05.MetodosMagicos

July 22, 2024

1 Atributos de clase protegidos

Los atributos dentro de una clase pueden ser modificados por el usuario directamente. Volvamos al ejemplo de la clase `Persona`.

[ ]:
```
class Persona:
    def __init__(self, nombre, edad, num_cuenta):
        self.nombre, self.edad, self.num_cuenta = nombre, edad, num_cuenta

    def mostrarInformación(self):
        print(f'{self.nombre} -> {self.edad}')
```

Con esta clase `Persona` se crea un objeto llamado `cliente1`.

[ ]:
```
cliente1 = Persona("Juan", 34, 123456789)
cliente1.mostrarInformación()
```

```
Juan -> 34
```

El objeto `cliente1` tiene los valores se sus propiedades pasadas por el constructor al momento de ser creado, y no existe limitante alguna para ser modificados directamente. Por ejemplo:

[ ]:
```
cliente1.nombre = "Juan Perez"
cliente1.mostrarInformación()
```

```
Juan Perez -> 34
```

Aunque esto es permitido por el intérprete en `Python`, es considerado como una mala práctica de programación. Lo que se debe hacer es escribir un módulo dentro de la clase que reciba el nuevo nombre y haga el cambio del valor de la propiedad. De esta manera se tiene un control y certeza sobre los cambios en las propiedades. Considere además el caso de que en la propiedad se almacene información sensible que no se desea revelar, sería necesario limitar el acceso a la propiedad que contenga esa información.

[ ]:
```
cliente1.num_cuenta
```

[ ]:
```
123456789
```

Dado que `Python` no posee mecanismo para evitar el acceso o que se modifiquen los valores de las propiedades, existe una convención para marcar los atributos como protegidos. Para ello se utiliza

el prefijo guión bajo _ en el nombre de la propiedad. Esto indica a cualquier otro programador que dicha propiedad no debería ser accedida o modificada fuera de la clase.

[ ]:
```python
class Persona:
    def __init__(self, nombre, edad, num_cuenta):
        self.nombre, self.edad, self._num_cuenta = nombre, edad, num_cuenta
        # Nótese que num_cuenta se marcó como propiedad protegida

    def mostrarInformación(self):
        print(f'{self.nombre} -> {self.edad}')
```

[ ]:
```python
cliente1 = Persona("Juan", 34, 123456789)
cliente1.mostrarInformación()
```

```
Juan -> 34
```

[ ]:
```python
cliente1._num_cuenta
```

[ ]:
```
123456789
```

Aunque la propiedad *_num_cuenta* aún puede ser impresa e incluso modificada, marcarla como protegida indica al programador que hacerlo va en contra de las buenas prácticas de codificación.

2 Métodos especiales

Los métodos especiales en Python, también conocidos como **métodos mágicos** o **dunder methods**, son funciones integradas dentro de las clases que permiten definir comportamientos específicos para operaciones estándar. Tienen dos guiones bajos al principio y al final de sus nombres, como `__init__` o `__str__`.

2.1 Algunos métodos especiales comunes:

- **`__init__`**: Inicializa una nueva instancia de una clase.
- **`__str__`**: Define cómo representar un objeto como una cadena de texto, usado en `str(obj)` y `print(obj)`.
- **`__repr__`**: Proporciona una representación oficial de un objeto, usada en `repr(obj)`.
- **`__len__`**: Devuelve el tamaño o longitud de un objeto, usado en `len(obj)`.
- **`__getitem__`, `__setitem__`, `__delitem__`**: Permiten acceder, modificar y eliminar elementos por índice.
- **`__iter__`, `__next__`**: Permiten que un objeto sea iterable, como en un bucle `for`.
- **`__eq__`, `__lt__`, `__gt__`**: Implementan comparaciones (igualdad, menor que, mayor que, etc.).
- **`__add__`, `__sub__`, `__mul__`, etc.**: Definen el comportamiento de operadores aritméticos.

Ya hemos utilizado el método `__init__` que es el constructor, y se ejecuta automáticamente cada vez que se crea un objeto/instancia de la clase.

El nombre **método especial** o más aún **método mágico** puede ser engañoso, ya que técnicamente no hay algo especial o mágico en ellos. Lo único especial acerca de ellos es el nombre, el cual asegura

que serán llamados en situaciones especiales. Por ejemplo, el método `__init__` que se ejecuta al crear un objeto.

Por ejemplo, considere la *fórmula general barométrica* para calcular la presión atmosférica p dada la altura h.

$$p = p_0 e^{-Mgh/RT}$$

donde M es la masa molar del aire, g es la constante gravitacional, R es la constante del gas, T la temperatura y p_0 la presión del aire a nivel del mar. Se define además $h_0 = \frac{RT}{Mg}$.

$$p = p_0 e^{-h/h_0}$$

Ahora se define una clase para el cálculo barométrico.

[ ]:
```
import math

class Barometric:
    def __init__(self, T):
        g = 9.81         # m/s²
        R = 8.314        # J/(K*mol)
        M = 0.02896      # kg/mol
        self.h0 = R*T/M/g
        self.p0 = 100          # kPa

    def value(self, h):
        return self.p0 * math.exp(-h/self.h0)
```

[ ]:
```
bar1 = Barometric(292.15)
bar1.value(2200)
```

[ ]:
```
77.31204126500637
```

Esta forma de la clase permite obtener el valor de la presión para cierto valor de T y h. El valor de la presión de obtiene al llamar al método `value` y pasarle el argumento h.

Sería más simple de utilizar si se pudiese llamar directamente al objeto sin necesidad de emplear el método intermedio. Para ello existe el método especial `__call__`.

Veamos una nueva versión de la clase empleando este método especial.

[ ]:
```
import math

class Barometric:
    def __init__(self, T):
        g = 9.81         # m/s²
        R = 8.314        # J/(K*mol)
        M = 0.02896      # kg/mol
        self.h0 = R*T/M/g
```

```
        self.p0 = 100        # kPa

    def __call__(self, h):
        return self.p0 * math.exp(-h/self.h0)
```

[ ]:
```
bar2 = Barometric(292.15)
print(bar2(2200))

# Es equivalente a esta forma de la llamada
print(bar2.__call__(2200))
```

```
77.31204126500637
77.31204126500637
```

2.2 Método especial para imprimir

Es posible imprmir un objeto `a` empleando un `print(a)`, lo cual funciona bien para los objetos propios de Python como cadenas y listas. Sin embrago, si nosotros creamos una clase, ese print no necesariamente mostrará información útil. Por ello tendremos que resolver ese problema definiendo el método `__str__` dentro de la clase. El método `__str__` debe devolver de preferencia una cadena y no debe recibir argumentos excepto por `self`.

Redefiniendo la clase `Barometric`, queda así:

[ ]:
```
import math

class Barometric:
    def __init__(self, T):
        g = 9.81         # m/s²
        R = 8.314        # J/(K*mol)
        M = 0.02896      # kg/mol
        self.h0 = R*T/M/g
        self.p0 = 100        # kPa
        self.T = T

    def __call__(self, h):
        return f'p(h = {h}, T = {self.T}) = {self.p0 * math.exp(-h/self.h0)}␣
 ↪kPa'

    def __str__(self):
        return f'p0 * exp(-Mgh/(RT)) [kPa]; T = {self.T}ºK'
```

[ ]:
```
bar3 = Barometric(292.15)
print(bar3(2200))
print(bar3)
```

```
p(h = 2200, T = 292.15) = 77.31204126500637 kPa
p0 * exp(-Mgh/(RT)) [kPa]; T = 292.15ºK
```

2.3 Métodos especiales para operaciones matemáticas

Hasta ahora hemos cubierto los métodos `__init__`, `__call__` y `__str__`, pero hay más de ellos. Por ejemplo, los métodos `__add__`, `__sub__` y `__mul__`. Definir estos métodos dentro de la clase nos permite emplear expresiones como $c = a + b$, donde a y b son instancias de una clase.

```
c = a + b       # c = a.__add__(b)

c = a - b       # c = a.__sub__(b)

c = a * b       # c = a.__mul__(b)

c = a / b       # c = a.__div__(b)

c = a ** b      # c = a.__pow__(b)
```

Para la mayoría de los casos, cualquiera de estas operaciones devuelve un objeto de la misma clase que los operandos.

De manera similar, también existen métodos especiales para comparar objetos:

```
a == b          # a.__eq__(b)

a != b          # a.__ne__(b)

a < b           # a.__lt__(b)

a <= b          # a.__le__(b)

a > b           # a.__gt__(b)

a >= b          # a.__ge__(b)
```

Estos métodos deben ser implementados para devolver un booleano, para que sea consistente con el comportamiento de los operadores de comparación.

El contenido de los métodos al momento de definirlos dependen del desarrollador, lo único especial acerca de los métodos es su nombre, ya que mediante este pueden ser llamados automáticamente por varios operadores.

Por ejemplo, si se desea multiplicar dos objetos $c = a*b$, Python buscará el método llamado `__mul__` en la instancia a. Si el método existe, será llamado pasandole como argumento la instancia b y cualquiera que sea la devolución del método `__mul__` se asigna a c.

2.4 Método especial `__repr__`

Este método especial es semejante al método `__str__`, ya que devuelve una cadena con información acerca del objeto. Por un lado la cadena devuelta por `__str__` muestra información que es fácilmente leíble y por otro lado la cadena devuelta por `__repr__` contiene la información necesaria para recrear el objeto.

Para un objeto `a` el método `__repr__` se puede llamar mediante la función nativa de Python llamada

repr(a).

[ ]:
```
import math

class Barometric:
    def __init__(self, T):
        g = 9.81         # m/s²
        R = 8.314        # J/(K*mol)
        M = 0.02896      # kg/mol
        self.h0 = R*T/M/g
        self.p0 = 100          # kPa
        self.T = T

    def __call__(self, h):
        return f'p(h = {h}, T = {self.T}) = {self.p0 * math.exp(-h/self.h0)}␣
 ↪kPa'

    def __str__(self):
        return f'p0 * exp(-Mgh/(RT)) [kPa]; T = {self.T}ºK'

    def __repr__(self):
        """ Return code for regenerating this instance """
        return f'Barometric({self.T})'
```

[ ]:
```
b3 = Barometric(292.15)
print(b3)
repr(b3)
```

```
p0 * exp(-Mgh/(RT)) [kPa]; T = 292.15ºK
```

[ ]:
```
'Barometric(292.15)'
```

[ ]:
```
b4 = eval(repr(b3))
print(b4)
```

```
p0 * exp(-Mgh/(RT)) [kPa]; T = 292.15ºK
```

Estos resultados confirman que el método `__repr__` funciona de acuerdo a lo esperado, dado que `eval(repr(b3))` devuelve un objeto idéntico a `b3`.

Ambos métodos `__str__` y `__repr__` muestran información acerca de un objeto, la diferencia es que una muestra información legible para humanos y la segunda información legible para Python.

2.5 Mostrar contenido de una clase

Algunas veces resulta útil mostrar el contenido de una clase, por ejemplo para realizar debugging.

Considere la siguiente clase de ejemplo que solo contiene un comentario, el constructor y una propiedad:

[ ]:
```
class A:
    """ Una clase de muestra """
    def __init__(self, value):
        self.v = value
```

Si se realiza un dir(A) se mostrarán varios métodos y propiedades que se han definido automáticamente en la clase.

[ ]:
```
dir(A)
```

[ ]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__']
```

Además, si se crea un objeto de la clase y se ejecuta el método `dir(a)` observaremos la misma salida mostrado con la clase pero también los valores creados por el constructor al momento de la creación del objeto.

[ ]:
```
a = A(2)
dir(a)
```

[ ]:
```
['__class__',
 '__delattr__',
 '__dict__',
```

```
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'v']
```

2.6 Método especial `__doc__`

El método especial `__doc__` muestra los comentarios que existen dentro de la definición de la clase que han sido escritos dentro de triples comillas dobles. Estos textos y este método son los que sirven para construir la documentación del código posteriormente.

[ ]:
```
a.__doc__
```

[ ]:
```
' Una clase de muestra '
```

[ ]:
```
A.__doc__
```

[ ]:
```
' Una clase de muestra '
```

[ ]:
```
import numpy
numpy.__doc__
```

[ ]:
```
'\nNumPy\n=====\n\nProvides\n  1. An array object of arbitrary homogeneous
items\n  2. Fast mathematical operations over arrays\n  3. Linear Algebra,
Fourier Transforms, Random Number Generation\n\nHow to use the
documentation\n----------------------------\nDocumentation is available in two
forms: docstrings provided\nwith the code, and a loose standing reference guide,
```

```
available from\n`the NumPy homepage <https://numpy.org>`_.\n\nWe recommend
exploring the docstrings using\n`IPython <https://ipython.org>`_, an advanced
Python shell with\nTAB-completion and introspection capabilities.  See below for
further\ninstructions.\n\nThe docstring examples assume that `numpy` has been
imported as ``np``::\n\n  >>> import numpy as np\n\nCode snippets are indicated
by three greater-than signs::\n\n  >>> x = 42\n  >>> x = x + 1\n\nUse the built-
in ``help`` function to view a function\'s docstring::\n\n  >>> help(np.sort)\n
… # doctest: +SKIP\n\nFor some objects, ``np.info(obj)`` may provide
additional help.  This is\nparticularly true if you see the line "Help on ufunc
object:" at the top\nof the help() page.  Ufuncs are implemented in C, not
Python, for speed.\nThe native Python help() does not know how to view their
help, but our\nnp.info() function does.\n\nTo search for documents containing a
keyword, do::\n\n  >>> np.lookfor(\'keyword\')\n  … # doctest:
+SKIP\n\nGeneral-purpose documents like a glossary and help on the basic
concepts\nof numpy are available under the ``doc`` sub-module::\n\n  >>> from
numpy import doc\n  >>> help(doc)\n  … # doctest: +SKIP\n\nAvailable
subpackages\n---------------------\nlib\n    Basic functions used by several
sub-packages.\nrandom\n    Core Random Tools\nlinalg\n    Core Linear Algebra
Tools\nfft\n    Core FFT routines\npolynomial\n    Polynomial tools\ntesting\n
NumPy testing tools\ndistutils\n    Enhancements to distutils with support for\n
Fortran compilers support and more  (for Python <=
3.11).\n\nUtilities\n---------\ntest\n    Run numpy unittests\nshow_config\n
Show numpy build configuration\nmatlib\n    Make everything
matrices.\n__version__\n    NumPy version string\n\nViewing documentation using
IPython\n-----------------------------------\n\nStart IPython and import `numpy`
usually under the alias ``np``: `import\nnumpy as np`.  Then, directly past or
use the ``%cpaste`` magic to paste\nexamples into the shell.  To see which
functions are available in `numpy`,\ntype ``np.<TAB>`` (where ``<TAB>`` refers
to the TAB key), or use\n``np.*cos*?<ENTER>`` (where ``<ENTER>`` refers to the
ENTER key) to narrow\ndown the list.  To view the docstring for a function,
use\n``np.cos?<ENTER>`` (to view the docstring) and ``np.cos??<ENTER>`` (to
view\nthe source code).\n\nCopies vs. in-place
operation\n-----------------------------\nMost of the functions in `numpy`
return a copy of the array argument\n(e.g., `np.sort`).  In-place versions of
these functions are often\navailable as array methods, i.e. ``x =
np.array([1,2,3]); x.sort()``.\nExceptions to this rule are documented.\n\n'
```

2.7 Otors métodos especiales

El método `__module__` devuelve el nombre del módulo al cual pertenece la clase, en el siguiente ejemplo se devuelve `__main__` dado que el objeto fue creado dentro de ese módulo.

[ ]:
```
a.__module__
```

[ ]:
```
'__main__'
```

El método especial `__dict__` devuelve un diccionario con las propiedades y los valores de un objeto.

[ ]:
```
a.__dict__
```

[ ]:
```
{'v': 2}
```

Una instancia contiene todos los atributos de la clase creados automáticamente por Python. Si se agregan nuevos valores al objeto, éstos son añadidos al diccionario.

[ ]:
```
a.myVar = 10
a.__dict__
```

[ ]:
```
{'v': 2, 'myVar': 10}
```

[ ]:
```
a.__getattribute__("v")
a.__getattribute__("myVar")
```

[ ]:
```
10
```

2.7.1 Ejemplo 1

Crear una clase que calcule la evaluación de la derivada de una función. Utilizar una definición genérica para el cálculo de una aproximación de la derivada.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

[ ]:
```
class Derivada:
    def __init__(self, f, h=1e-5):
        self.f = f  # Función f(x) a derivar
        self.h = h

    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h
```

[ ]:
```
def f(x):
    return x**3

dfdx = Derivada(f)

dfdx(1)
```

[ ]:
```
3.000030000110953
```

2.7.2 Ejemplo 2.

Crear una clase que calcule la aproximación de una integral definida. Para ello consideremos la regla compuesta del trapecio.

$$A = \int_a^b f(x)dx \approx \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)\right]$$

considere:

$$h = \frac{b-a}{n}$$

y

$$x_j = a + j \cdot h,$$

donde n es la cantidad de trapecios y h la altura de cada uno de ellos.

[ ]:
```
class TrapecioCompuesta:
    def __init__(self, f, n=1):
        self.f, self.n = f, n

    def __call__(self, a, b):
        f, n = self.f, self.n
        h = (b-a)/n
        suma = 0
        for j in range(1,n):
            suma += f(a + j*h)

        return f'A = {h/2*(f(a) + 2*suma + f(b))} u^2'
```

[ ]:
```
import math
def f(x):
    return math.sin(x)

intf = TrapecioCompuesta(f, 10000)

intf(math.pi, 2*math.pi)
```

[ ]:
```
'A = -1.9999999835506608 u^2'
```

2.7.3 Ejemplo 3. Polinomios

Crear una clase que permita construir un polinomio

$$P(x) = a_0 + a_1 x + a_2 x^2$$

La clase debe incluir la funcionalidad de evaluar un polinomio en un valor dado, y sumar dos polinomios. Se deben utilizar los métodos especiales para usarlos de la forma indicada:

- `__init__`: para constuir un polinomio de la forma `p = Polynomial([1,-1])`

- `__str__`: para imprimir el polinomio
- `__call__`: para evaluar el polinomio de la forma `p(2.0)`
- `__add__`: para realizar la suma de polinomios
- `__mul__`: para realizar la multiplicación de polinomios

Además se debe incluir un método para realizar la derivada del polinomio.

2.7.4 Solución

Creación de la clase Polynomial, el constructor y el método ´**call**´

[ ]:
```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s
```

[ ]:
```
p1 = Polynomial([1,-1])
p1(4)
```

[ ]:
```
-3
```

Implementación de la suma de polinomios

[ ]:
```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        # return self + other

        # we start with longest list and add it to the other
        if len(self.coeff) > len(other.coeff):
            coeffsum = self.coeff[:] # copy list
            for i in range(len(other.coeff)):
                coeffsum[i] += other.coeff[i]

        else:
            coeffsum = other.coeff[:] # copy list
```

```
            for i in range(len(self.coeff)):
                coeffsum[i] += self.coeff[i]

        return Polynomial(coeffsum)
```

[ ]:
```
p1 = Polynomial([-3, 0, 2, 1])      # x³ + 2x² -3
p2 = Polynomial([1, 1, 1])         # x² + x + 1
p3 = p1 + p2    # x³ + 3x² + x - 2
print(p3.coeff)
```

```
[-2, 1, 3, 1]
```

Para la multiplicación se requiere realizar un proceso un poco más complicado. Nos referiremos a la expresión matemática para la multiplicación de polinomios.

$$p_1 \cdot p_2 = \left(\sum_{i=0}^{M} c_i x^i\right)\left(\sum_{j=0}^{N} d_j x^j\right) = \sum_{i=0}^{M}\sum_{j=0}^{N} c_i d_j x^{i+j},$$

donde

$$p_1 = c_0 + c_1 x + c_2 x^2 + \cdots + c_M x^M$$

y

$$p_2 = d_0 + d_1 x + d_2 x^2 + \cdots + d_N x^N$$

[ ]:
```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        # return self + other

        # we start with longest list and add it to the other
        if len(self.coeff) > len(other.coeff):
            coeffsum = self.coeff[:] # copy list
            for i in range(len(other.coeff)):
                coeffsum[i] += other.coeff[i]

        else:
            coeffsum = other.coeff[:] # copy list
            for i in range(len(self.coeff)):
                coeffsum[i] += self.coeff[i]
```

```
        return Polynomial(coeffsum)

    def __mul__(self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff = [0]*(M+N+1) # [0 for i in range(10)]    # List of M+N+1 zeros
        for i in range(0, M+1):
            for j in range(0, N+1):
                coeff[i+j] += self.coeff[i] * other.coeff[j]

        return Polynomial(coeff)
```

[ ]:
```
p1 = Polynomial([-3, 0, 2, 1])      # x³ + 2x² -3
p2 = Polynomial([1, 1, 1])         # x² + x + 1
p4 = p1 * p2    # x³ + 3x² + x - 2
print(p4.coeff)
```

```
[-3, -3, -1, 3, 3, 1]
```

Para el cálculo de la derivada del polinomio, se puede utilizar la regla:

$$\frac{d}{dx}\sum_{i=0}^{n} c_i x^i = \sum_{i=1}^{n} i c_i x^{i-1}$$

Por lo tanto, si c es la lista de coeficientes del polinomio, la derivada tiene una lista de coeficientes en dc, donde $dc[i-1] = i \cdot c[i]$ para todos los valores de i desde 1 hasta el índice mayor en c. Recuerde que la derivada de un polinomio se reduce en grado en 1, por lo tanto la lista dc tendrá un elemento menos que la lista c.

[ ]:
```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        # return self + other

        # we start with longest list and add it to the other
        if len(self.coeff) > len(other.coeff):
            coeffsum = self.coeff[:] # copy list
            for i in range(len(other.coeff)):
                coeffsum[i] += other.coeff[i]
```

```
        else:
            coeffsum = other.coeff[:] # copy list
            for i in range(len(self.coeff)):
                coeffsum[i] += self.coeff[i]

        return Polynomial(coeffsum)

    def __mul__(self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff = [0]*(M+N+1) # [0 for i in range(10)]    # List of M+N+1 zeros
        for i in range(0, M+1):
            for j in range(0, N+1):
                coeff[i+j] += self.coeff[i] * other.coeff[j]

        return Polynomial(coeff)

    def differentiate(self):
        for i in range(1, len(self.coeff)):
            self.coeff[i-1] = i * self.coeff[i]
        del self.coeff[-1]

    def derivative(self):
        dpdx = Polynomial(self.coeff[:])
        dpdx.differentiate()
        return dpdx
```

[ ]:
```
p1 = Polynomial([-3, 0, 2, 1])      # x³ + 2x² -3
p1.derivative()
```

[ ]:
```
<__main__.Polynomial at 0x7fa9cc924550>
```

Por último, hace falta agregar la función `__str__` para mostrar el polinomio de una forma legible. El método debe devolver una representación del polinomio lo más cercana posible a como se escribe en Matemáticas.

[ ]:
```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
```

```
        # return self + other

        # we start with longest list and add it to the other
        if len(self.coeff) > len(other.coeff):
            coeffsum = self.coeff[:] # copy list
            for i in range(len(other.coeff)):
                coeffsum[i] += other.coeff[i]

        else:
            coeffsum = other.coeff[:] # copy list
            for i in range(len(self.coeff)):
                coeffsum[i] += self.coeff[i]

        return Polynomial(coeffsum)

    def __mul__(self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff = [0]*(M+N+1) # [0 for i in range(10)]    # List of M+N+1 zeros
        for i in range(0, M+1):
            for j in range(0, N+1):
                coeff[i+j] += self.coeff[i] * other.coeff[j]

        return Polynomial(coeff)

    def differentiate(self):
        for i in range(1, len(self.coeff)):
            self.coeff[i-1] = i * self.coeff[i]
        del self.coeff[-1]

    def derivative(self):
        dpdx = Polynomial(self.coeff[:])
        dpdx.differentiate()
        return dpdx

    def __str__(self):
        s = ''
        for i in range(0, len(self.coeff)):
            if self.coeff[i] != 0:
                s += f' + {self.coeff[i]:g}*x^{i:g}'
        # fix layout
        s = s.replace('+ - ', '- ')
        s = s.replace(' 1*', ' ')
        s = s.replace('x^0','1')
        s = s.replace('*1', '')
        s = s.replace('x^1', 'x')
        if s[0:3] == ' + ':
```

```
            s = s[3:]
        if s[0:3] == ' - ':
            s = '-' + s[3:]
        return s
```

$$p_1(x) = x^3 + 2x^2 - 3$$

[ ]:
```
p1 = Polynomial([-3, 0, 2, 1])      # x³ + 2x² -3
print(p1)
```

```
-3 + 2*x^2 + x^3
```

$$\frac{dp_1}{dx} = 3x^2 + 4x$$

[ ]:
```
p2 = p1.derivative()
print(p2)
```

```
4*x + 3*x^2
```