Pensamiento Matemático 1

Lino AA Notarantonio
Departamento de Economía, Santa Fe

September 25, 2019

Contents

1 Introducción

El propósito de este documento es implementar en Python algunos temas del curso de Pensamiento Matemático 1 para los alumnos de la Escuela de Ciencias Sociales y Gobierno.

El documento empieza con algunos temas generales numéricos. Después, se introducen tópicos básicos de Python y luego se introduce el módulo `sympy` que se usará para manipular de manera simbólica funciones y sus derivadas, entre otros.

Iniciamos importando los módulos necesarios.

[1]:
```
from math import *
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

2 Operaciones de punto flotante

En cálculo numérico por medio de una computadora, los números reales, por lo general, no tienen una representación decimal finita.

Usando el sistema decimal, sabemos que el número $1/10$ tiene una terminación decimal finita, $1/10 = .1$, pero $1/3 = .3333\ldots$ no.

Una situación semejante ocurre con la representación de los números reales en una computadora. Las computadoras ocupan *aritmética binaria* para operar; en la representación binaria de los números reales, por lo general, no se tiene una terminación binaria finita.

Además, las computadoras pueden guardar en memoria sólo una cantidad finita de *dígitos significativos* después del punto decimal, lo que lleva a *errores de redondeo*.

Cada computadora moderna sigue las especificaciones del estándar IEEE-754 para representar números reales en aritmética binaria; una distinción adicional es la aritmética en *simple precisión* y *doble precisión*. Por lo general, Python usa representación IEEE-754 en doble precisión.

A continuación se presentan unas consecuencia de aritmética binaria en computadoras.

Cálculo del epsilon de la máquina La primera consecuencia es que el cero numérico no es *necesariamente* igual al cero matemático. El *epsilon de la máquina*, ε_s, es el valor más grande que es numéricamente lo mismo que cero. Se presenta una implementación del epsilon de la máquina usando un ciclo *while*; la idea es de dividir el valor inicial de ε_s hasta que $\varepsilon_s + 1 = 1$:

[2]:
```
eps = 1
while eps + 1 > 1:
    eps = eps/2
print(eps)
eps + 1
```

```
1.1102230246251565e-16
```

[2]:
```
1.0
```

Se puede observar que el *orden de magnitud* de ε_s es de 10^{-16}.

Otro problema que se encuentra con el cálculo numérico es que, aún si ciertas fracciones tienen terminación decimal finita, no necesariamente tienen terminación *binaria* finita.

Por ejemplo,

$$.1 + .1 + .1 = .3$$

pero

[3]:
```
.1 + .1 + .1
```

[3]:
```
0.30000000000000004
```

Podemos verificar cómo $.1 + .1 + .1$ no es numéricamente lo mismo que .3, usando el *operador de comparación* `==` (valores: `True`, `False`)

[4]:
```
.1 + .1 + .1 == .3
```

[4]:
```
False
```

3 Introducción a Python

3.1 Variables

Una *variable* almacena en memoria un valor de un tipo dado. Los tipos de datos en Python son

- números enteros
- números de punto flotante
- números complejos
- cadenas de caracteres
- tuplas
- listas
- booleanas (verdadero; falso)

Dada una variable, se puede determinar su tipo usando `type()`.

El tipado de las variables en Python es *dinámico*, es decir se define la variable "on the fly": no es necesario declarar la variable, con el tipo, con antelación.

Ejemplos de los diferentes tipos de datos se dan a continuación.

[5]:
```
a = 4
type(a)
```

[5]:
```
int
```

En la línea de código arriba, el símbolo `=` el es *operador de asignación*, que asigna el número entero `4` (a la derecha) a la variable `a` (a la izquierda). La asignación `a = 4` declara la variable `a` como entera, por el valor que se pasa a esta variable.

La asignación `b = 4.0` declara la variable `b` como de punto flotante:

[6]:
```
b = 4.0
type(b)
```

[6]:
```
float
```

Cuando se manipulan variables de tipo diferentes, el resultado es de tipo más alto

[7]:
```
a*b
type(a*b)
```

[7]:
```
float
```

A continuación, se define una lista, que es una variable de tipo `str` (caractéres)

[8]:
```
c = "lista" # lista
type(a*c)
```

[8]:
```
str
```

Operaciones algebráicas como suma, producto por un entero, tienen el efecto de *concatenación* de las listas.

[9]:
```
string1 = 'press Return to exit'
string2 = 'the program'
print(string1 + ' ' + string2)
```

```
press Return to exit the program
```

[10]:
```
5*'Hello'
```

[10]:
```
'HelloHelloHelloHelloHello'
```

El resultado es una lista (tipo más alto en la jerarquia)

[11]:
```
type(5*'Hello')
```

[11]:
```
str
```

Manipulación de una stringa usando un bucle (*loop*)

[12]:
```
for i in range(5):
    print('hello')
```

```
hello
hello
hello
hello
hello
```

Implementación del valor absoluto usando un control de flujo `if-else`

[13]:
```
def val_abs(x):
    if x > 0:
        return x
    else:
        return -x
```

[14]:
```
val_abs(-4)
```

[14]:
```
4
```

A continuación desarrollaremos más en detalles las estructuras de control más comunes

3.2 Estructuras de control

Una estructura de control incluye una iteración o una selección.

Python proporciona dos estructuras de control iterativas:

- `while`
- `for`

Una estructura iterativa necesita de una condición de paro, de manera que pueda terminar de manera correcta.

3.2.1 Estructura de control while

El bucle `while` se ejecuta hasta que la condición de paro se vuelva falsa.

A continuación se crea una lista vacía y se añaden los valores calculados por el bucle:

[15]:
```
N = 1; lista = []
while N <= 10:
    lista.append(2*N)
    N = N+1
lista
```

[15]:
```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

3.2.2 Estructura de control for

La estructura iterative `for` es muy útil cuando los valores sobre los cuales se itera se encuentran en un rango definido.

A continuación elencaremos los primeros enteros cuadrados menores o iguales a 10; crearemos una lista y añadiremos cada entero cuadrado a esta lista:

[16]:
```
lista=[]
for i in range(1,11):
    lista.append(i**2)
lista
```

[16]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Se puede usar el bucle `for` para iterar sobre caracteres.

A continuación se presenta un ejemplo en donde, dada una lista de palabras, se extrae una letra a la vez y se añade a la lista `listaletras`. Se inicializa `listaletras` como una lista vacía.

[17]:
```
lista = ['perro', 'gato', 'Rosa Maria']
listaletras = []
for i in lista:
    for j in i:
        listaletras.append(j)
listaletras
```

[17]:
```
['p',
 'e',
 'r',
 'r',
 'o',
 'g',
 'a',
 't',
 'o',
 'R',
 'o',
 's',
 'a',
 ' ',
```

```
 'M',
 'a',
 'r',
 'i',
 'a']
```

3.2.3 Estructura de selección

Una estructura de selección típica es un `if else`; si la condición del `if` es verdadera se ejecuta, si no lo es, el programa pasa el control a la condición del `else`.

A continuación se presentan estructuras de control para calcular, de ser posible, la ráiz cuadrada de un número real. Si el número seleccionado no es positivo, o cero, entonces se da un mensaje informativo; de lo contrario, se calcula la raíz cuadrada del número dado:

[18]:
```
for i in range(-6,9):
    if i<0:
        print('No se puede')
    else:
        print(sqrt(i))
```

```
No se puede
No se puede
No se puede
No se puede
No se puede
No se puede
0.0
1.0
1.4142135623730951
1.7320508075688772
2.0
2.23606797749979
2.449489742783178
2.6457513110645907
2.8284271247461903
```

También se puede usar un constructo `if else` anidado. A continuación, se presenta un ejemplo en donde una lista de calificaciones numéricas se cambia a una lista de calificaciones en letras. Se comenta el output de las calificaciones individuales.

[19]:
```
scores = [40, 95, 82, 72, 56, 91, 88]
letterscore = []
for score in scores:
    if score >= 90:
        #print('A')
        letterscore.append('A')
    else:
        if score >= 80:
            #print('B')
```

```
            letterscore.append('B')
        else:
            if score >= 70:
                #print('C')
                letterscore.append('C')
            else:
                #print('F')
                letterscore.append('F')
letterscore
```

[19]: `['F', 'A', 'B', 'C', 'F', 'A', 'B']`

Un constructo más ágil, quizás, es usando `elif`:

[20]:
```
scores = [40, 95, 82, 72, 56, 91, 88]
letterscore = []
for score in scores:
    if score >= 90:
        letterscore.append('A')
    elif score >= 80:
        letterscore.append('B')
    elif score >= 70:
        letterscore.append('C')
    else:
         letterscore.append('F')
letterscore
```

[20]: `['F', 'A', 'B', 'C', 'F', 'A', 'B']`

3.3 List Comprehension

Una *list comprehension* permite definir una lista mediante estructuras de selección (`for`, `if`).

Por ejemplo, si queremos generar una lista con los primeros diez enteros elevados al cuadrado, entonces podemos usar una estructura de selección (nótese el uso del método `append()` para añadir los cuadrados a la lista):

[21]:
```
lista=[]
for i in range(1,11):
    lista.append(i**2)
lista
```

[21]: `[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]`

Usando una *list comprehension* el código resulta más simple:

[22]:
```
lista=[i**2 for i in range(1,11)]
lista
```

[22]: `[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]`

Una *list comprehension* se define como una lista que contiene la estructura de selección `for`.

Cualquier secuencia definida mediante una iteración se puede usar para definir una lista mediante una *list comprehension*.

Una *list comprehension* permite también la inclusión de un criterio de selección, de manera que sólo unos cuantos términos se creen.

Por ejemplo, extraemos de la palabra "computadora" las vocales; `car.upper()` es el método que cambia el caracter a mayúscula:

[23]:
```
vocales = [car.upper() for car in 'computadora' if car in 'aeiou']
vocales
```

[23]:
```
['O', 'U', 'A', 'O', 'A']
```

Un ejemplo numérico que incluya sólo los cuadrados de los enteros *pares*, entre 1 y 10, es

[24]:
```
lista=[i**2 for i in range(1,11) if i%2 == 0]
lista
```

[24]:
```
[4, 16, 36, 64, 100]
```

Ciertas operaciones algebráicas (por ejemplo, la suma de elementos usando `sum()`) no se pueden realizar de manera directa en una *list comprehension*. Se puede usar el método `reduce` del módulo `functools` sobre una función anónima (`lambda`).

Un ejemplo de este técnica se encuentra más adelante, en el cálculo de una anualidad.

4 El módulo SymPy

En esta sección se introduce el módulo `sympy` (*Symbolic Python*). Este módulo permite manipular de manera simbólica, no necesariamente numérica, funciones en Python.

Iniciamos con algunos ejemplos introductorios, como el manejo de números irracionales como π, e (la base de los logaritmos naturales), entre otros.

[25]:
```
sym.pi
```

[25]: π

[26]:
```
sym.exp(1)
```

[26]: e

Es importante enfatizar que la misma función matemática puede estar implementada en Python en módulos diferentes.

Por ejemplo, la función `sqrt()` es implementada en los módulo `math` y `sympy`, con características diferentes.

En el módulo `math`, `sqrt(2)=1.4142135623731`; en el módulo `sympy`, `sym.sqrt(2)` $=\sqrt{2}$.

[27]:
```
sym.pi**sqrt(2)
```

[27]: $\pi^{1.4142135623731}$

[28]:
```
sym.pi**sym.sqrt(2)
```

[28]: $\pi^{\sqrt{2}}$

Es posible evaluar numéricamente una expresión de `sympy` usando el método `evalf()`, como se muestra a continuación:

[29]:
```
(sym.pi**sym.sqrt(2)).evalf()
```

[29]: 5.04749726737091

Si se requieren menos dígitos significativos, por ejemplo, 5 dígitos, entonces se puede pasar el número apropiado al método `evalf()`:

[30]:
```
(sym.pi**sym.sqrt(2)).evalf(5)
```

[30]: 5.0475

El parte fuerte de sympy está en la posibilidad de manipulaciones algebráicas simbólicas.

[31]:
```
a = sym.Rational(1,3)
b = sym.Rational(1, 2)
a + sym.sqrt(b)
```

[31]: $\frac{1}{3} + \frac{\sqrt{2}}{2}$

Comparado con

[32]:
```
(1/3) + sqrt(1/2)
```

[32]: `1.040440114519881`

El módulo sympy incluye el símbolo `sym.oo` (dos 'o' minúsculas seguidas), que simboliza el infinito matemático

[33]:
```
1/sym.oo
```

[33]: 0

La siguiente condición lógica resulta verdadera, `True` cuando se compara `sym.oo` con números reales.

[34]:
```
sym.oo > 10**(10000)
```

[34]: True

A continuación se muestran unos ejemplos de manipulación simbólica de sympy.

Empezamos con la simplificación de la expresión

$$x + y - 2x + 3y$$

[35]:
```
x = sym.Symbol('x')
y = sym.Symbol('y')
x+y-2*x+3*y
```

[35]: $-x + 4y$

Después, podemos expander la expresión $(x + y)^6$,

[36]:
```
sym.expand((x + y) ** 6)
```

[36]: $x^6 + 6x^5y + 15x^4y^2 + 20x^3y^3 + 15x^2y^4 + 6xy^5 + y^6$

Podemos simplificar expresiones trigonométricas,

[37]:
```
sym.trigsimp(sym.sin(x)/sym.cos(x)*(sym.sin(y)**2 - sym.cos(y)**2))
```

[37]: $-\cos(2y)\tan(x)$

Simplificar la expresión

$$\frac{x + x\sin(x)}{x^2 - x\cos(x)}$$

[38]:
```
sym.simplify((x + x * sym.sin(x)) / (x**2 - x * sym.cos(x)))
```

[38]: $\frac{\sin(x) + 1}{x - \cos(x)}$

Resolver ecuaciones, lineales y no, de manera paramétrica, o no:

[39]:
```
a = sym.Symbol('a')
b = sym.Symbol('b')
c = sym.Symbol('c')
sym.solveset(a*x +b , x)
```

[39]: $\left\{-\frac{b}{a}\right\}$

[40]:
```
sym.solveset(3*x + 4/5 , x)
```

[40]: $\{-0.266666666666667\}$

La bien conocida ecuación cuadrática, usando los símbolos definidos arriba:

[41]:
```
sym.solveset(a*x**2 + b*x + c , x)
```

[41]: $\left\{-\frac{b}{2a} - \frac{\sqrt{-4ac+b^2}}{2a}, -\frac{b}{2a} + \frac{\sqrt{-4ac+b^2}}{2a}\right\}$

Encontrar las raíces del polinomio $x^4 - 16$:

[42]:
```
sym.solveset(x ** 4 - 16, x)
```

[42]: $\{-2, 2, -2i, 2i\}$

Sistemas de ecuaciones lineales Considera el siguiente modelo de mercado,

$$D = 12 - 2P$$
$$S = -2 + 1.6P$$

Se quiere determinar el precio, P, y cantidad de equilibrio, Q.

Considerando que, en el equilibrio, $D = S = Q$, entonces, se debe resolver el sistema de ecuciones lineales

$$0 = 12 - 2P - Q$$
$$0 = -2 + 1.6P - Q$$

Podemos usar la función `sym.solve()` para encontrar los valores de equilibrio:

[43]:
```
P = sym.Symbol('P')
Q = sym.Symbol('Q')
equilibrio = sym.solve((- 2 * P - Q + 12, 1.6 * P - Q - 2), (P, Q))
equilibrio[P], equilibrio[Q]
```

[43]:
```
(3.88888888888889, 4.22222222222222)
```

También se pueden resolver algunos sistemas no lineales. Por ejemplo,

$$Q = 12 - .05P^2 - 2P$$
$$Q = -2 + 1.6P$$

En un contexto de equilibrio de mercado, el par (P, Q) con valores negativos no es aceptable.

[44]:
```
sym.solve((- 2 * P - .05 * P ** 2 - Q + 12, 1.6 * P - Q - 2), (P, Q))
```

[44]:
```
[(-75.6988664825584, -123.118186372093), (3.69886648255842, 3.91818637209347)]
```

5 Valor futuro y anualidades

En esta sección se implementarán algunos temas relacionados con valor futuro y anualidades (cálculo de un plan de ahorro).

Primeramente, se introducen las funciones exponenciales, que son necesarias para estos temas.

5.1 Funciones exponenciales

Una función exponencial

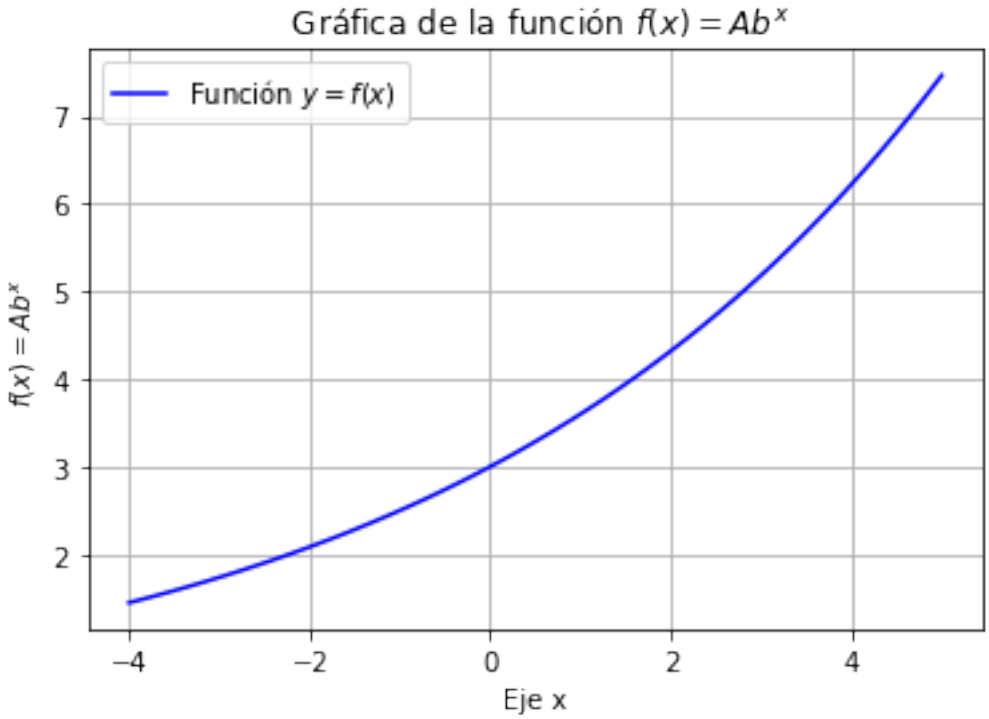
$$f(x) = Ab^x,$$

donde A, b son números reales.

Consideramos, por ejemplo, $A = 3$, $b = 1.2$, con $-4 \leq x \leq 5$.

[45]:
```
A = 3
b = 1.2
x = np.linspace(-4,5,100)
y = A*b**x
```

[46]:
```
plt.plot(x, y, '-b', label = 'Función $y = f(x)$')
plt.xlabel('Eje x')
plt.ylabel('$f(x) = Ab^x$')
plt.grid(True)
plt.legend()
plt.title(' Gráfica de la función $f(x) = A b^x$')
plt.show()
```



5.2 Interés compuesto

Supón que invertimos un monto P en un activo que paga un interés anual $r = .07$ (que corresponde a un 7% anual).

Después de un año, el monto resultante será igual a

$$S_1 = P(1 + .07).$$

Después de dos años, el monto resultante (asumiendo que se reinvierta todo S_1) será

$$S_2 = S_1(1 + .07) = P(1 + .07)^2.$$

Después n años,

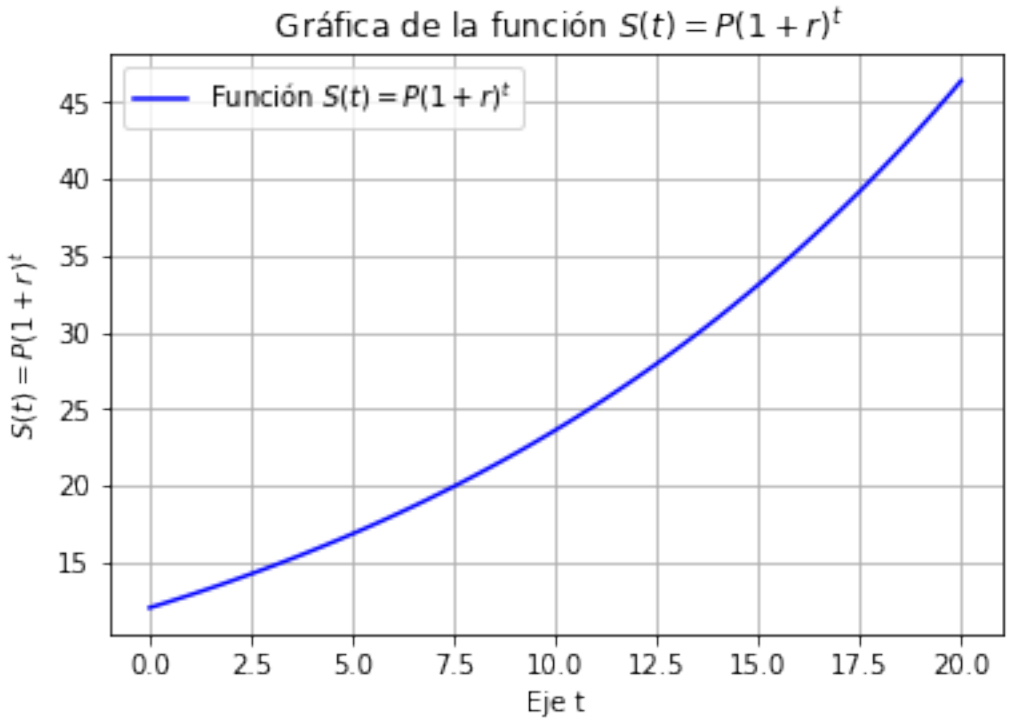
$$S_n = P(1 + .07)^n.$$

Graficamos $S(t)$, como una función del tiempo $0 < t < 20$, con $P = 12$ (unidades monetarias), para visualizar el crecimiento del dinero en el tiempo.

La función $S(t)$ se conoce como el *valor futuro* del principal P.

[47]:
```
r = .07
P = 12
t = np.linspace(0,20,100)
St = P*( 1 + r)**t
```

[48]:
```
plt.plot(t, St, '-b', label = 'Función $S(t) = P(1 + r)^t$')
plt.xlabel('Eje t')
plt.ylabel('$S(t) = P(1 + r)^t$')
plt.grid(True)
plt.legend()
plt.title(' Gráfica de la función $S(t) = P(1 + r)^t$')
plt.show()
```



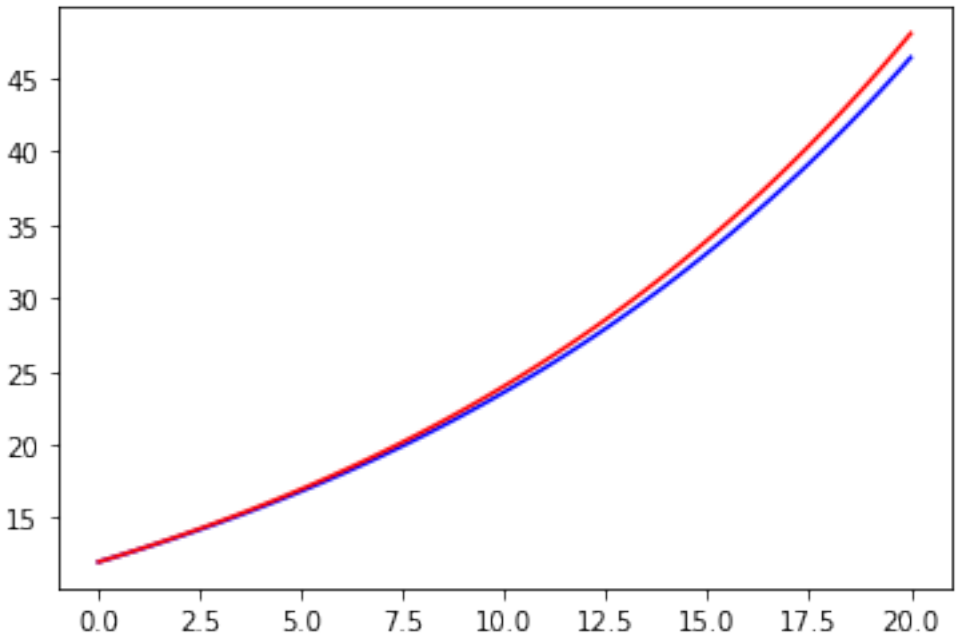
Supón que el interés del 7% ($r = .07$) se capitaliza trimestralmente; entonces,

$$T(t) = P\left(1 + \frac{r}{4}\right)^{4t}$$

La función $T(t)$ es el valor futuro con interés $r = .07$, capitalizado *trimestralmente*.

[49]:
```
Tt = P*( 1 + r/4)**(4*t)
```

[50]:
```
plt.plot(t, St, '-b')
plt.plot(t, Tt, '-r')
plt.show()
```



Supón que el interés del 7% ($r = .07$) se capitaliza mensualmente; entonces,

$$M(t) = P\left(1 + \frac{r}{12}\right)^{12t}$$

La función $M(t)$ es el valor futuro con interés $r = .07$, capitalizado *mensualmente*.

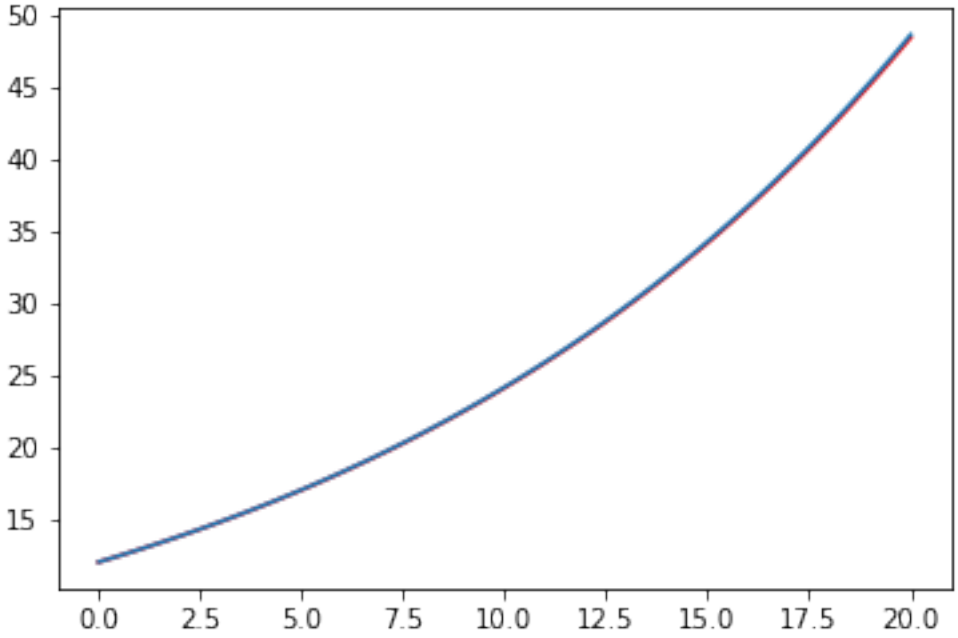
Si se dejara que la frecuencia de capitalización, n, se incrementara cada vez más, $n \to \infty$, entonces resulta

$$\lim_{n\to\infty} P\left(1 + \frac{r}{n}\right)^{nt} = e^{rt}$$

En este último caso, se dice que el interés se capitaliza *continuamente*.

A continuación, se presentan por simplicidad de visualización sólo las gráficas del interés capitalizado mensualmente y continuamente.

[51]:
```
Mt = P*( 1 + r/12)**(12*t)
plt.plot(t, Mt, 'r')
plt.plot(t, P*np.exp(r*t))
plt.show()
```



5.3 Plan de ahorro

En un plan de ahorro, se deposita un monto constante, A, periodicamente, al inicio de cada año; considerando un horizonte de planeación de n años, tendremos un flujo de depósitos desde $t = 0$ (inicio del primer año) hasta $t = n - 1$ (inicio del último año del plan):

$$t = 0 : \ S_1 = A(1 + r)^n$$

hasta

$$t = n - 1 : \ S_n = A(1 + r)$$

Los montos parciales, S_i, constituyen una serie (o progresión) geométrica.

El monto total, al término del año N, será

$$S = \sum_{i=1}^{n} S_i = \sum_{i=1}^{n} A(1 + r)^i = A \left(\frac{1 + r}{r} \right) \left[(1 + r)^n - 1 \right].$$

A continuación, presentamos

- una *list comprehension*, que es una manera rápida de generar una lista, para los montos intermedios S_i;
- una función para poder calcular el monto total al término del horizonte del plan.

5.3.1 List comprehension

Declaramos los parámetros del plano, *e.g.*, $A = 500$ (pesos anuales), $n = 10$, $r = .08$ (tasa anual):

[52]:
```
A = 500
n = 10
r = .08
montos_parciales = [A * (1+r)**(10-i) for i in range(n)]
montos_parciales
```

[52]:
```
[1079.462498636394,
 999.5023135522166,
 925.4651051409413,
 856.9121343897605,
 793.4371614720003,
 734.6640384000003,
 680.2444800000002,
 629.8560000000001,
 583.2,
 540.0]
```

La función `range(n)` tiene valores de 0 hasta $n - 1$; de esta manera, el exponente, $10 - i$, permite el cálculo de montos desde el tiempo $t = 0$ hasta $t = 10$.

Suma de un list comprehension Para sumar los elementos de un *list comprehension* no se puede usar `sum()` directamente. Conviene usar el método `reduce` del módulo `functools` aplicado a una función (anónima) `lambda`; para una discusión de las funciones `lambda` pueden revisar esta liga: https://medium.com/better-programming/lambda-map-and-filter-in-python-4935f248593

[53]:
```
import functools
functools.reduce(lambda a, b: a + b, montos_parciales)
```

[53]:
```
7822.743731591313
```

5.3.2 Función

El cálculo del monto final al término del horizonte de ahorro lo podemos calcular también definiendo una función apropiada en Python. En la segunda línea del código se pone la fórmula que obtuvimos arriba, con los valores apropiados de `A`, `n` y `r`; en la tercera línea se despliega el resultado.

[54]:
```
def ahorro(A,n,r):
    S = A * (1+r)/r * ((1.08) ** n - 1)
    print(S)
```

El mismo cálculo del monto final que hicimos con la *list comprehension* se realiza usando `ahorro(15,10,.08)`:

[55]:
```
ahorro(500,10,.08)
```

```
7822.74373159132
```

6 Funciones elementales

6.1 Polinomios

Entre las funciones más comunes, se encuentran los polinomios. Un polinomio de grado n (n entero) es

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x_1 + a_0,\ a_n \neq 0.$$

Un polinomio de grado n está determinado por sus *coeficientes*.

Por ejemplo, $p(x) = x^3 - 2x^2 + 4x - 3$ está perfectamente identificado por el *vector* `[1, -2, 4, -3]`.

La clase `poly1d` de librería `numpy` permite el manejo de los polinomios por medio de vectores.

A continuación se presentan ejemplos de polinomios y su manejo, específicamente:

- Definición de un polinomio de grado n.
- Evaluación de un polinomio en un valor dado.
- Suma/resta; división de polinomios.
- Gráfica de polinomios.

[56]:
```
p = np.poly1d([1, -2, 4, -3])
print(p)
```

```
   3     2
1 x - 2 x + 4 x - 3
```

Se puede cambiar la variable, de ser necesario:

[57]:
```
p = np.poly1d([1, -2, 4, -3], variable = 't')
print(p)
```

```
   3     2
1 t - 2 t + 4 t - 3
```

Calcular el polinomio en $t = -.3$:

[58]:
```
p(-.3)
```

[58]:
```
-4.407
```

Sumar *algebráicamente* polinomios, usando `polyadd`. A continuación calcularemos

$$p(x) - p_2(x)$$

donde $p(x) = x^3 - 2x^2 + 4x - 3$, $p_2(x) = -x^3 + 2x^2 + 3x + 2$.

Usando el método `polyadd`, la resta de los polinomios se calcula mediante $p(x) + (-p_2(x))$.

[59]:
```
p2 = np.poly1d([-1,2,3,2])
p3 = np.polyadd(p, -p2)
print(p3)
```

```
   3      2
2 x - 4 x + 1 x - 5
```

Se pueden dividir dos polinomios, con residuo. Por ejemplo,

$$\frac{x^3 - 2x^2 + 4x - 3}{-x^3 + 2x^2 + 3x + 2} = -1 + \frac{7x - 1}{-x^3 + 2x^2 + 3x + 2}$$

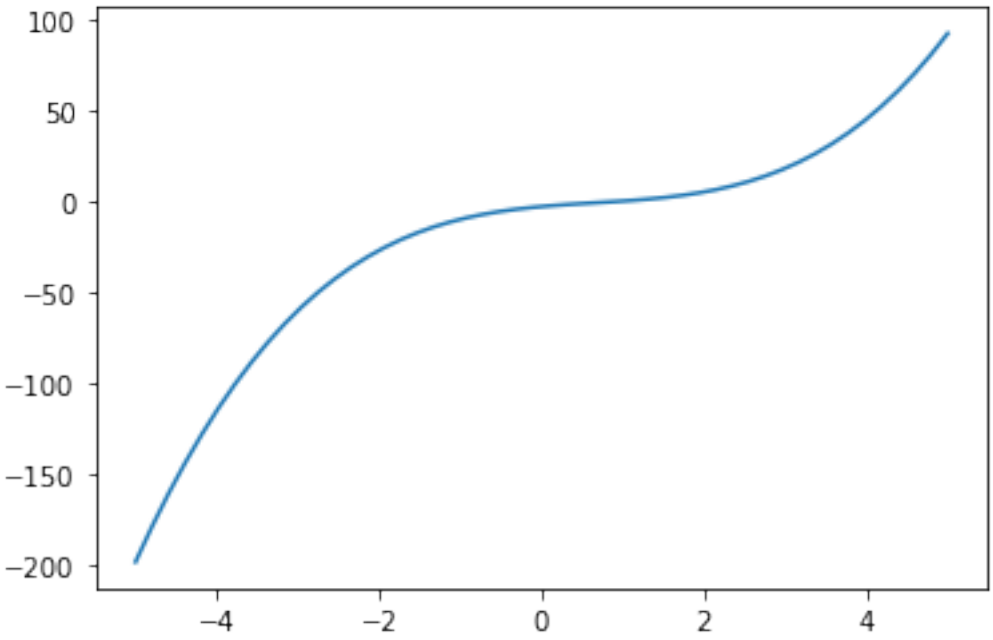
El resultado es

[60]:
```
p4 = np.polydiv(p,p2)
print(p4)
```

```
(poly1d([-1.]), poly1d([ 7., -1.]))
```

Para graficar un polinomio, definimos un intervalo apropiado en x y definimos la función correspondiente $y = p(x)$. Después, usamos `matpotlib` para la gráfica.

[61]:
```
x = np.linspace(-5,5)
y = p(x)
plt.plot(x,y)
plt.show()
```



Para calcular raíces de un polinomio con Python, podemos usar el método `r`, aplicado al polinomio. El resultado es un `array` con las raíces, posiblemente complejas conjugadas.

Como ejemplo, las raíces del polinomio $p(x) = x^3 - 2x^2 + 4x - 3$ son

$$0.5 \pm 1.6583124i, \quad 1$$

Nótese que la unidad imaginaria, i, se denota con j en Python.

[62]:
```
p.r
```

[62]:
```
array([0.5+1.6583124j, 0.5-1.6583124j, 1. +0.j       ])
```

Para verificar que los valores regresados son las raíces,

[63]:
```
p(p.r)
```

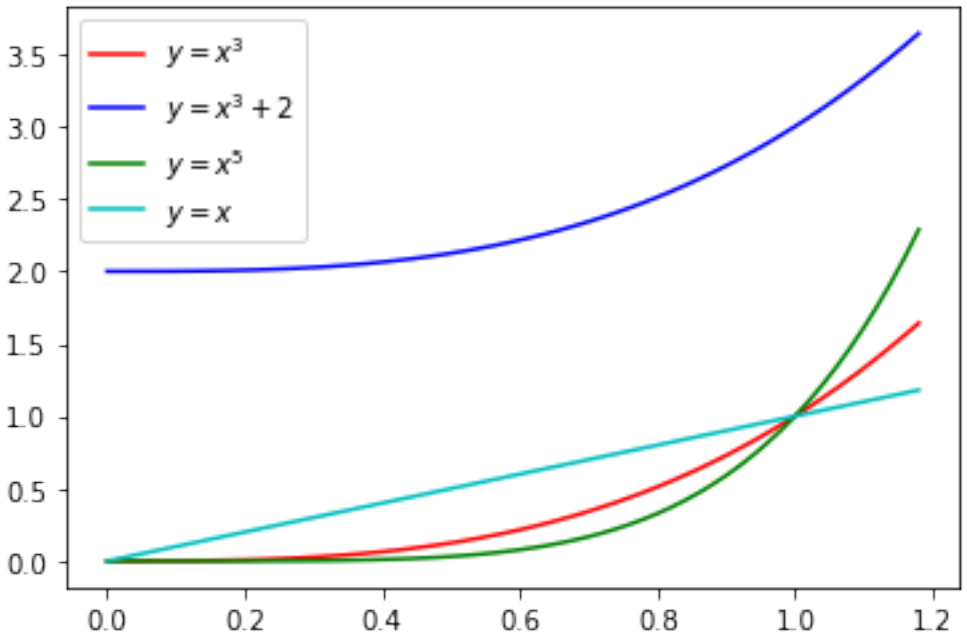
[63]:
```
array([1.33226763e-15-1.55431223e-15j, 1.33226763e-15+1.55431223e-15j,
       0.00000000e+00+0.00000000e+00j])
```

Los valores regresados son $(0, 0, 0)$ en la precisión de la computadora.

[ ]:
```
```

[64]:
```
x = np.arange(0.0, 1.2,0.02)
f1 = x**3
f2 = x**3 + 2
f3 = x**5
f4 = x

plt.plot(x, f1, 'r', label = '$y = x^3$')
plt.plot(x, f2, 'b', label = '$y = x^3 + 2$')
plt.plot(x, f3, 'g', label = '$y = x^5$')
plt.plot(x, f4, 'c', label = '$y = x$')
plt.legend()
plt.show()
```



7 Derivadas

En este capítulo veremos tópicos relacionados con el *cambio* de una función.

7.1 Cambio promedio de una función

Si $y = f(x)$ es una función y x_0 es un punto en su dominio, entonces el *cambio promedio* cuando pasamos de x_0 a $x_0 + \Delta x$ es

$$\frac{\Delta f}{\Delta x} = \frac{f(x_0) + \Delta x) - f(x_0)}{\Delta x}.$$

La expresión del cambio promedio se conoce como *cociente de Newton* o *cociente incremental*.

Si $f(x) = a + bx$, entonces el cambio promedio es constante, $\Delta f / \Delta x = b$.

En el código a continuación, se prefiere usar $h = \Delta x$.

7.1.1 Cambios con SymPy

Usando el módulo `sympy` podemos realizar cálculos simbólicos y numéricos de las funciones elementales.

Iniciamos con el cálculo del cambio promedio de una función. Después, pasaremos al cálculo de las derivadas (tasa instantánea de cambio).

Como se mostrará abajo, el cambio promedio de la función $y = f(x)$ depende de cuánto cambia la variable x, es decir, de cuanto vale Δx.

El cambio promedio se puede definir mediante la siguiente función de Python:

[65]:
```
def cambioprom(f,x,h):
    df = (f(x+h)-f(x))/(h)
    return df
```

Calculamos el cambio promedio de la función $y = \sqrt{x}$, con $\Delta x = .001$:

[66]:
```
df1 = cambioprom(sym.sqrt,1, .01)
print('El cambio promedio =', df1)
```

```
El cambio promedio = 0.498756211208895
```

Si $\Delta x = .0001$, el valor del cambio promedio también cambia:

[67]:
```
df2 = cambioprom(sym.sqrt, 1, .0001)
df1 - df2
```

[67]: -0.00123128941507122

Calculamos, de manera simbólica, el cambio promedio de una función cuadrática

$$y = ax^2 + bx + c$$

[68]:
```
x = sym.Symbol('x')
a = sym.Symbol('a')
b = sym.Symbol('b')
c = sym.Symbol('c')
h = sym.Symbol('h')

def q(x):
```

```
        return a*x**2 + b*x + c
sym.simplify(cambioprom(q, 0, h))
```

[68]: $ah + b$

Calculamos el cambio simbólico de la función

$$f(x) = x^3 + 3\sqrt{x} + 2$$

```
def f(x):
        return x**3 + 3*sym.sqrt(x) + 2
cambioprom(f, 1, h)
```

[69]: $$\frac{3\sqrt{h+1} + (h+1)^3 - 4}{h}$$

y después, usando $h = -.001$:

```
cambioprom(f, 1, -.001)
```

[70]: 4.49737618761681

Cuando $h = .00001$, el cambio promedio de la función $f(x)$ es

```
cambioprom(f,1, .00001)
```

[71]: 4.50002625012047

7.2 Derivadas

En los ejemplos anteriores emerge cierto patrón: el cambio promedio se acerca a cierto valor, cuando el valor h se acerca a cero.

La derivada formaliza esta intuición, considerando el cambio promedio, cuando $h \to 0$:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

siempre y cuando el límite exista.

En el módulo `sympy` se puede usar la función `sym.diff()` para calcular la derivada de una función, en un punto dado x.

Por ejemplo, si

$$f(x) = x^3 + 3\sqrt{x} + 2$$

su derivada es

```
sym.diff(f(x), x)
```

[72]: $$3x^2 + \frac{3}{2\sqrt{x}}$$

Se puede también asignar el resultado de una derivada para calcular los valores de la variable donde la derivada es igual a cero.

Por ejemplo, en el caso de la función cuadrática

$$q(x) = ax^2 + bx + c,$$

$q'(x) = 0$ cuando $x = -b/2a$:

```
fprime = sym.diff(q(x), x)
sym.solveset(fprime, x)
```

[73]: $$\left\{ -\frac{b}{2a} \right\}$$

7.2.1 Propiedades de límites y derivadas

La derivada de una función está ligada al proceso de *límite* de una función.

$$\lim_{x \to x_0} f(x)$$

Intuitivamente hablando, el concepto de límite permite determinar los valores de la función, cuando los valores de la variable independiente se acercan a x_0.

Considérese, por ejemplo, la función

$$f(x) = \frac{3x^2 + 3x - 18}{x - 2}.$$

Cuando los valores de la variable x se encuentran cerca de $x_0 = 2$, los valores de la función se acercan al valor 15, como se puede observar abajo:

[74]:
```
x = [1.9, 1.99, 1.999, 1.9999, 1.99999, 2.00001, 2.0001, 2.001, 2.01, 2.1]
for k in x:
    print([k,  (3* k ** 2 + 3 * k - 18) / (k - 2)])
```

```
[1.9, 14.699999999999976]
[1.99, 14.969999999999915]
[1.999, 14.996999999999137]
[1.9999, 14.999700000012925]
[1.99999, 14.999969999864291]
[2.00001, 15.000030000224527]
[2.0001, 15.000299999993736]
[2.001, 15.003000000001752]
[2.01, 15.029999999999752]
[2.1, 15.299999999999997]
```

Sustituyendo en la función el valor $x_0 = 2$, el numerador y el denominador se hacen cero, $0/0$. Esto implica que $x_0 = 2$ es una raíz del numerador y del denominador.

Factorizando el numerador,

[75]:
```
import sympy as sym
x = sym.Symbol('x')
sym.factor(3*x**2+3*x-18)
```

[75]: $3\,(x-2)\,(x+3)$

podemos entonces simplificar la función, alrededor de $x_0 = 2$, de manera que

$$f(x) = 3(x+3).$$

El comportamiento de la función alrededor del valor $x_0 = 2$ es ahora claro, porque se acerca al valor $3(2+3) = 15$.

El cálculo del límite está implementado en SymPy con la función `sym.limit()`:

[76]:
```
sym.limit((3*x**2+3*x-18)/(x-2), x, 2)
```

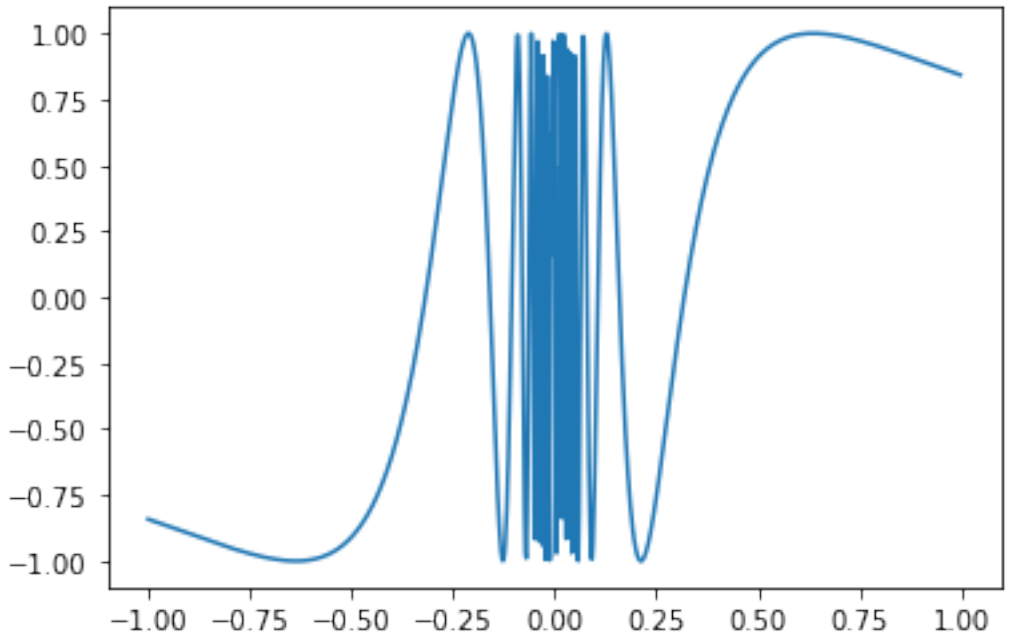
[76]: 15

El límite de una función puede no existir cuando los valores de la función no se acercan a ningún valor concreto.

Un ejemplo bien conocido de una función que no tiene límite, cuando $x \to 0$, es

$$f(x) = \sin(\frac{1}{x})$$

[77]:
```
from numpy import *
x = arange(-1.0, 1.0, 0.002)
plt.plot(x, sin(1/x))
plt.show()
```



Propiedades del límite Las reglas más usadas para el cálculo de límites se dan a continuación. Se supone que

$$\lim_{x \to x_0} f(x) = A, \quad \lim_{x \to x_0} g(x) = B.$$

1. $\lim_{x \to x_0} [f(x) + g(x)] = A + B.$
2. $\lim_{x \to x_0} [f(x) - g(x)] = A - B.$
3. $\lim_{x \to x_0} [f(x) \cdot g(x)] = A \cdot B.$
4. $\lim_{x \to x_0} \left[\frac{f(x)}{g(x)}\right] = \frac{A}{B}, B \neq 0.$
5. $\lim_{x \to x_0} [f(x)]^r = A^r$, si A^r está definido.

Propiedades de la derivada Dos propiedades de la derivada son consecuencia de las correspondientes propiedades del límite (suma; resta). A continuación, las dos funciones $f(x)$, $g(x)$ tienen derivadas $f'(x_0)$, $g'(x_0)$, respectivamente, en x_0.

1. La derivada de la suma es la suma de las derivadas:
$$\frac{d}{dx}[f(x)+g(x)] = \frac{df}{dx} + \frac{dg}{dx}.$$

2. La derivada de la resta es la resta de las derivadas:
$$\frac{d}{dx}[f(x)-g(x)] = \frac{df}{dx} - \frac{dg}{dx}.$$

Otras propiedades importantes de la derivada son

3. la derivada del producto
$$\frac{d}{dx}\left[f(x)\cdot g(x)\right] = g(x)\frac{df}{dx} + f(x)\frac{dg}{dx}.$$

4. la derivada del cociente
$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{f'(x)g(x) - f(x)g'(x)}{[g(x)]^2}$$

Reglas de derivación

1. **Regla de la potencia**
$$\frac{dx^a}{dx} = ax^{a-1},$$
donde a es una constante arbitraria.

2. La derivada de una constante es cero

3. Las constantes multiplicativas se conservan al derivar:
$$\frac{d}{dx}\left[Af(x)\right] = Af'(x)$$

4. **Regla generalizada de la potencia**
$$\frac{d}{dx}[g(x)]^a = a[g(x)]^{a-1}\frac{dg}{dx}.$$

5. **Regla de la cadena** Sean $y = y(u)$, $u = u(x)$ funciones derivables de sus argumentos. Entonces la función compuesta
$$y = y(u(x))$$
es también una función derivable de x; además,
$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}.$$

[ ]: