través del parámetro self, los métodos de instancia pueden acceder libremente a atributos y otros métodos en el mismo objeto. Esto les da mucho poder cuando se trata de modificar el estado de un objeto.

No solo pueden modificar el estado del objeto, los métodos de instancia también pueden acceder a la clase misma a través del atributo self.__ class__. Esto significa que los métodos de instancia también pueden modificar el estado de la clase. Dado que la clase es solo una plantilla, self permite el acceso a los atributos y métodos de cada objeto en Python.

```
obj = MiClase()
obj.metodo()
```

Resultado

```
('metodo de instancia llamado', <__main__.MiClase at 0x10568b910>)
```

Esto confirma que el método (el método de instancia) tiene acceso a la instancia del objeto (impreso como <__main__.MiClase at 0x10568b910>) a través del argumento self.

Cuando se llama al método, Python reemplaza el argumento self con el objeto de instancia, obj. Podríamos ignorar el azúcar sintáctico* de la sintaxis de puntos (obj.method ()) y pasar el objeto de instancia manualmente para obtener el mismo resultado:

```
MiClase.metodo(obj)
```

Resultado

```
('metodo de instancia llamado', <__main__.MiClase at 0x10568b910>)
```

Nota*: El azúcar sintáctico es la sintaxis dentro de un lenguaje de programación que está diseñado para facilitar la lectura o la expresión.

¿Puede adivinar qué pasaría si intenta llamar al método sin crear primero una instancia?

Los métodos de instancia también pueden acceder a la propia clase a través del atributo self.__class__. Esto hace que los métodos de instancia sean poderosos en términos de restricciones de acceso: pueden modificar el estado en la instancia de objeto y en la propia clase.

Para ilustrar los métodos de instancia de la forma más simple veamos el siguiente fragmento:

```
class Perro:
    def ladrar(self):
        print("guau guau")
```

```
cachupin = Perro()
cachupin.ladrar()
```

Resultado:

```
guau guau
```

En la mayor parte de los casos usamos este tipo de métodos.

Métodos de clase

Comparemos lo anterior con el segundo método, MiClase.metodo_de_clase. Este método está marcado con un decorador @classmethod para marcarlo como un método de clase. En lugar de aceptar un parámetro self, los métodos de clase toman un parámetro cls que apunta a la clase, y no a la instancia del objeto, cuando se llama al método.

Debido a que el método de clase solo tiene acceso a este argumento cls, no puede modificar el estado de la instancia del objeto. Eso requeriría acceso a self. Sin embargo, los métodos de clase aún pueden modificar el estado de la clase que se aplica a todas las instancias de la clase.

Para ilustrar la diferencia comparemos un método de instancia con método de clase similar.

Método de instancia

```
class Inst:

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

    def presentar(self):
        print("Hola, Soy %s, y mi nombre es is %s" %(self, self.name))

mi_inst = Inst("Instancia de Prueba")
otra_inst = Inst("Otra Instancia")
mi_inst.presentar()
Otra_inst.presentar()
```

Resultado:

```
Hola, Soy <__main__.Inst object at 0x1070c9b50>, y mi nombre es
Instancia de Prueba
Hola, Soy <__main__.Inst object at 0x1067e5610>, y mi nombre es Otra
Instancia
```

Algo a tomar en cuenta para entender mejor estos conceptos tiene que ver con el azúcar sintáctico:

En el ejemplo anterior sin azúcar sintáctico podríamos haber escrito:

```
Inst.presentar(mi_inst)
```

Método de clase:

```
class Cls:

    @classmethod
    def presentar(cls):
        print("Hola, Soy %s!" %cls)

cls.presentar()
```

Resultado

```
Hola, Soy <class '__main__.cls'>!
```

La idea del método de clase es muy similar al método de instancia, con la única diferencia de que, en lugar de pasar la instancia de forma oculta como primer parámetro, ahora estamos pasando la clase en sí como primer parámetro.

Métodos estáticos

El tercer método, MiClase.metodo_estatico fue marcado con un decorador @staticmethod para marcarlo como un método estático. Este tipo de método no toma un parámetro self ni cls (pero, por supuesto, es libre de aceptar un número arbitrario de otros parámetros).

Ya que hemos descrito los tres tipos de métodos teóricamente veamos una explicación práctica.

Un método estático no puede modificar el estado del objeto ni el estado de la clase. Los métodos estáticos están restringidos en cuanto a los datos a los que pueden acceder, y son principalmente una forma de asignar un espacio de nombres (namespace) a sus métodos. Eso lo veremos en el siguiente apartado, se menciona solo para ilustrar el parecido entre métodos de clase y métodos estáticos. MiClase se configuró de tal manera que la implementación de cada método devuelve una tupla que contiene información para que podamos rastrear lo que está sucediendo y a qué partes de la clase u objeto puede acceder el método.

Este ejemplo incluye los tres tipos de métodos que hemos estado discutiendo

```
from datetime import date

class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    # un metodo de clase para crear un objeto de clase
    # Persona por año de nacimiento.
    @classmethod
    def desde_agno_nacimiento(cls, nombre, agno):
        return cls(nombre, date.today().year - agno)

    # un metodo estatico para revisar si una persona es adulta o no.
    @staticmethod
    def es_adulto(edad):
        return edad > 18

persona1 = Persona('marcela', 21)
persona2 = Persona.desde_agno_nacimiento('marcela', 1999)

print(person1.age)
print(person2.age)

# imprimir el resultado
print(Person.es_adulto(22))
```

Resultado

```
21
21
True
```

Note que persona1 es inicializado (crear instancia) de la forma más común, con un método de instancia, en este caso reservado __init__. persona2 utiliza el método de clase para instanciarse, lo cual es un uso común en factories (constructores alternativos) tal concepto escapa al material de este capítulo. El método estático no requiere instancia, simplemente se llama directamente como si fuera una especie de función Persona.is_adulto(22)

Comportamiento de un objeto

Los comportamientos son acciones que pueden ocurrir en un objeto. Los comportamientos que se pueden realizar en una clase específica de objetos se denominan métodos (ver sección anterior).

Diferencia entre método y comportamiento

El comportamiento de un objeto está definido por sus métodos, que como hemos visto son los métodos y subrutinas definidas dentro de la clase de objeto.

3.1.1.5. Principios Básicos

Abstracción

Abstracción significa mostrar solo información esencial y ocultar los detalles. La abstracción de datos se refiere a proporcionar solo información esencial sobre los datos al mundo exterior, ocultando los detalles de fondo o la implementación.

Considere un ejemplo de la vida real de un hombre que conduce un automóvil. El hombre solo sabe que presionar el acelerador aumentará la velocidad del automóvil o aplicar los frenos detendrá el automóvil, pero no sabe cómo al presionar el acelerador la velocidad realmente aumenta, no sabe sobre el mecanismo interno del automóvil o el Implementación de acelerador, frenos, etc. en el automóvil. Eso es la abstracción.

La abstracción en Python se logra mediante el uso de clases e interfaces abstractas.

Una clase abstracta es una clase que generalmente proporciona una funcionalidad incompleta y contiene uno o más métodos abstractos. Los métodos abstractos son los métodos que generalmente no tienen ninguna implementación, se deja a las subclases proporcionar la implementación de los métodos abstractos.

Una **interfaz** debería proporcionar los nombres de los métodos sin los cuerpos de estos. Las subclases deben proporcionar implementación para todos los métodos definidos en una interfaz. Tenga en cuenta que en Python (a diferencia de otros lenguajes) no hay soporte para crear interfaces explícitamente, tendrá que usar la clase abstracta. En Python puede crear una interfaz usando la clase abstracta, si crea una clase abstracta que contiene solo métodos abstractos que actúa como una interfaz en Python.

En el ejemplo hay una clase abstracta llamada Pago que tiene un método abstracto pago(). - con minúscula. Hay dos clases secundarias PagoTarjetaCredito y PagoBilleteraMovil derivadas de la clase Pago que implementan el método abstracto pago() según su funcionalidad.

Como usuarios, nos abstraemos de esa implementación cuando se crea un objeto de PagoTarjetaCredito y se invoca el método pago() utilizando ese objeto, para nosotros simplemente se invoca el método pago de la clase PagoTarjetaCredito. Cuando se

crea un objeto PagoBilleteraMovil y se invoca el método pago() utilizando ese objeto, se invoca el método pago de la clase PagoBilleteraMovil.

```
from abc import ABC, abstractmethod

class Pago(ABC):

  def imprimir(self, monto):
    print('Monto de compra- ', monto)

  @abstractmethod
  def pago(self, monto):
    pass

class PagoTarjetaCredito(Pago):

  def pago(self, monto):
    print('Pago de tarjeta de credito por- ', monto)

class PagoBilleteraMovil(Pago):

  def pago(self, monto):
    print('Pago de billetera movil por- ', monto)


obj = PagoTarjetaCredito()
obj.pago(100)
obj.imprimir(100)
print(isinstance(obj, Pago))
obj = PagoBilleteraMovil()
obj.pago(200)
obj.imprimit(200)
print(isinstance(obj, Pago))
```

Resultado

```
Pago de tarjeta de credito por-  100
Monto de compra- 100
True
Pago de billetera movil por-  200
Monto de compra- 200
True
```

Utilizamos la librería abc para tener acceso a las Clases Base Abstractas - (Abstract Base Classes) https://docs.python.org/3/library/abc.html

Encapsulamiento

Encapsulamiento es otro de los cuatro conceptos fundamentales de la programación orientada a objetos (siendo los otros abstracción, herencia y polimorfismo. Estos últimos dos serán vistos más adelante).

La idea general de este mecanismo es simple. Si tiene un atributo que no es visible desde el exterior de un objeto y lo agrupa con métodos que le brindan acceso de lectura o escritura, entonces puede ocultar información específica y controlar el acceso al estado interno del objeto.

Python no tiene la palabra clave private, a diferencia de otros lenguajes orientados a objetos, pero se puede encapsular.

En su lugar, se basa en la convención: una variable de clase a la que no se debe acceder directamente debe ir precedida de un guión bajo.

```
class Robot(object):
  def __init__(self):
    self.a = 123
    self._b = 456
    self.__c = 789

obj = Robot()
print(obj.a)
print(obj._b)
print(obj.__c)
```

Resultado:

```
123
456

AttributeError Traceback (most recent call last)
...
---> 10 print(obj.__c)
AttributeError: 'Robot' object has no attribute '__c'
```

Entonces, ¿qué pasa con los guiones bajos y el error?

Un solo guión bajo: Variable privada, no se debe acceder directamente. Pero nada le impide hacer eso (excepto las convenciones).

Un guión bajo doble: variable privada, más difícil de acceder, pero aún posible.

Ambos siguen siendo accesibles: Python tiene variables privadas por convención.

Getters y setters

Las variables privadas están diseñadas para cambiarse utilizando métodos getter y setter. Estos proporcionan acceso indirecto a ellas:

```
class Robot(object):
   def __init__(self):
      self.__version = 22

   def getVersion(self):
      print(self.__version)

   def setVersion(self, version):
      self.__version = version

obj = Robot()
obj.getVersion()
obj.setVersion(23)
obj.getVersion()
```

Resultado:

```
22
23
```

3.1.2. Referencias.

[1] Mark Lutz, Python Pocket Reference, Fifth Edition 2014.

[2] Matt Harrison, Ilustrated Guide to Python3, 2017.

[3] Eric Matthes, Python Crash Course, 2016.

[4] Magnus Lie Hetland, Beginning Python: From Novice to Professional, 2017.

[5] Python Tutorial.
https://www.w3schools.com/python/