

F210 Programación Aplicada a finanzas

OOP. Parte 2. Polinomios

El trabajo consiste en crear una clase de polinomios "poly" que nos permitan realizar todas las operaciones algebraicas sobre ellos.

1. Un polinomio de grado n será definido por una estructura de datos conteniendo la siguiente informacion:

- (a) Un entero n indicando el grado del polinomio
- (b) Una estructura de almacenamiento $coefs$ conteniendo los coeficientes del polinomio. $coefs[0]$ corresponde a la potencia 0, $coefs[1]$ a la potencia 1 y así sucesivamente, de forma tal que el polinomio resulte

$$p_n(x) = \sum_{i=0}^n coefs[i] * x^i$$

2. La clase debe tener los siguientes métodos:

- (a) Instanciador `__init__(self, n=0, coefs=[0])` – el default es un polinomio de grado 0, una constante, con valor 0. Debe validarse la consistencia entre n y la lista de coeficientes.
- (b) Una funcion `get_expression(self)` que devuelva un string con la formula del polinomio contenida en la instancia, reportando solo los coeficientes distintos de 0. Por ejemplo, para el polinomio $p_3(x) = 2 + 3x - 5x^3$ la salida debería ser

$$p(x) = 2x^0 + 3x^1 - 5x^3$$

Un coeficiente se considerara 0 si su valor absoluto es menor a $1e-5$.

- (c) Una funcion `poly_plt(self, a, b, **kwargs)` (debera importa `matplotlib`) que grafique el polinomio en el intervalo $[a, b]$. `**kwargs` debe ser un diccionario conteniendo los parametros de graficacion que considere necesarios.
- (d) Las instancias de esta clase deben ser *callable* (es decir, se vuelven funciones), lo que implica que deben tener definido dentro de la clase el **magic method** `__call__`. De este modo, si A es una instancia de `poly`, al escribir en la línea de comandos $A(x)$ Python debe devolver a la salida el polinomio asociado a A evaluado en x de acuerdo con la expresion ya presentada

$$p_n(x) = \sum_{i=0}^n coefs[i] * x^i$$

- (e) Magic methods `__add__` y `__radd__` (operador "+") que permitan sumar un escalar (int o float) o un polinomio de clase `poly` a otro polinomio con las siguientes especificaciones:

- En caso que se trate de un escalar, se procedera a construir un polinomio de grado 0 y repetir la operacion, ahora entre polinomios.

- Si los sumandos son de distinto grado, se debe extender el polinomio de menor grado al de mayor grado y proceder a la suma.
- (f) Magic methods `__sub__` y `__rsub__` (operador "-") que permitan restar un escalar (int o float) o un polinomio de clase `poly` a otro polinomio con las siguientes especificaciones:
- En caso que se trate de un escalar, se procedera a construir un polinomio de grado 0 y repetir la operacion, ahora entre polinomios.
 - Si los polinomios son de distinto grado, se debe extender el polinomio de menor grado al de mayor grado y proceder a la resta.
- (g) Magic methods `__mul__`, `__rmul__` (operador "*") que permitan multiplicar un escalar (int o float) o un polinomio de clase `poly` a otro polinomio con las siguientes especificaciones:
- En caso que se trate de un escalar, se procedera a construir un polinomio de grado 0 y repetir la operacion, ahora entre polinomios.
- (h) Magic methods `__floordiv__` y `__rfloordiv__` (operador "//") que permitan dividir de forma entera un escalar (int o float) o un polinomio de clase `poly` por otro polinomio con las siguientes especificaciones:
- En caso que se trate de un escalar, se procedera a construir un polinomio de grado 0 y repetir la operacion, ahora entre polinomios.
- (i) Magic methods `__mod__` y `__rmod__` (operador "%") que permitan obtener el polinomio resto de `floordiv`.
- (j) Una funcion `rootfind(self)` que, mediante el metodo de rootfinding de su eleccion, busque una raiz real del polinomio
- (k) Un método `findroots(self)` que busque todas las raíces reales del polinomio. (Pista: valgame del hecho que si x_0 es una raíz de $p_n(x)$, entonces $p_n(x)$ es divisible exactamente por el monomio $(x - x_0)$ una cantidad k de veces, donde k es la multiplicidad de la raíz x_0 , quedando para la búsqueda residual el polinomio divisor de grado $n - k$). El método debe devolver a la salida una lista conteniendo tuplas con pares (x_0, k) – raíz y multiplicidad –, y por último un objeto de tipo `poly` con el polinomio residual $p_r(x)$ para el que no se haya encontrado raíces reales. Si el polinomio $p_n(x)$ no tuviera raíces reales, debe devolver una lista sin tuplas y solamente con $p_r(x) = p_n(x)$ como polinomio residual, y si tuviera n raíces reales, debera devolverse la lista de tuplas y el polinomio residual $p_r(x) = 0$.
- (l) Un método `factorize(self)` que llame a `findroots` y que a partir de la salida imprima la factorizacion del polinomio de la forma
- $$p_n(x) = (x - x_0) * k_0 * (x - x_1) * k_1 * \dots * p_r(x)$$
- (m) Método `fprime(self, k, x0 = None)` que evalua la derivada n -esima del polinomio. Si x_0 es `None`, debe devolver la derivada enésima como un miembro de la familia `poly`, si x_0 es un valor, debe devolver la derivada enésima evaluada en x_0
3. Construya dos subclases de polinomios `linear(poly)` y `quadratic(poly)` y haga el overriding de todos los métodos que le parezca tenga sentido particularizar para hacer más eficiente el funcionamiento de esas clases.