Módulo 3 - Programación Orientada a Objetos - Métodos

Métodos a detalle

Resumamos todos los hechos relacionados con el uso de métodos en las clases de Python.

Como ya sabes, un **método es una función que está dentro de una clase**.

Existe un requisito fundamental: un **método está obligado a tener al menos un parámetro** (no existen métodos sin parámetros; un método puede invocarse sin un argumento, pero no puede declararse sin parámetros).

El primer (o único) parámetro generalmente se denomina *self*. Te sugerimos que lo sigas nombrando de esta manera, darle otros nombres puede causar sorpresas inesperadas.

El nombre self sugiere el propósito del parámetro: **identifica el objeto para el cual se invoca el método**.

Si vas a invocar un método, no debes pasar el argumento para el parámetro self, Python lo configurará por ti.

```
class Classy:
    def method(self):
        print("método")


obj = Classy()
obj.method()
```

El código da como salida:

```
método
```

Toma en cuenta la forma en que hemos creado el objeto, **hemos tratado el nombre de la clase como una función**, y devuelve un objeto recién instanciado de la clase.

Si deseas que el método acepte parámetros distintos a self, debes:

- Colocarlos después de self en la definición del método.
- Pasarlos como argumentos durante la invocación sin especificar self.

Justo como aqui:

```
class Classy:
    def method(self, par):
        print("método:", par)


obj = Classy()
obj.method(1)
obj.method(2)
obj.method(3)
```

El código da como salida:

```
método: 1
método: 2
método: 3
```

El parámetro *self* es usado para obtener acceso a la instancia del objeto y las variables de clase.

El ejemplo muestra ambas formas de utilizar el parámetro self:

```
class Classy:
    varia = 2
    def method(self):
        print(self.varia, self.var)


obj = Classy()
obj.var = 3
obj.method()
```

El código da como salida:

```
2 3
```

El parámetro *self* también se usa para invocar otros métodos desde dentro de la clase.

Justo como aquí:

```
class Classy:
    def other(self):
        print("otro")

    def method(self):
        print("método")
        self.other()


obj = Classy()
obj.method()
```

El código da como salida:

```
método
otro
```

Si se nombra un método de esta manera: __init__, no será un método regular, será un **constructor**.

Si una clase tiene un constructor, este se invoca automática e implícitamente cuando se instancia el objeto de la clase.

El constructor:

- Esta **obligado a tener el parámetro *self*** (se configura automáticamente).
- **Pudiera (pero no necesariamente) tener mas parámetros** que solo *self*; si esto sucede, la forma en que se usa el nombre de la clase para crear el objeto debe tener la definición __init__.
- **Se puede utilizar para configurar el objeto**, es decir, inicializa adecuadamente su estado interno,

crea variables de instancia, crea instancias de cualquier otro objeto si es necesario, etc.

El ejemplo muestra un constructor muy simple pero funcional.

```
class Classy:
    def __init__(self, value):
        self.var = value


obj_1 = Classy("objeto")

print(obj_1.var)
```

Ejecútalo. El código da como salida:

```
objeto
```

Ten en cuenta que el constructor:

- **No puede retornar un valor**, ya que está diseñado para devolver un objeto recién creado y nada más.
- **No se puede invocar directamente desde el objeto o desde dentro de la clase** (puedes invocar un constructor desde cualquiera de las superclases del objeto, pero discutiremos esto más adelante).

Como __init__ es un método, y un método es una función, puedes hacer los mismos trucos con constructores y métodos que con las funciones ordinarias.

El ejemplo en el editor muestra cómo definir un constructor con un valor de argumento predeterminado. Pruébalo.

```
class Classy:
    def __init__(self, value = None):
        self.var = value


obj_1 = Classy("objeto")
obj_2 = Classy()

print(obj_1.var)
print(obj_2.var)
```

El código da como salida:

```
objeto
None
```

Todo lo que hemos dicho sobre el manejo de los nombres también se aplica a los nombres de métodos, un método cuyo nombre comienza con __ está (parcialmente) oculto.

El ejemplo muestra este efecto:

```
class Classy:
    def visible(self):
        print("visible")

    def __hidden(self):
        print("oculto")
```

```
obj = Classy()
obj.visible()

try:
    obj.__hidden()
except:
    print("fallido")

obj._Classy__hidden()
```

El código da como salida:

```
visible
fallido
oculto
```

La vida interna de clases y objetos

Cada clase de Python y cada objeto de Python está pre-equipado con un conjunto de atributos útiles que pueden usarse para examinar sus capacidades.

Ya conoces uno de estos: es la propiedad __dict__.

Observemos como esta propiedad trata con los métodos

```
class Classy:
    varia = 1
    def __init__(self):
        self.var = 2

    def method(self):
        pass

    def __hidden(self):
        pass


obj = Classy()

print(obj.__dict__)
print(Classy.__dict__)
```

Ejecútalo para ver que produce. Verifica el resultado.

```
{'var': 2}
{'__module__': '__main__', 'varia': 1, '__init__': <function Classy.__init__ at
0x7fcb0ae8c320>, 'method': <function Classy.method at 0x7fcb0ae8c3b0>,
'_Classy__hidden': <function Classy.__hidden at 0x7fcb0ae8c440>, '__dict__':
<attribute '__dict__' of 'Classy' objects>, '__weakref__': <attribute '__weakref__'
of 'Classy' objects>, '__doc__': None}
```

Encuentra todos los métodos y atributos definidos. Localiza el contexto en el que existen: dentro del objeto o dentro de la clase.

__dict__ es un diccionario. Otra propiedad incorporada que vale la pena mencionar es una cadena llamada __name__.

La propiedad contiene el **nombre de la clase**. No es nada emocionante, es solo una cadena.

Nota: el atributo __name__ está ausente del objeto, **existe solo dentro de las clases**.

Si deseas **encontrar la clase de un objeto en particular**, puedes usar una función llamada `type()`, la cual es capaz (entre otras cosas) de encontrar una clase que se haya utilizado para crear instancias de cualquier objeto.

Observa el código en el editor, ejecútalo y compruébalo tu mismo.

```
class Classy:
    pass


print(Classy.__name__)
obj = Classy()
print(type(obj).__name__)
```

La salida del código es:

```
Classy
Classy
```

Nota: algo como esto

```
print(obj.__name__)
```

causará un error.

__module__ es una cadena, también a**lmacena el nombre del módulo que contiene la definición de la clase**.

Vamos a comprobarlo: ejecuta el código en el editor.

La salida del código es:

```
__main__
__main__
```

Como sabes, cualquier módulo llamado __main__ en realidad no es un módulo, sino es el **archivo actualmente en ejecución**.

__bases__ es una tupla. La **tupla contiene clases** (no nombres de clases) que son superclases directas de la clase.

El orden es el mismo que el utilizado dentro de la definición de clase.

Te mostraremos solo un ejemplo muy básico, ya que queremos resaltar **cómo funciona la herencia**.

Además, te mostraremos cómo usar este atributo cuando discutamos los aspectos orientados a objetos de las excepciones.

Nota: **solo las clases tienen este atributo**, los objetos no.

Hemos definido una función llamada printBases(), diseñada para presentar claramente el contenido de la tupla.

```
class SuperOne:
    pass


class SuperTwo:
    pass


class Sub(SuperOne, SuperTwo):
    pass


def printBases(cls):
    print('( ', end='')

    for x in cls.__bases__:
        print(x.__name__, end=' ')
    print(')')


printBases(SuperOne)
printBases(SuperTwo)
printBases(Sub)
```

Su salida es:

```
( object )
( object )
( SuperOne SuperTwo )
```

Nota: **una clase sin superclases explícitas apunta a object** (una clase de Python predefinida) como su antecesor directo.

Reflexión e introspección

Todo esto permite que el programador de Python realice dos actividades importantes específicas para muchos lenguajes objetivos. Las cuales son:

- **Introspección**, que es la capacidad de un programa para examinar el tipo o las propiedades de un objeto en tiempo de ejecución.
- **Reflexión**, que va un paso más allá, y es la capacidad de un programa para manipular los valores, propiedades y/o funciones de un objeto en tiempo de ejecución.

En otras palabras, no tienes que conocer la definición completa de clase/objeto para manipular el objeto, ya que el objeto y/o su clase contienen los metadatos que te permiten reconocer sus características durante la ejecución del programa.

Investigando Clases

¿Qué puedes descubrir acerca de las clases en Python? La respuesta es simple: todo.

Tanto la reflexión como la introspección permiten al programador hacer cualquier cosa con cada objeto, sin importar de dónde provenga.

```
class MyClass:
    pass


obj = MyClass()
obj.a = 1
obj.b = 2
obj.i = 3
obj.ireal = 3.5
obj.integer = 4
obj.z = 5


def incIntsI(obj):
    for name in obj.__dict__.keys():
        if name.startswith('i'):
            val = getattr(obj, name)
            if isinstance(val, int):
                setattr(obj, name, val + 1)


print(obj.__dict__)
incIntsI(obj)
print(obj.__dict__)
```

La función llamada `incIntsI()` toma un objeto de cualquier clase, escanea su contenido para encontrar todos los atributos enteros con nombres que comienzan con i, y los incrementa en uno.

¿Imposible? ¡De ninguna manera!

Así es como funciona:

- La línea 1: define una clase muy simple…
- Las líneas 3 a la 10: … la llenan con algunos atributos.
- La línea 14: ¡esta es nuestra función!
- La línea 15: escanea el atributo __dict__, buscando todos los nombres de atributos.
- La línea 16: si un nombre comienza con i…
- La línea 17: … utiliza la función `getattr()` para obtener su valor actual; nota: `getattr()` toma dos argumentos: un objeto y su nombre de propiedad (como una cadena) y devuelve el valor del atributo actual.
- La línea 18: comprueba si el valor es de tipo entero, emplea la función `isinstance()` para este propósito (discutiremos esto más adelante).
- La línea 19: si la comprobación sale bien, incrementa el valor de la propiedad haciendo uso de la función `setattr()`; la función toma tres argumentos: un objeto, el nombre de la propiedad (como una cadena) y el nuevo valor de la propiedad.

El código da como salida:

```
{'a': 1, 'integer': 4, 'b': 2, 'i': 3, 'z': 5, 'ireal': 3.5}
{'a': 1, 'integer': 5, 'b': 2, 'i': 4, 'z': 5, 'ireal': 3.5}
```

Puntos Clave

1. Un método es una función dentro de una clase. El primer (o único) parámetro de cada método se suele llamar *self*, que está diseñado para identificar al objeto para el que se invoca el método con el fin de acceder a las propiedades del objeto o invocar sus métodos.

2. Si una clase contiene un **constructor** (un método llamado __init__), este no puede devolver ningún valor y no se puede invocar directamente.

3. Todas las clases (pero no los objetos) contienen una propiedad llamada __name__, que almacena el nombre de la clase. Además, una propiedad llamada __module__ almacena el nombre del módulo en el que se ha declarado la clase, mientras que la propiedad llamada __bases__ es una tupla que contiene las superclases de una clase.

Por ejemplo:

```
class Sample:
    def __init__(self):
        self.name = Sample.__name__
    def myself(self):
        print("Mi nombre es " + self.name + " y vivo en " + Sample.__module__)


obj = Sample()
obj.myself()
```

El código da como salida:

```
Mi nombre es Sample y vivo en __main__
```

ejercicio

Objetivos

- Mejorar las habilidades del estudiante para definir clases desde cero.
- Definir y usar variables de instancia.
- Definir y usar métodos.

Escenario

Necesitamos una clase capaz de contar segundos. ¿Fácil? No es tan fácil como podrías pensar, ya que tendremos algunos requisitos específicos.

Léelos con atención, ya que la clase sobre la que escribes se utilizará para lanzar cohetes en misiones

internacionales a Marte. Es una gran responsabilidad. ¡Contamos contigo!

Tu clase se llamará *Timer* (temporizador en español). Su constructor acepta tres argumentos que representan **horas** (un valor del rango [0..23]; usaremos tiempo militar), **minutos** (del rango [0. .59]) y **segundos** (del rango [0..59]).

Cero es el valor predeterminado para todos los parámetros anteriores. No es necesario realizar ninguna comprobación de validación.

La clase en sí debería proporcionar las siguientes facilidades:

- Los objetos de la clase deben ser «imprimibles», es decir, deben poder convertirse implícitamente en cadenas de la siguiente forma: «hh:mm:ss», con ceros a la izquierda agregados cuando cualquiera de los valores es menor que 10.
- La clase debe estar equipada con métodos sin parámetros llamados next_second() y previous_second (), incrementando el tiempo almacenado dentro de los objetos en +1/-1 segundos respectivamente.

Emplea las siguientes sugerencias:

- Todas las propiedades del objeto deben ser privadas.
- Considera escribir una función separada (¡no un método!) para formatear la cadena con el tiempo.

```
class Timer:
    def __init__( ??? ):
        #
        # Escribir código aquí.
        #

    def __str__(self):
        #
        # Escribir código aquí.
        #

    def next_second(self):
        #
        # Escribir código aquí.
        #

    def prev_second(self):
        #
        # Escribir código aquí.
        #


timer = Timer(23, 59, 59)
print(timer)
timer.next_second()
print(timer)
timer.prev_second()
print(timer)
```

Ejecuta tu código y comprueba si el resultado es el mismo que el nuestro.

Salida Esperada

```
23:59:59
00:00:00
```

```
23:59:59
```

ejercicio

Objetivos

- Mejorar las habilidades del estudiante para definir clases desde cero.
- Definir y usar variables de instancia.
- Definir y usar métodos.

Escenario

Tu tarea es implementar una clase llamada Weeker. Sí, tus ojos no te engañan, este nombre proviene del hecho de que los objetos de esta clase podrán almacenar y manipular los días de la semana.

El constructor de la clase acepta un argumento: una cadena. La cadena representa el nombre del día de la semana y los únicos valores aceptables deben provenir del siguiente conjunto:

```
Lun Mar Mie Jue Vie Sab Dom
```

Invocar al constructor con un argumento desde fuera de este conjunto debería generar la excepción WeekDayError (defínela tu mismo; no te preocupes, pronto hablaremos sobre la naturaleza objetiva de las excepciones). La clase debe proporcionar las siguientes facilidades:

- Los objetos de la clase deben ser «imprimibles», es decir, deben poder convertirse implícitamente en cadenas de la misma forma que los argumentos del constructor.
- La clase debe estar equipada con métodos de un parámetro llamados add_days(n) y subtract_days(n), siendo n un número entero que actualiza el día de la semana almacenado dentro del objeto mediante el número de días indicado, hacia adelante o hacia atrás.
- Todas las propiedades del objeto deben ser privadas.

Completa la plantilla que te proporcionamos en el editor, ejecuta su código y verifica si tu salida se ve igual que la nuestra.

```
class WeekDayError(Exception):
    pass


class Weeker:
    #
    # Escribir código aquí.
    #

    def __init__(self, day):
        #
        # Escribir código aquí.
        #

    def __str__(self):
        #
        # Escribir código aquí.
```

```
        #

    def add_days(self, n):
        #
        # Escribir código aquí.
        #

    def subtract_days(self, n):
        #
        # Escribir código aquí.
        #


try:
    weekday = Weeker('Lun')
    print(weekday)
    weekday.add_days(15)
    print(weekday)
    weekday.subtract_days(23)
    print(weekday)
    weekday = Weeker('Lun')
except WeekDayError:
    print("Lo siento, no puedo atender tu solicitud.")
```

Salida Esperada

```
Lun
Mar
Dom
Lo siento, no puedo atender tu solicitud.
```

Objetivos

- Mejorar las habilidades del estudiante para definir clases desde cero.
- Definir y usar variables de instancia.
- Definir y usar métodos.

Escenario

Visitemos un lugar muy especial: un plano con el sistema de coordenadas cartesianas (puedes obtener más información sobre este concepto aquí: https://en.wikipedia.org/wiki/Cartesian_coordinate_system).

Cada punto ubicado en el plano puede describirse como un par de coordenadas habitualmente llamadas x y y. Queremos que escribas una clase en Python que almacene ambas coordenadas como números flotantes. Además, queremos que los objetos de esta clase evalúen las distancias entre cualquiera de los dos puntos situados en el plano.

La tarea es bastante fácil si empleas la función denominada hypot() (disponible a través del módulo math) que evalúa la longitud de la hipotenusa de un triángulo rectángulo (más detalles aquí: https://en.wikipedia.org/wiki/Hypotenuse) y aquí: https://docs.python.org/3.7/library/math.html#trigonometric-functions.

Así es como imaginamos la clase:

- Se llama Point.
- Su constructor acepta dos argumentos (x y y respectivamente), ambos por defecto se igualan a cero.
- Todas las propiedades deben ser privadas.
- La clase contiene dos métodos sin parámetros llamados getx() y gety(), que devuelven cada una de las dos coordenadas (las coordenadas se almacenan de forma privada, por lo que no se puede acceder a ellas directamente desde el objeto).
- La clase proporciona un método llamado distance_from_xy(x,y), que calcula y devuelve la distancia entre el punto almacenado dentro del objeto y el otro punto dado en un par de números flotantes.
- La clase proporciona un método llamado distance_from_point(point), que calcula la distancia (como el método anterior), pero la ubicación del otro punto se da como otro objeto de clase Point.

Completa la plantilla que te proporcionamos en el editor, ejecuta tu código y verifica si tu salida se ve igual que la nuestra.

```
import math


class Point:
    def __init__(self, x=0.0, y=0.0):
        #
        # Escribir el código aquí.
        #

    def getx(self):
        #
        # Escribir el código aquí.
        #

    def gety(self):
        #
        # Escribir el código aquí.
        #

    def distance_from_xy(self, x, y):
        #
        # Escribir el código aquí.
        #

    def distance_from_point(self, point):
        #
        # Escribir el código aquí.
        #


point1 = Point(0, 0)
point2 = Point(1, 1)
print(point1.distance_from_point(point2))
print(point2.distance_from_xy(2, 0))
```

Salida esperada

```
1.4142135623730951
1.4142135623730951
```

ejercicio

Objetivos

- Mejorar las habilidades del estudiante para definir clases desde cero.
- Emplear composición.

Escenario

Ahora vamos a colocar la clase *Point* (ver Lab 3.4.1.14) dentro de otra clase. Además, vamos a poner tres puntos en una clase, lo que nos permitirá definir un triángulo.¿Cómo podemos hacerlo?

La nueva clase se llamará Triangle y esto es lo que queremos:

- El constructor acepta tres argumentos - todos ellos son objetos de la clase Point.
- Los puntos se almacenan dentro del objeto como una lista privada
- La clase proporciona un método sin parámetros llamado perimeter(), que calcula el perímetro del triángulo descrito por los tres puntos; el perímetro es la suma de todas las longitudes de los lados (lo mencionamos para que conste, aunque estamos seguros de que tú mismo lo conoces perfectamente).

Completa la plantilla que te proporcionamos en el editor, ejecuta tu código y verifica si tu salida se ve igual que la nuestra.

```
import math


class Point:
    #
    # El código copiado del laboratorio anterior.
    #


class Triangle:
    def __init__(self, vertice1, vertice2, vertice3):
        #
        # Escribir el código aquí.
        #

    def perimeter(self):
        #
        # Escribir el código aquí.
        #


triangle = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))
print(triangle.perimeter())
```

A continuación puedes copiar el código de la clase Point, el cual se utilizo en el laboratorio anterior:

```
class Point:
    def __init__(self, x=0.0, y=0.0):
        self.__x = x
        self.__y = y
```

Salida esperada

```
3.414213562373095
```