Clase 1 - Variables y funciones

March 25, 2017

1 Introducción

Esta es la primera clase de un cursillo de Python 3 que se dictó en marzo del 2017 en la Universidad Nacional de Colombia, sede Manizales. El cursillo está orientado a matemáticos y estudiantes de matemáticas que ya están ligeramente familiarizados con varios conceptos en programación (como, por ejemplo, qué es un condicional o un ciclo y cómo se usan). Todos los apuntes se hicieron en jupyter notebook, y todo en todo el curso se usó la distribución de anaconda.

En este jupyter notebook se pueden encontrar los contenidos de la primera clase del curso introductiorio a python. En esta clase se trataron los siguientes temas:

1. Usar Jupyter Notebook para procesar python.
2. Python como calculadora.
3. Tipos de variables y asignación.
4. Flujo de control: condicionales y ciclos.
5. Definición de funciones.

2 Usando Jupyter Notebook y Spyder

Jupyter notebook es algo así como un editor de python. Las ventajas que tiene son las siguientes:

1. Es modular por celdas, y las celdas pueden contener no solo código, sino texto. Esto permite comentar nuestros códigos de una forma más limpia.
2. En las partes de texto admite Markdown, un lenguaje de marcas (como HTML), y adentro de Markdown podemos escribir código TeX (por ejemplo $F(x, y, z) = (e^{x+y}, e^y + z)$ o $\sum_{i=1}^{n} a_i$)
3. Si trabajamos en Linux (o si tenemos instalado Pandoc), podemos descargar nuestro trabajo como un pdf que podemos enviar a colegas para revisiones.

2.1 Atajos usuales

En Jupyter tenemos dos modos (que nombraremos por conveniencia *modo azul* y *modo edición*). Mientras estamos en modo azul, podemos movernos entre celdas y ejecutar los siguientes atajos con el teclado:

- *ctrl* + *enter* para compilar una celda.
- *a* para insertar una celda arriba de la celda actual.
- *b* para insertar una celda debajo de la celda actual.

- *d d* para eliminar una celda.
- *i i* para interrumpir el código (por si quedamos en un ciclo infinito o por si nuestro computador se está quemando).

Un resumen más completo de los atajos se puede encontrar en el pequeño teclado al lado de *CellToolbar*, en la barra superior.

Podemos entrar a *modo edición* haciendo clic en una celda una o dos veces o presionando *enter*. Podemos salir de modo edición (y entrar a modo azul) presionando *esc*.

2.2 Un poco de Markdown

Markdown permite poner texto en *cursiva* al rodearlo de asteriscos, en **negrilla** al rodearlo entre dobles asteriscos. Podemos hacer listas usando el guión usual y hacer listas numeradas usando 1., 2., . . . ; podemos inclusive poner hipervínculos poniendo entre llaves [,] el texto y justo después entre paréntesis el link. Por ejemplo esto. Por último, podemos poner títulos, subtítulos y así usando numeral de forma iterada (es decir, # Título, ## Subtítulo. . .)

2.3 Sobre IPython

IPython es un emulador de consola de python. La única diferencia en la que nos vamos a concentrar es su manejo de la ayuda: usando '?' después de una función o paquete podemos obtener ayuda *in situ*.

2.4 Sobre Spyder

Como ya vimos, podemos usar Jupyter Notebook para experimentar y documentar el código. Una vez tenemos todos los experimentos hechos y la idea clara, podemos pasar a Spyder y escribir el script como un archivo .py. Spyder permite también experimentar un poco al ofrecer una (o más) terminales de IPython.

3 Python como calculadora

Python se puede usar como calculadora, y almacena los enteros cuán grandes sean.

```
In [1]: 2+2

Out[1]: 4

In [2]: (463456*23425)**250

Out[2]: 835655238362011361648381696169681259176238980722311968966295444824203985793
```

Sin embargo hay que tener cuidado: la aritmética de punto flotante en python no es la más precisa.

```
In [3]: 100 - 99.95

Out[3]: 0.04999999999999716

In [4]: 0.1 + 0.1 + 0.1 == 0.3
```

```
Out[4]: False

In [5]: 1/10 + 1/10 + 1/10 == 3/10

Out[5]: False

In [6]: 1 + 1 + 1 == 3

Out[6]: True
```

4 Tipos de variables en python

En python hay enteros, reales, cadenas de caracteres, booleanos, listas, conjuntos y diccionarios.

```
In [7]: print(type('Hola mundo'))
        print(type(2))
        print(type(2.0))
        print(type(False))
        print(type([1,2,3]))
        print(type({1,2,3}))
        print(type({1: 1}))

<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'list'>
<class 'set'>
<class 'dict'>
```

4.1 Operaciones entre variables

4.1.1 Operaciones entre strings

Recordemos que podemos acceder a todos los atributos y métodos de una clase (o tipo de variable) escribiendo por ejemplo *dir(str)*.

```
In [8]: 'Hola' + 'Mundo' # Concatenación

Out[8]: 'HolaMundo'

In [9]: 'hola'.upper()

Out[9]: 'HOLA'

In [10]: 'hola'.islower()

Out[10]: True
```

Podemos acceder a cierto caracter en un string en python. Para hacerlo, ponemos entre llaves la posición a la que queremos acceder:

```
In [11]: 'hola'[0]

Out[11]: 'h'
```

Como pueden notar, python comienza indexando en 0: la primera posición está rotulada con 0, la segunda con 1 y así.

```
In [12]: 'hola'[1]

Out[12]: 'o'
```

4.1.2 Operaciones entre enteros

```
In [13]: 1 + 2 #Suma

Out[13]: 3

In [14]: 1*3*4 #Multiplicación

Out[14]: 12

In [15]: 3 ** 4 #Exp.

Out[15]: 81

In [16]: 4 / 3 #División normal

Out[16]: 1.3333333333333333

In [17]: 4 // 3 #División entera

Out[17]: 1

In [18]: 21 % 6 #Módulo

Out[18]: 3
```

4.1.3 Operaciones entre floats

```
In [19]: 0.1 + 0.2

Out[19]: 0.30000000000000004

In [20]: 1.4 * 3

Out[20]: 4.199999999999999

In [21]: 4.5 ** 2.3

Out[21]: 31.7971929089206

In [22]: 4.5.as_integer_ratio()

Out[22]: (9, 2)

In [23]: 0.1.as_integer_ratio()

Out[23]: (3602879701896397, 36028797018963968)
```

4.1.4 Operaciones entre bools

```
In [24]: False or True

Out[24]: True

In [25]: True and not True

Out[25]: False

In [26]: not True

Out[26]: False
```

4.1.5 Operaciones entre listas

```
In [27]: [1,2] + [3,4,5] # Concatenación

Out[27]: [1, 2, 3, 4, 5]

In [28]: A = [1,2,3]
         A.append(4) # Añadir un elemento al final
         A

Out[28]: [1, 2, 3, 4]

In [29]: A = [1,2,3]
         A[0] # Acceder al elemento en la posición 0.

Out[29]: 1
```

En la clase entrante veremos más propiedades y operaciones que se pueden hacer con listas. Además, veremos los diccionarios y cómo se trabaja con ellos.

5 Asignación de variables y comparación.

La asignación de variables se hace con un "=", por ejemplo:

```
In [30]: verdad = True
         hola = 'hola'
         numero_favorito = 42

In [31]: print(type(verdad))
         print(type(hola))
         print(type(numero_favorito))

<class 'bool'>
<class 'str'>
<class 'int'>
```

```
In [32]: a = numero_favorito - 1
         a

Out[32]: 41

In [33]: b = 7

In [34]: b += 1
         b

Out[34]: 8

In [35]: b = b + 1
         b

Out[35]: 9
```

Podemos comprarar dos valores usando "==", por ejemplo:

```
In [36]: numero_favorito = 42
         numero_favorito == 24

Out[36]: False

In [37]: numero_favorito == 42

Out[37]: True
```

6 Flujo de control

En python están los flujos de control usuales (condicionales y ciclos), además de uno no tan normal llamado try-except. El alcance de cada flujo de control está indicado por el nivel de indentación.

6.1 Condicionales

```
In [38]: a = 9
         if a < 0:
             print('a es negativo')
         elif a == 0:
             print('a vale 0')
         else:
             print('a es positivo')

a es positivo
```

6.2 Ciclos

el ciclo for tiene una filosofía diferente al resto: mientras que en MATLAB y en C++ se itera cambiando el valor de una variable cierta cantidad de veces, en python se itera sobre objetos **secuenciables** (strings, listas, entre otros).

```
In [39]: for letter in 'Miguel':
             print(letter)

M
i
g
u
e
l
```

Lo que va justo después de la palabra *for* es una variable muda.

```
In [40]: for x in [1,2,3]:
             print(x+1)

2
3
4
```

Es una buena práctica ser tan explísitos como podamos con las variables, de tal forma que sea más legible cuando volvamos al código 80 años después o cuando se lo presentemos a un colega.

```
In [41]: for number in [1,2,3]:
             print(number+1)

2
3
4
```

Para iterar un número entero en un rango (práctica usual en el resto de lenguajes de programación), usamos los objetos *range*:

```
In [42]: for i in range(0,10):
             print(i)

0
1
2
3
4
5
```

```
6
7
8
9
```

Como podemos notar, los objetos *range* nunca toman el límite superior.
Podemos alterar el paso ingresándolo como un tercer parámetro:

```
In [43]: for j in range(0,10,2):
             print(j)

0
2
4
6
8
```

Los ciclos for admiten siempre **un** objeto para iterar sobre él. Sin embargo, éste objeto puede contener más información: puede ser por ejemplo la tupla (i,j,k), y estar iterando sobre una lista [(1,1,1), (1,1,2)...]. Por ejemplo:

```
In [44]: for i,j in [(1,1), (2,2), (2,1)]:
             print('i: ' + str(i) + ', ' + 'j: ' + str(j))

i: 1, j: 1
i: 2, j: 2
i: 2, j: 1
```

Y está también el ciclo while:

```
In [45]: a = 10
         while a > 0:
             a = a - 1
             print(a)

9
8
7
6
5
4
3
2
1
0
```

Por último, está *try-except*. La estructura del try-except es simple: se ejecuta lo que está en la parte de *try*. Si no ocurre ningún error o excepción, se omite la parte del *except*. Si sí hay un error, la ejecución para y se pasa al *except*.

```
In [46]: a = 4
         while a>-5:
             try:
                 print(1/a)
             except:
                 print('hubo un error')
             a -= 1
```

```
0.25
0.3333333333333333
0.5
1.0
hubo un error
-1.0
-0.5
-0.3333333333333333
-0.25
```

7 Definiendo funciones en python

La estructura de las funciones en python es la siguiente:

```
def nombre(parámetros):
    código
    return valor #no necesariamente.
```

Por ejemplo, definamos una función que pida un número n y devuelva el factorial $\prod_{k=1}^{n} k = n!$

```
In [47]: def factorial(n):
             acumulador = 1
             for k in range(1,n + 1):
                 acumulador *= k

             return acumulador
```

```
In [49]: n = factorial(5)
         print(n)
```

```
120
```

8 Ejercicios

8.1 Primer ejercicio

Escriba una función que tome dos listas A y B y devuelva una lista con todos las parejas ordenadas con primera componente en A y segunda en B (es decir, una *lista* $A \times B$).

8.2 Segundo ejercicio

Supongamos que construimos el objeto *función conjuntista* en python como una lista de parejas ordenadas, es decir

```
A = [1,2,3]
B = ['a', 'b', 'c']
f = [(1, 'a'), (2, 'b'), (3, 'c')]
```

Escriba una función que, dada una lista f de parejas ordenadas y dos listas A y B, determine si f es en efecto una función de A en B y retorne *True* o *False* dependiendo del caso. Recuerde que un conjunto (en este caso una lista) de parejas ordenadas se dice función si

1. La lista es un subconjunto de $A \times B$.
2. Todo elemento de A es primera componente de una pareja.
3. A cada elemento del dominio le corresponde una única imagen.

Aunque no hemos hablado formalmente de las variables de tipo *tuple* (es decir, parejas ordenadas, triplas, etc), su comportamiento es prácticamente idéntico al de las listas. Podemos acceder al primer elemento de la tupla

```
pareja = (1, 'a')
```

diciendo

```
pareja[0]
```

Es decir

```
pareja[0] == 1 # es True
pareja[1] == 'a' # también es True
```

Pista: Uno puede evaluar pertenencia con la palabra clave *in*. Por ejemplo, si definimos f como en el enunciado, la evaluación

```
(1, 'a') in f
```

sería *True*.

8.3 Tercer ejercicio

Escriba una función que retorne *True* si un número es par y *False* si un número es impar.

Pista: si esta función le toma más de una línea, es probable que esté haciendo algo mal.

Clase 2 - Listas y diccionarios

March 25, 2017

1 Introducción

En esta segunda clase resolvemos los ejercicios planteados en la primera y atendemos las posibles dudas al respecto, y entramos en más detalle en listas y diccionarios.

2 Desarrollo de los ejercicios de la clase anterior

2.1 Primer ejercicio

Escriba una función que tome dos listas A y B y devuelva una lista con todos las parejas ordenadas con primera componente en A y segunda en B (es decir, una *lista* $A \times B$).

```
In [1]: def productocartesiano(lista1, lista2):
            producto = []
            for elemento1 in lista1:
                for elemento2 in lista2:
                    producto.append((elemento1, elemento2))

            return producto
```

```
In [2]: productocartesiano([1,2,3], ['a', 'b', 'c'])
```

```
Out[2]: [(1, 'a'),
         (1, 'b'),
         (1, 'c'),
         (2, 'a'),
         (2, 'b'),
         (2, 'c'),
         (3, 'a'),
         (3, 'b'),
         (3, 'c')]
```

2.2 Segundo ejercicio

Supongamos que construimos el objeto *función conjuntista* en python como una lista de parejas ordenadas subconjunto de $A \times B$ con A y B listas, es decir

```
A = [1,2,3]
B = ['a', 'b', 'c']
f = [(1, 'a'), (2, 'b'), (3, 'c')]
```

Escriba una función que, dada una lista f de parejas ordenadas, determine si ella es en efecto una función (es decir, retorne *True* si la lista es función y *False* en caso contrario.)

```
In [3]: def esfuncion(f, A, B):
            lista_de_preimagenes = []

            # Verificamos que f sea subconjunto de A x B
            for (preimagen, imagen) in f:
                lista_de_preimagenes.append(preimagen)
                if preimagen not in A or imagen not in B:
                    # print('f no es subconjunto de AxB')
                    return False

            # Verificamos que todo elemento de A es una preimagen
            if lista_de_preimagenes != A:
                # print('dominio de f no es A')
                return False

            # Verificamos la unicidad de la imagen.
            for (preimagen1, imagen1) in f:
                for (preimagen2, imagen2) in f:
                    if preimagen1 == preimagen2 and imagen1 != imagen2:
                        # print('no hay unicidad en las imágenes')
                        return False

            return True
```

Notamos que en esta implementación preferimos claridad a eficiencia.

```
In [4]: A = [1,2,3]
        f = [(1, 1), (1, 2), (2, 3)]
        print(esfuncion(f, A, A))
        g = [(1, 1), (2, 2), (3, 3)]
        print(esfuncion(g, A, A))
```

```
False
True
```

2.3 Tercer ejercicio

Escriba una función que retorne *True* si un número es par y *False* si un número es impar.

```
In [5]: def espar(n):
            return n%2 == 0
```

```
In [6]: print(espar(2))
        print(espar(3))

True
False
```

3 Almacenamiento y Objetos mutables e inmutables

Hablemos ahora sobre cómo python almacena variables y sobre cómo ciertas variables son mutables y cómo otras son inmutables.

3.1 Almacenamiento de variables

Cuando escribimos en nuestros códigos

```
a = 5
```

ocurren tres cosas:

1. python crea el objeto 5 y lo almacena en la memoria.
2. python crea la variable a.
3. python hace que la variable a apunte al objeto 5.

Vale la pena tener en cuenta que una variable nunca apunta a otra variable: por ejemplo cuando escribimos

```
b = a
```

la variable *b* no apunta a *a* sino al objeto relacionado (i.e. 5).

```
In [7]: a = 5
        b = a
        # print(a is b) (es True)
        a = a + 1
        # print(a is b) (es False, porque ya no son el mismo obj.)
        print(b) # b no se ha modificado.

5
```

3.2 Objetos mutables e inmutables

Un objeto se dice **mutable** si se puede modificar, y se dice **inmutable** en caso contrario. En python, todos los objetos normales son **inmutables** salvo

- las listas
- los conjuntos
- los diccionarios

Es decir: los enteros, los strings, los floats, los bools son todos imutables.

Hay que tener especial cuidado con la asignación con objetos mutables. Por ejemplo:

```
In [8]: L1 = [1,2,3]
        L2 = L1
        L1[0] = 10
        print(L2) # Como L2 estaba apuntando al mismo objeto que L1,
        # aparecerá [10, 2, 3]
```

```
[10, 2, 3]
```

Es decir, cuando hacemos `L1[0] = 10` estamos modificando el objeto mutable `[1,2,3]` a `[10,2,3]`. Como `L2` estaba apuntando a este objeto, queda `L2 = [10, 2, 3]`.

Sin embargo, si volvemos al ejemplo anterior:

```
In [9]: a = 5
        b = a
        a = a + 1
        print(b)
```

```
5
```

vemos que `b` no se ha modificado porque el objeto al que apuntaba `a` (i.e. `5`) es inmutable, entonces cuando decimos `a = a+1` estamos en verdad creando un objeto nuevo (en este caso `6`) y haciendo que `a` apunte a él.

4 Listas

Como habíamos hablado, existen los objetos de tipo `list` en python. Habíamos hablando de cómo acceder a sus elementos, de cómo saber su longitud usando `len` y de cómo pegarle más elementos usando `append`. Ahora vamos a hablar un poco más sobre listas.

4.1 Entrando a elementos en listas

Estamos acostumbrados a entrar a los elementos en las listas usando índices positivos, pero también podemos usar índices negativos (en donde -1 es el último elemento, -2 el penúltimo…). Por ejemplo:

```
In [10]: L = [1,2,3,4]
         print(L[0])
         print(L[3])
         print(L[-1])
```

```
1
4
4
```

4.2 Definiendo listas por extensión y por comprensión

Solemos definir listas así:

```
In [11]: L = [1,2,3,4,5,6]
```

pero, ¿qué pasa si necesitamos una lista con los números del 1 al 100? Podríamos intentar la siguiente aproximación (que no es para nada *pythonica*)

```
In [12]: L = []
         for k in range(1,101):
             L.append(k)
         print(L)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
```

Sin embargo existe una forma mucho más elegante y corta de definir una lista al describir sus elementos:

```
In [13]: L = [k for k in range(1, 101)]
         print(L)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
```

¡Miren eso!, con solo una línea hacemos lo que antes nos demoraba 3. Podemos mezclar condicionales también:

```
In [14]: K = [m for m in L if m >= 50]
         print(K)
```

```
[50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70
```

4.3 Un par de métodos más

Recordemos que un método es una función específica a una clase. Revisemos otros métodos asociados a las listas:

```
In [15]: L = [1,2,3,2,2]
         print(L.count(2)) # para contar todas las incidencias del objeto 2 en la l
         M = L.copy() # para copiar la lista creando un objeto nuevo con el mismo c
         L.insert(3, 'hola') # para insertar un objeto antes del índice indicado.
         print(L)
         L.extend(M) # extiende L pegándole los objetos de M. M puede ser cualquier
         print(L)
         M.sort()
         print(M)
         print(L.index(2))
```

```
3
[1, 2, 3, 'hola', 2, 2]
[1, 2, 3, 'hola', 2, 2, 1, 2, 3, 2, 2]
[1, 2, 2, 2, 3]
1
```

Notamos una diferencia fundamental entre algunos de estos métodos: el método `list.copy()` crea un objeto nuevo, mientras que métodos como `list.sort()` o `list.insert(index, object)` **modifican la lista original**, a éste último tipo de métodos los solemos llamar métodos *in-place*.

4.4 Slicing

Slicing consiste en sacar ciertas partes de una lista. La sintaxis es la siguiente: `lista[inicio:fin:paso]`. Como siempre, nunca se toma el elemento final, entonces si queremos sacar por ejemplo el segmento inicial hasta la posición 4, debemos escribir `lista[0:5]` en vez de `lista[0:4]`. Por último, el paso siempre está predeterminado a ser 1 y podemos evitar escribir las posiciones iniciales y finales si éstas son el comienzo y el fin de la lista respectivamente.

```
In [16]: L = ['a','b','c','d']
         M = L[0:3]
         print(M)
         K = L[:3]
         print(K)
         G = L[-2:]
         print(G)
         L_invertida = L[::-1]
         print(L_invertida)

['a', 'b', 'c']
['a', 'b', 'c']
['c', 'd']
['d', 'c', 'b', 'a']
```

Todo objeto secuenciable puede ser sujeto a slicing, por ejemplo:

```
In [17]: nombre = 'Miguel'
         diminutivo = nombre[:-1]
         print(diminutivo)
         sucesion = (1,2,3,4,5,6,7,8)
         print(sucesion[3:-3])

Migue
(4, 5)
```

4.5 La función `enumerate`

Usualmente es útil recuperar el índice en el que está cierto elemento a la hora de hacer un ciclo for. Para hacerlo, usamos el método enumerate.

```
In [18]: print(enumerate(['a', 'b', 'c']))
         print(list(enumerate(['a', 'b', 'c'])))

<enumerate object at 0x7fefe85c8af8>
[(0, 'a'), (1, 'b'), (2, 'c')]
```

```
In [19]: lista_de_invitados = ['Iván', 'Daniel', 'Jorge']
         for indice, invitado in enumerate(lista_de_invitados):
             print(invitado + ' es el invitado número ' + str(indice))

Iván es el invitado número 0
Daniel es el invitado número 1
Jorge es el invitado número 2
```

5 Diccionarios

5.1 ¿Qué es un diccionario?

Un diccionario es un objeto en python que se compone de llaves (en inglés *keys*) y valores (*values*).

```
edades = {'Miguel': 22, 'Daniel': 25, 'Bruma': 9}
```

podemos pensar en `edades` como una *función conjuntista* que a la llave (o preimagen) `'Miguel'` le asocia el valor (o imagen) 22. Las llaves deben siempre ser objetos inmutables.

Diferente a las listas, los diccionarios no tienen ningún sentido del orden.

```
In [20]: edades = {'Miguel': 22, 'Daniel': 25, 'Bruma': 9}
         edades['Bruma']

Out[20]: 9
```

Podemos ingresar nuevas *parejas* en el diccionario de la siguiente forma:

```
In [21]: edades = {'Miguel': 22, 'Daniel': 25, 'Bruma': 9}
         edades['Diana'] = 49
         print(edades)

{'Daniel': 25, 'Miguel': 22, 'Bruma': 9, 'Diana': 49}
```

Podemos también modificar los valores de cierta llave:

```
In [22]: edades['Miguel'] += 1
         print(edades)
```

```
{'Daniel': 25, 'Miguel': 23, 'Bruma': 9, 'Diana': 49}
```

```
In [23]: print(edades.keys()) # es un iterable con las preimágenes
         print(edades.items()) # es un iterable con las parejas ordenadas
         print(edades.values()) # es un iterable con valores.
```

```
dict_keys(['Daniel', 'Miguel', 'Bruma', 'Diana'])
dict_items([('Daniel', 25), ('Miguel', 23), ('Bruma', 9), ('Diana', 49)])
dict_values([25, 23, 9, 49])
```

5.2 Diccionarios por extensión y por comprensión

Como con las listas, podemos definir diccionarios por comprensión.

```
In [24]: invitados = ['Miguel', 'Agatha', 'Bruma']
         diccionario_de_invitados = {invitado: indice for (indice, invitado) in enu
         print(diccionario_de_invitados)
```

```
{'Agatha': 1, 'Miguel': 0, 'Bruma': 2}
```

5.3 Ejercicio: diccionario inverso

1. Escriba una función que tome un diccionario y retorne True si éste es una función biyectiva y False en caso contrario.
2. Escriba una función que tome un diccionario biyectivo y devuelva el diccionario inverso.

```
In [25]: def esinvertible(diccionario):
             if len(set(diccionario.keys())) != len(diccionario):
                 return False

             if len(set(diccionario.values())) != len(diccionario):
                 return False

             return True
```

```
In [26]: print(esinvertible({1: 1, 2: 2, 3: 2}))
         print(esinvertible({1: 1, 2: 2, 3: 3}))
```

```
False
True
```

```
In [27]: def diccionarioinverso(diccionario):
             if esinvertible(diccionario):
                 return {value: key for (key, value) in diccionario.items()}
             else:
                 raise ValueError('diccionario no es invertible')
```

```
In [28]: diccionario_invertible = {1: 'a', 2: 'b', 3: 'c'}
         inverso = diccionarioinverso(diccionario_invertible)
         print(inverso)
```

```
{'b': 2, 'a': 1, 'c': 3}
```

5.4 Proyecto: encriptador básico usando python.

Construyamos un encriptador para mensajes usando diccionarios en python.

```
In [29]: alfabeto = 'abcdefghijklmnopqrstuvwxyz'
         valor_de_cambio = 2
         encriptador = {alfabeto[i]: alfabeto[(i+valor_de_cambio) % len(alfabeto)]
         print(encriptador)
         desencriptador = diccionarioinverso(encriptador)
         print(desencriptador)
```

```
{'j': 'l', 'y': 'a', 'z': 'b', 'u': 'w', 'a': 'c', 'q': 's', 'i': 'k', 'b': 'd', 'w
{'j': 'h', 'z': 'x', 'u': 's', 'a': 'y', 'q': 'o', 'i': 'g', 'b': 'z', 'w': 'u', 'h
```

```
In [30]: def encriptar(mensaje, alfabeto, offset):
             encriptador = {alfabeto[i]: alfabeto[(i+offset) % len(alfabeto)] for i
             desencriptador = diccionarioinverso(encriptador)
             mensaje_encriptado = ''
             for letra in mensaje:
                 mensaje_encriptado = mensaje_encriptado + encriptador[letra]

             return mensaje_encriptado

         def desencriptar(mensaje, alfabeto, offset):
             encriptador = {alfabeto[i]: alfabeto[(i+offset) % len(alfabeto)] for i
             desencriptador = diccionarioinverso(encriptador)
             mensaje_original = ''
             for letra in mensaje:
                 mensaje_original = mensaje_original + desencriptador[letra]

             return mensaje_original
```

```
In [31]: alfabeto = 'abcdefghijklmnopqrstuvwxyz'
         print(encriptar('hola', alfabeto, 5))
         print(desencriptar('mtqf', alfabeto, 5))
```

```
mtqf
hola
```

6 Ejercicios

6.1 Primer ejercicio

Escriba por compresión las siguientes listas:

1. Una lista con el cuadrado de todos los números en range(0,100).
2. Una lista con todas las letras de su nombre.

6.2 Segundo ejercicio

Cree una función que tome un string y devuelva un diccionario cuyas llaves son las letras del string y sus respectivos valores son la cantidad de veces que ellas aparecen en el string. Por ejemplo,

```
contador('esternocleidomastoideo')
{'e': 4, 's': 2, 't': 2, ...}
```

6.3 Tercer ejercicio

Cree una función que tome un string y que devuelva un diccionario que cuente las palabras y sus repeticiones bajo las siguientes especificaciones: 1. Todas las llaves deben estar en minúsuculas (es decir, 'Hola' y 'hola' deberían sumar al mismo contador). 2. Se deben ignorar los signos de puntuación (es decir, " '(),.;:?¡¿[]{}-' ").

Por ejemplo

```
contadorpalabras('Hola, mi nombre es Miguel y el nombre de mi gata es Agatha')
{'hola': 1, 'mi': 2, 'nombre': 2, 'es': 2, ...}
```

Pista: para este ejercicio vendría bien revisar los métodos `split` y `replace` en strings.

Clase 3 - IO

March 25, 2017

1 Introducción

En esta tercera clase hablamos sobre cómo leer y escribir archivos usando python. Pero primero realizaremos los ejercicios de la segunda clase.

2 Desarrollo de los ejercicios de la clase anterior

2.1 Primer ejercicio

```
In [1]: L1 = [i ** 2 for i in range(0,101)]
        L2 = [letra for letra in 'Miguel']

In [2]: print(L1)
        print(L2)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 3
['M', 'i', 'g', 'u', 'e', 'l']
```

2.2 Segundo ejercicio

```
In [3]: def contador(string):
            diccionario_de_letras = dict()
            for letra in string:
                if letra in diccionario_de_letras:
                    diccionario_de_letras[letra] += 1
                else:
                    diccionario_de_letras[letra] = 1
            return diccionario_de_letras

In [4]: contador('aaaababbA')

Out[4]: {'A': 1, 'a': 5, 'b': 3}
```

2.3 Tercer ejercicio

```
In [5]: def contadorpalabras(string):
            #Primero, limpiamos el string de acuerdo a las indicaciones.

            for conector in '(),.;:?¡¿![]{}-':
                string = string.replace(conector, '')
            string = string.lower()

            # Después, separamos las palabras usando .split
            listadepalabras = string.split(' ')

            # Por último, contamos
            diccionario_de_palabras = dict()
            for palabra in listadepalabras:
                if palabra in diccionario_de_palabras:
                    diccionario_de_palabras[palabra] += 1
                else:
                    diccionario_de_palabras[palabra] = 1
            return diccionario_de_palabras

In [6]: contadorpalabras('Hola, mi nombre es Miguel (y Miguel es también el nombre

Out[6]: {'colega': 1,
         'de': 1,
         'el': 1,
         'es': 2,
         'hola': 1,
         'mi': 1,
         'miguel': 2,
         'nombre': 2,
         'también': 1,
         'un': 1,
         'y': 1}
```

3 Algo más sobre imprimir strings

Es muy común que a la hora de escribir archivos o sacar resultados necesitemos manipular un string de tal forma que varíe dependiendo de una variable. Por ejemplo, supongamos que estamos sacando unas gráficas y que necesitamos cambiar su nombre de 'fig1' a 'fig2' a 'fig3' automáticamente. Para hacerlo, aprovechamos el método `format` para strings:

```
In [7]: for k in range(10):
            print('El número actual es {}'.format(k))

El número actual es 0
El número actual es 1
El número actual es 2
```

```
El número actual es 3
El número actual es 4
El número actual es 5
El número actual es 6
El número actual es 7
El número actual es 8
El número actual es 9
```

```
In [8]: lista = ['a', 'b', 'c', 'd']
        for elemento in lista:
            print('El elemento actual es {}'.format(elemento))
```

```
El elemento actual es a
El elemento actual es b
El elemento actual es c
El elemento actual es d
```

Es probable que necesitemos ir variando más de una variable, para eso adoptamos la siguiente notación:

```
In [9]: diccionario_de_edad = {'Miguel': 22, 'Diana': 49, 'Sara': 18}
        for persona in diccionario_de_edad:
            print('{llave} tiene {valor} años'.format(llave = persona, valor=diccio
```

```
Miguel tiene 22 años
Sara tiene 18 años
Diana tiene 49 años
```

Podemos agregarle más condiciones al formato adentro de las llaves siguiendo la siguiente convención:

- Usamos la letra 'd' para indicar que la variable es un entero con signo.
- Usamos la letra 'f' para indicar que la variable es un float.
- Usamos la letra 's' para indicar que la variable es un string.

Por ejemplo:

```
In [10]: from math import e
```

```
In [11]: print('La constante de Euler es aproximadamente igual a {constante:1.10f}'
```

```
La constante de Euler es aproximadamente igual a 2.7182818285
```

Podemos obligarlo a imprimir ceros adelante usando `05d`, por poner un ejemplo.

```
In [12]: for k in range(10):
             print('Estamos en la posición {variable:05d}'.format(variable = k))
```

```
Estamos en la posición 00000
Estamos en la posición 00001
Estamos en la posición 00002
Estamos en la posición 00003
Estamos en la posición 00004
Estamos en la posición 00005
Estamos en la posición 00006
Estamos en la posición 00007
Estamos en la posición 00008
Estamos en la posición 00009
```

4 Leer y escribir archivos de texto plano

Antes de leer y escribir archivos, vale la pena saber en qué directorio (i.e. carpeta) estamos. Para hacerlo usamos los comandos `pwd` (que significa "print working directory") y `cd` (que traduce "change directory"), de ser necesario.

```
In [13]: pwd
```

```
Out[13]: '/home/mgd/Dropbox/AGL/cursillo_python/Notebooks_clases'
```

```
In [14]: ls # Para conocer los archivos que hay en el directorio
```

```
Clase 1.ipynb  Clase 3 - Final.ipynb  textos/
Clase 2.ipynb  Clase 3.ipynb          Untitled.ipynb
```

```
In [15]: cd textos/
```

```
/home/mgd/Dropbox/AGL/cursillo_python/Notebooks_clases/textos
```

```
In [16]: pwd
```

```
Out[16]: '/home/mgd/Dropbox/AGL/cursillo_python/Notebooks_clases/textos'
```

```
In [17]: ls
```

```
ejemplo_csv   ejemplo_de_escritura_por_lineas.txt  ejemplo_de_lectura.txt
ejemplo_csv~  ejemplo_de_escritura.txt             tabla_en_latex.txt
```

4.1 Leer archivos

Para escribir y leer archivos usamos las función `open`:

```
In [18]: open?
```

```
In [19]: ejemplo = open('ejemplo_de_lectura.txt')
```

```
In [20]: print(ejemplo.read())
```

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
```

```
In [21]: print(ejemplo.read()) # Una vez "ejemplo" se lee, queda vacío.
```

```
In [22]: ejemplo = open('ejemplo_de_lectura.txt')
         lineas = ejemplo.readlines()
         for linea in lineas:
             print(linea)
```

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod

tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,

quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo

consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse

cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non

proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
```

notamos que los strings que leemos usando `read` o `readlines` traen ciertos caracteres especiales (por ejemplo `\n`). Éstos se llaman *literales del string*. Entre otros están:

- `\\` backslash.
- `\'` comilla simple.
- `\"` comilla doble.
- `\n` nueva línea.
- `\t` tabulación.
- `\r` retorno de carro.

Por ejemplo:

```
In [23]: print('Hola \t Mundo')
```

```
Hola	 Mundo
```

Usualmente (y a la hora de procesar archivos) necesitamos quitar estos caracteres especiales. Para hacerlo usamos el método `replace` para strings.

4.2 Proyecto: encontrar la letra más común en un libro

Supongamos que tenemos un archivo `libro.txt` que contiene todo un libro y necesitamos saber cuál es la letra más común que aparece en él. Escribamos una función que haga precisamente eso.

Para hacerlo, dividimos la tarea en varias partes:

1. Necesitamos abrir el archivo y extraer el string con todo el contenido.
2. Necesitamos limpiar este string, quitando los literales del string.
3. Necesitamos contar la cantidad de letras y almacenar los resultados en alguna variable.
4. Necesitamos devolver el resultado al usuario.

```
In [32]: ls
```

```
DUBLINERS.txt                    THETRAGEDYOFROMEOANDJULIET.txt
THEBROTHERSGRIMMFAIRYTALES.txt   THUSSPAKEZARATHUSTRA.txt
THETIMEMACHINE.txt
```

```
In [33]: def contadordeletras(path_del_libro):
             # Primero abrimos el archivo y almacenamos el string
             libro_archivo = open(path_del_libro)
             libro = libro_archivo.read()

             # Limpiamos el string
             libro = libro.replace('\n', '')
             libro = libro.replace('\r', '')
             libro = libro.lower()

             # Contamos las letras
             diccionario_de_letras = dict()
             for letra in libro:
                 if letra.isalpha():
                     if letra in diccionario_de_letras:
                         diccionario_de_letras[letra] += 1
                     else:
                         diccionario_de_letras[letra] = 1

             # Devolvemos el resultado
             return diccionario_de_letras
```

Ya tenemos implementada la función. Intentémosla en *Así habló Zarathustra* de F. Nietzsche (en inglés)

```
In [34]: path = 'THUSSPAKEZARATHUSTRA.txt'
         print(contadordeletras(path))
```

```
{'y': 8792, 's': 24889, 'r': 21721, 'u': 12035, 'i': 25735, 'p': 5774, 'f': 7847, '
```

Sería muy interesante poder organizar las llaves de un diccionario por su valor. Después de una breve búsqueda en internet, notamos que podemos usar la librería `operator` y el siguiente pedazo de código:

```
In [35]: import operator

         diccionario = contadordeletras(path)

         lista_ordenada = sorted(diccionario.items(), key = operator.itemgetter(1))
         lista_ordenada.reverse()
         print(lista_ordenada)
```

```
[('e', 50291), ('t', 39313), ('a', 30973), ('o', 29789), ('h', 29357), ('n', 25885)
```

En *Así habló Zarathustra* aparece más la 't' que la 'a'.

4.3 Escribir archivos

```
In [57]: open?
```

Como podemos ver al preguntarle a IPython sobre la función `open`, podemos escribir sobre archivos dependiendo de la bandera que le pasemos:

- `'w'` para escribir sobre el archivo. Usar `'w'` resulta en sobreescribir.
- `'x'` para crear un archivo nuevo y abrirlo.
- `'a'` para escribir al final de un archivo existente.
- `'r+'` para tanto leer como escribir

```
In [36]: archivo_nuevo = open('ejemplo_de_escritura.txt', 'x')
```

Usando el método write, podemos escribir en el archivo usando el método `write`:

```
In [37]: archivo_nuevo.write('Hola mundo')

Out[37]: 10

In [38]: archivo_nuevo.close()
```

Análogamente, podemos escribir archivos usando `writelines`:

```
In [39]: segundo_archivo = open('ejemplo_de_escritura_por_lineas.txt', 'x')
         lista_de_filas = ['Hola, mundo\n', 'mi nombre es Miguel']
         segundo_archivo.writelines(lista_de_filas)
         segundo_archivo.close()

In [40]: sobre_escribir = open('ejemplo_de_escritura.txt', 'w')
         sobre_escribir.write('Aquí estoy sobreescribiendo lo que estuviera en el a
         sobre_escribir.close()

In [41]: no_sobre_escribir = open('ejemplo_de_escritura.txt', 'a')
         no_sobre_escribir.write('\nPero ahora no.')
         no_sobre_escribir.close()
```

4.4 Una froma más eficiente de leer el archivo

Para evitar tener que cerrar el archivo una vez terminamos con él, solemos usar la palabra clave `with`:

```
with open('archivo.txt') as archivo:
    # ...
```

5 Archivos CSV

Archivos CSV (comma separated values) son bastante comunes: pueden representar tablas, matrices... Podemos crear un archivo csv desde excel, por ejemplo.

Existe una librería de python para procesar archivos csv, y se llama precisamente `csv`:

Para usar los métodos de la librería `csv`, necesitamos un archivo de tipo io (como los que hemos estado importando).

```
In [50]: ls

ejemplo_csv   ejemplo_de_escritura_por_lineas.txt  ejemplo_de_lectura.txt
ejemplo_csv~  ejemplo_de_escritura.txt             tabla_en_latex.txt


In [51]: import csv

In [52]: with open('ejemplo_csv') as ejemplo:
             lista = csv.reader(ejemplo, delimiter=' ')
             print(list(lista))
             print(list(lista)) # Como siempre, se puede leer una sola vez nada más

[['1', '2', '3'], ['4', '5', '6'], ['7', '8', '9']]
[]
```

el objeto creado por `csv.reader` se colapsa a una lista de listas, en donde cada lista interna representa una fila del archivo original.

5.1 Proyecto: pasar de csv a tabla de LaTeX

Escribamos un script en python que nos permita obtener información sobre una tabla en csv para posteriormente escribir un archivo de texto plano con cómo luciría la tabla escrita en LaTeX. Recordamos que las tablas en LaTeX se hacen separando cada columna por un & y terminando la fila con un doble backslash. Para hacer esto, seguimos los siguientes pasos:

1. Comenzamos ingresando el archivo como un objeto io con `with open(path_del_archivo) as archivo:`.
2. Creamos un archivo nuevo para escribir sobre él con `open(path_del_archivo_nuevo, 'x')`.
3. Iteramos sobre cada fila del archivo original, operamos el contenido y escribimos sobre el archivo nuevo.

```
In [58]: import csv
         def tablaLaTeX(path_del_archivo, delimitador=';'): # excel por defecto.
             with open(path_del_archivo) as archivo:
                 try:
                     nuevo_archivo = open('tabla_en_latex.txt', 'x')
                 except:
                     nuevo_archivo = open('tabla_en_latex.txt', 'w')
                 for fila in csv.reader(archivo, delimiter = delimitador):
                     for k, elemento in enumerate(fila):
                         # Si el elemento no es final, lo escribimios elemento &
                         if k != len(fila) - 1:
                             nuevo_archivo.write(str(elemento) + '&')
                         else:
                             nuevo_archivo.write(str(elemento) + r'\\' + '\n')
                 nuevo_archivo.close()

In [59]: tablaLaTeX('ejemplo_csv', ' ')
```

6 Ejercicio

El ejercicio de la clase de hoy consiste en usar la función `contadorpalabras` en alguno de los textos que se analizaron en clase. Para hacerlo, debemos reimplementarla para sustituir ciertos caracteres especiales.

6.1 Primer ejercicio

Escriba una función `limpiador(texto)` que tome un string y elimine de él los siguientes literales del string y caracteres de unicode:

- `\n`
- `\r`
- `\u2018` (comilla simple izquierda)
- `\u2019` (comilla simple derecha)
- `\u0027` (apóstrofe)
- `\u201C` (comilla doble izquierda)
- `\u201D` (comilla doble derecha)
- Todos los signos de puntuación (al menos los que vienen de la librería `string`).

y, por último, retorne el string todo en minúsculas.

6.2 Segundo ejercicio

Escriba una función `contador_de_palabras(path_del_archivo)` que tome solo el *path* al archivo .txt y que realice lo siguiente:

1. Extraiga el contenido del archivo usando `open` y `read`.
2. Lo limpie usando la función auxiliar `limpiador` que escribimos en el primer ejercicio.
3. Cuente las palabras y almacene los resultados en un diccionario.

4. Genere una lista en donde se organicen los ítems del diccionario por sus valores (usando la librería `operator`, por ejemplo)
5. Devuelva ésta lista (o el reverso de ésta, para que estén de mayor a menor)

6.3 Tercer ejercicio

La ley de Zipf afirma que si en un texto la palabra más común ocurre n veces, la segunda palabra más común aparecerá $n/2$ veces, la tercera palabra más común aparecerá $n/3$ veces y así. ¿Qué tanto cumplen la ley de Zipf los 5 libros que descargamos?

Clase 4 - Numpy y Matplotlib

March 25, 2017

1 Introducción

En la clase de hoy trabajaremos con los paquetes de cálculo científico y graficación `numpy` y `matplotlib`, ambos miembros de la *suite* scipy. Con estos dos paquetes (junto con `scipy.linalg`) podemos reemplazar muchas de las utilidades que programas como Matlab nos dan: graficación, cálculos numéricos con matrices...

```
In [2]: import numpy as np
```

2 Numpy

2.1 Arreglos de Numpy.

Así como los arreglos y las matrices son el objeto esencial en matlab, los `numpy.array` y las `numpy.matrix` son los objetos esenciales para los cálculos en Numpy. Una de las diferencias principales entre los arreglos de numpy y las listas es que, mientras las listas pueden almacenar objetos de diferentes tipos, los arreglos solo admiten elementos de un tipo a la vez (es decir, no se pueden tener arreglos como [1, False, [1,2,3]].

```
In [3]: X = np.array([1,2,3])
        print(X)
        A = np.matrix('1,2; 3,4')
        print(A)
```

```
[1 2 3]
[[1 2]
 [3 4]]
```

Los arreglos y las matrices se pueden sumar, y ahora la suma no representa concatenación sino suma componente a componente:

```
In [4]: X = np.array([1,2,3])
        Y = np.array([2,4,6])
        print(X + Y)
```

```
[3 6 9]
```

```
In [5]: A = np.matrix([[1,2], [3,4]])
        B = np.matrix('0, 1; 1, 1')
        print(A + B)
        print(type(A+B))

[[1 3]
 [4 5]]
<class 'numpy.matrixlib.defmatrix.matrix'>
```

2.2 Funciones para crear arreglos

A la hora de crear arreglos grandes, podemos recurrir a las siguientes funciones (que resultan ser muy parecidas a las usadas en Matlab):

- `numpy.arange(inicio, fin, paso)`
- `numpy.linspace(inicio, fin, cantidad)`, para crear un arreglo con puntos distribuidos de forma lineal entre `inicio` y `fin`.

```
In [6]: np.arange(1,10)

Out[6]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

In [7]: np.linspace(1,10,5)

Out[7]: array([  1.  ,   3.25,   5.5 ,   7.75,  10.  ])
```

A veces se necesitan arreglos con números aleatorios, para eso podemos usar - `numpy.random.random(tamaño)` para obtener números entre 0 y 1 escogidos de forma uniforme al azar. - `numpy.random.randint(menor, mayor, tamaño))` para obtener números enteros entre el menor y el mayor - 1.

```
In [8]: np.random.random(10)

Out[8]: array([ 0.65331648,  0.47867165,  0.20051284,  0.80913836,  0.73259759,
                0.132641  ,  0.55043593,  0.14667835,  0.57072164,  0.09161002])

In [9]: np.random.random((2,2))

Out[9]: array([[ 0.00090092,  0.22721791],
               [ 0.8447204 ,  0.35449551]])

In [10]: np.random.randint(1, 10, 10)

Out[10]: array([8, 2, 3, 3, 3, 1, 8, 6, 8, 7])

In [11]: np.random.randint(1, 10, (2,2))

Out[11]: array([[5, 3],
                [7, 4]])
```

2.3 Slicing de arreglos

Además del slicing usual para listas, Numpy nos permite hacer slicing de los arreglos de una forma más inteligente.

```
In [12]: X = np.array([1,2,3])
         print(X[:2])
         print(X % 2 == 0)

[1 2]
[False  True False]
```

Podemos sacar de un arreglo todos los valores que cumplan cierto valor. Por ejemplo:

```
In [13]: X[X % 2 == 0]

Out[13]: array([2])
```

Es decir, primero sacamos una *máscara* (en este caso `X % 2 == 0`) que consiste en un arreglo booleano (en este caso `[False True False]`). Al pasarle esta máscara a X estamos pidiendo que devuelva solamente los valores `True`.

2.4 Ejercicio: estimar π usando números aleatorios

Podemos estimar π usando solo números enteros: la probabilidad de que dos número enteros seleccionados al azar sean primos relativos (i.e. su máximo común divisor sea 1) es $6/\pi^2$. Usando este hecho podemos estimar π de la siguiente forma:

2.4.1 1. Implementamos una función máximo común divisor:

Usando el algoritmo de euclides:

```
In [14]: def gcd(a,b):
             while b != 0:
                 a, b = b, a%b
             return a
```

Podríamos también usar una que ya esté implementada, como `math.gcd`.

2.4.2 2. Creamos arreglos con números aleatorios enteros de tamaño n.

```
In [15]: n = 1000
         X = np.random.randint(1, 1000, n)
         Y = np.random.randint(1, 1000, n)
```

2.4.3 3. Iteramos sobre ambos, y contamos cuántos son primos relativos usando un diccionario.

```
In [16]: diccionario_de_freq = {'primos relativos': 0, 'no primos relativos': 0}
         for k in range(n):
             if gcd(X[k], Y[k]) == 1:
                 diccionario_de_freq['primos relativos'] += 1
             else:
                 diccionario_de_freq['no primos relativos'] += 1
```

```
In [17]: print(diccionario_de_freq)
```

```
{'primos relativos': 637, 'no primos relativos': 363}
```

2.4.4 4. Calculamos la probabilidad usando la regla de Laplace.

```
In [18]: prob = diccionario_de_freq['primos relativos']/(diccionario_de_freq['primo
         print(prob)
```

```
0.637
```

si la probabilidad p es igual a $6/\pi^2$, despejamos y llegamos a que $\pi = \sqrt{6/p}$

```
In [19]: aprox_pi = np.sqrt(6/prob)
         print(aprox_pi)
```

```
3.06906374588
```

2.4.5 5. Podemos juntar todo en una función:

```
In [20]: def aproximar_pi(n):
             X = np.random.randint(1, 1000, n)
             Y = np.random.randint(1, 1000, n)
             dict_de_freq = {'pr': 0, 'npr': 0}
             for k in range(n):
                 if gcd(X[k], Y[k]) == 1:
                     dict_de_freq['pr'] += 1
                 else:
                     dict_de_freq['npr'] += 1
             prob = dict_de_freq['pr']/n
             return np.sqrt(6/prob)
```

```
In [21]: aproximar_pi(1000000)
```

```
Out[21]: 3.1379302549752111
```

3 Matplotlib

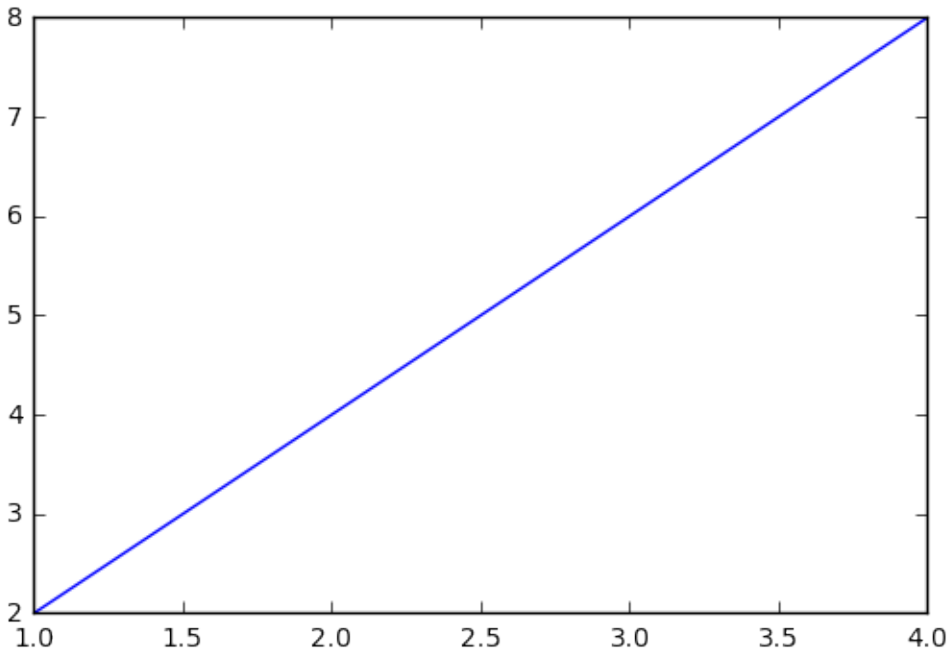
Mientras que `numpy` sirve para hacer los cálculos numéricos, la librería `matplotlib` sirve para realizar gráficos. La forma usual de importarla es la siguiente:

```
In [22]: import matplotlib.pyplot as plt
```

Podemos graficar objetos en listas o arreglos (o, en general, secuenciables numéricos):

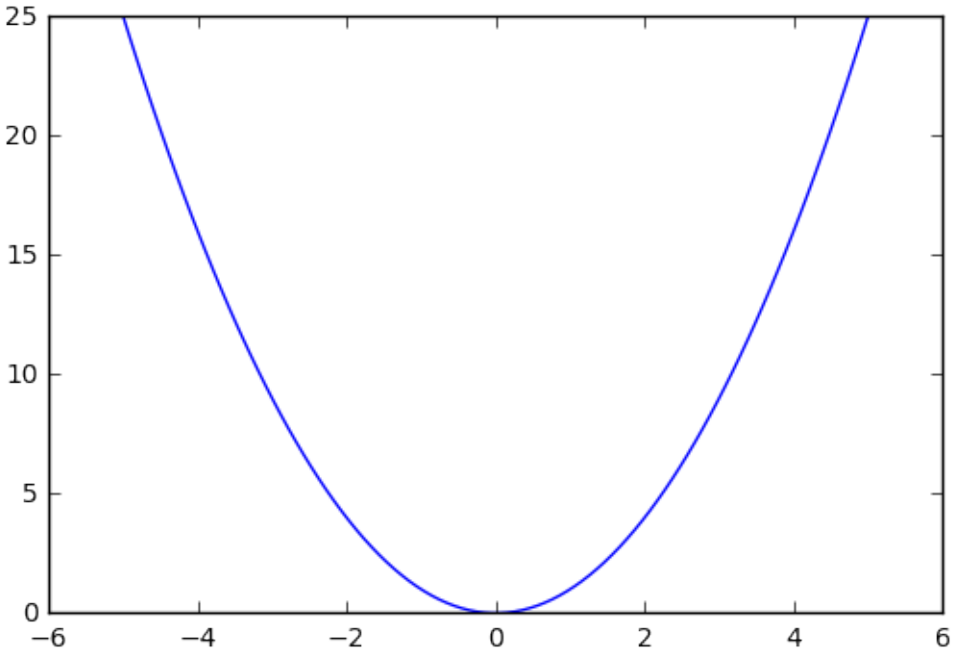
3.1 `plt.plot` y `plt.figure`

```
In [23]: plt.plot([1,2,3,4], [2,4,6,8])
         plt.show()
```



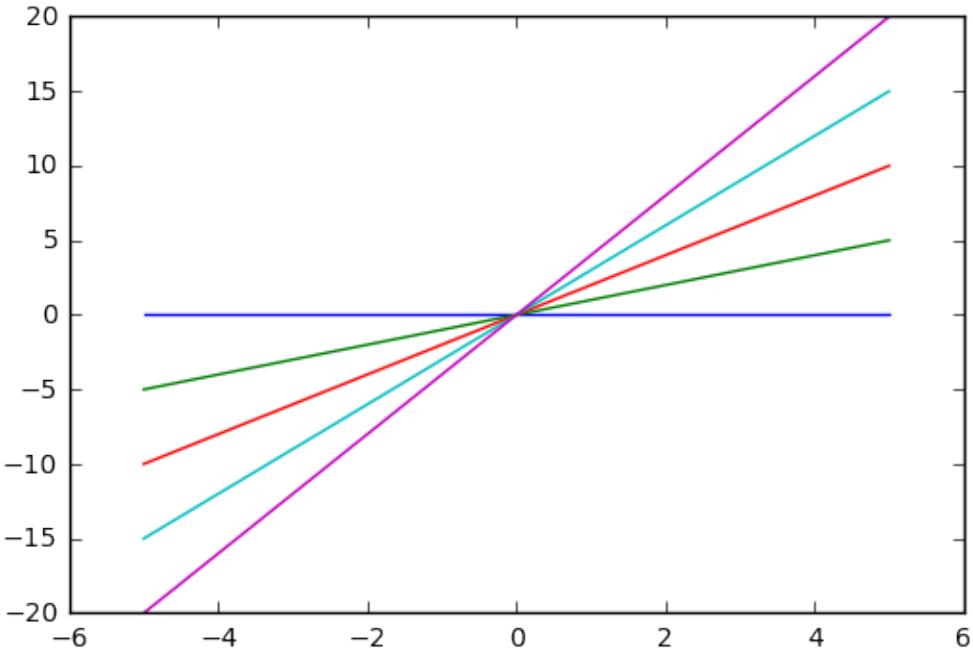
Podemos aprovechar la manipulación de arreglos de numpy para hacer funciones más elaboradas:

```
In [24]: X = np.linspace(-5, 5, 100)
         Y = X ** 2
         plt.plot(X,Y)
         plt.show()
```



Para dibujar más de una curva en una misma figura, iniciamos la figura usando `plt.figure()` y graficamos usando `plt.plot(...)`.

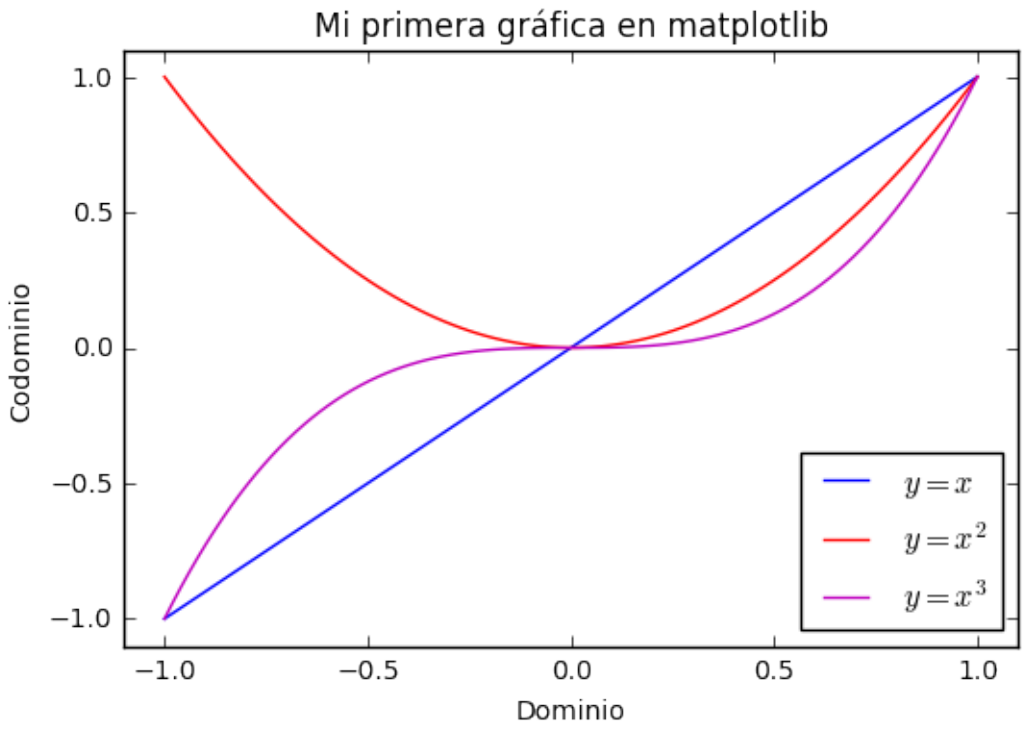
```
In [25]: X = np.linspace(-5, 5, 100)
         plt.figure()
         for k in range(5):
             plt.plot(X, k*X)
         plt.show()
         plt.close() # es una buena práctica cerrar las figuras.
```



3.2 Más características

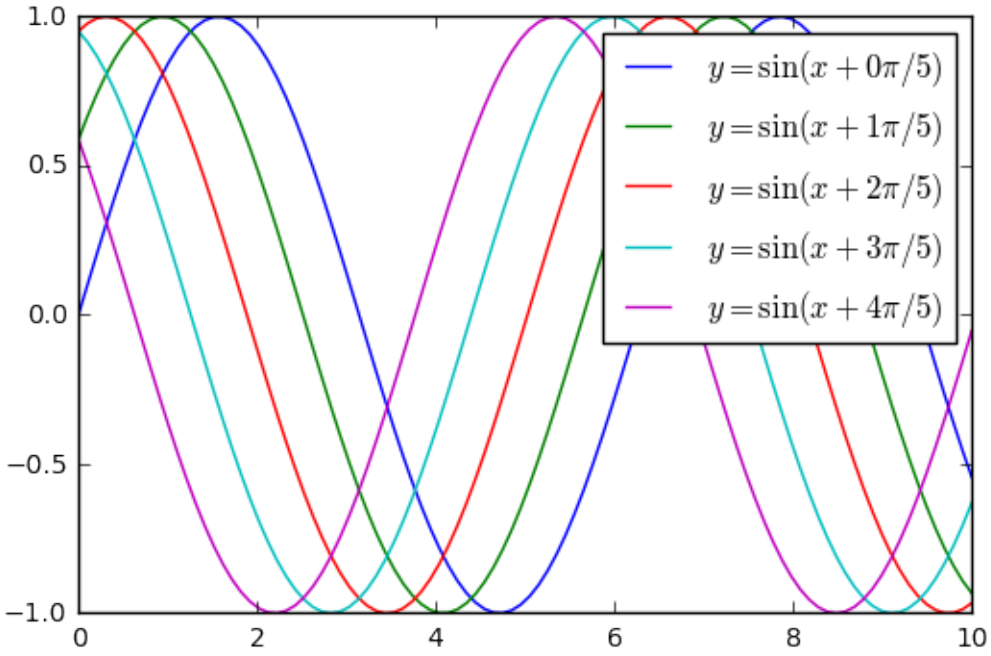
Podemos añadirle más características a las gráficas que realizamos, como título, leyenda, etiquetas para los ejes, límites para los ejes, color de relleno entre los ejes... por ejemplo:

```
In [26]: X = np.linspace(-1, 1, 100)
         plt.figure()
         plt.plot(X, X, 'b', label=r'$y = x$') # r de raw
         plt.plot(X, X ** 2, 'r', label=r'$y = x^2$')
         plt.plot(X, X ** 3, 'm', label=r'$y = x^3$')
         plt.legend(loc='best')
         plt.title('Mi primera gráfica en matplotlib')
         plt.xlim([-1.1, 1.1])
         plt.ylim([-1.1, 1.1])
         plt.xlabel('Dominio')
         plt.ylabel('Codominio')
         plt.show()
         plt.close()
```



Podríamos combinar graficación con nuestro conocimiento sobre formateo de strings:

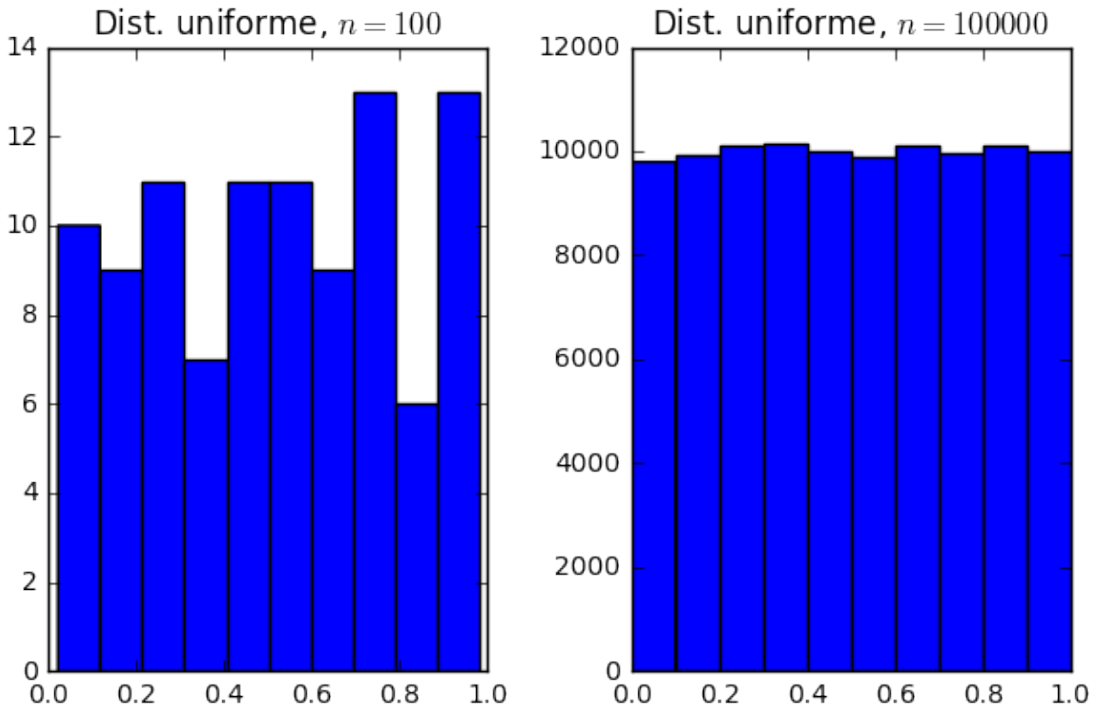
```
In [27]: X = np.linspace(0, 10, 100)
         plt.figure()
         for k in range(5):
             plt.plot(X, np.sin(X + k*np.pi/5), label=r'$y = \sin(x + {}\pi/5)$'.fo
         plt.legend(loc='upper right')
         plt.xlim([0,10])
         plt.show()
         plt.close()
```



3.3 Subplots y guardar imágenes

Además, y como en matlab, podemos crear subplots:

```
In [28]: X1 = np.random.random(100)
         X2 = np.random.random(100000)
         plt.figure()
         plt.subplot(121) #filas, columnas y posición
         plt.hist(X1)
         plt.title(r'Dist. uniforme, $n=100$')
         plt.subplot(122)
         plt.hist(X2)
         plt.title(r'Dist. uniforme, $n=100000$')
         plt.tight_layout() # para que se vea más lindo.
         plt.show()
         plt.close()
```



Por último, podemos guardar las imágenes que creamos usando `plt.savefig(nombre, formato, dpi,...)`

```
In [29]: X1 = np.random.normal(0, 1, 100000)
         X2 = np.random.gamma(1, 1, (100000,))
         plt.figure()
         plt.subplot(121)
         plt.hist(X1)
         plt.title('Dist. Normal')
         plt.subplot(122)
         plt.hist(X2)
         plt.title('Dist. Gamma')
         plt.tight_layout()
         plt.savefig('distribuciones.png', format='png')
         plt.close()
```

3.4 Ejercicio: hacer un diagrama de barras con un conteo de palabras

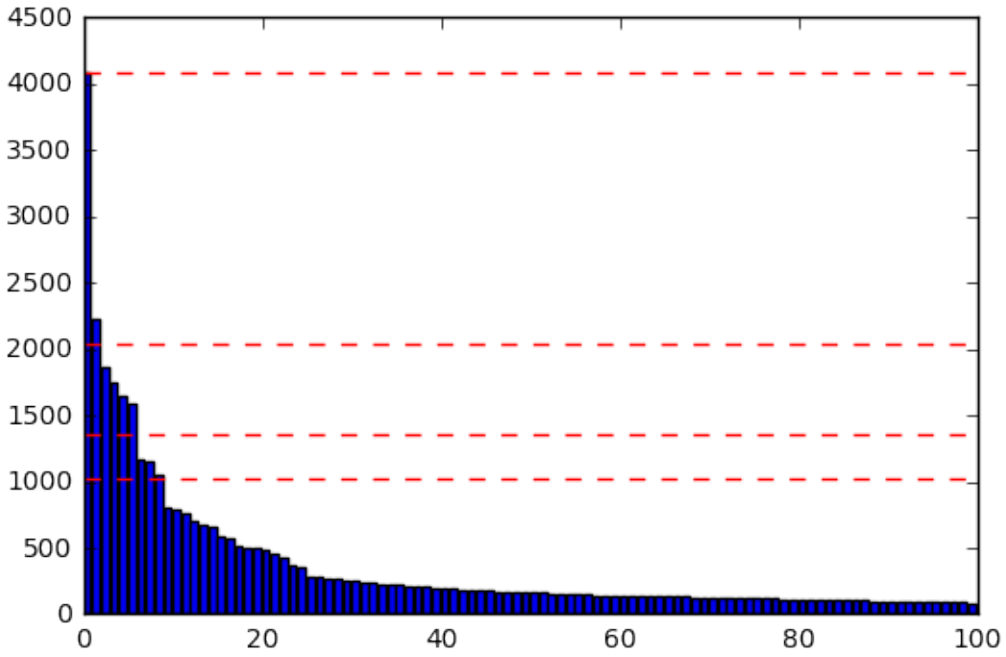
Como vimos en los ejercicios de la clase anterior, la distribución de frecuencias de palabras en textos parece obedecer la llamada "ley de Zipf", que afirma que la distribución se comporta muy parecido a una distribución de Pareto.

```
In [41]: import csv
```

```
In [42]: with open('tabla_palabras_DUBLINERS.csv') as dublinenses_archivo:
             dublinenses_csv = csv.reader(dublinenses_archivo, delimiter = ',')
```

```
            lista_de_palabras = []
            lista_de_valores = []
            for fila in dublinenses_csv:
                lista_de_palabras.append(fila[0])
                lista_de_valores.append(int(fila[1]))
```

```
In [43]: biggest_value = lista_de_valores[0]
         plt.bar(range(len(lista_de_palabras[:100])), lista_de_valores[:100])
         plt.axhline(biggest_value, color='red', ls='--')
         plt.axhline(biggest_value/2, color='red', ls='--')
         plt.axhline(biggest_value/3, color='red', ls='--')
         plt.axhline(biggest_value/4, color='red', ls='--')
         plt.show()
```



Con esto, tenemos lo suficiente para crear una función que guarde la imagen del análisis de Zipf:

```
In [44]: def imagenZipf(path_del_archivo_csv, titulo='Análisis de Zipf', nombre_arc
             with open(path_del_archivo_csv) as archivo:
                 archivo_csv = csv.reader(archivo, delimiter = ',')
                 lista_de_palabras = []
                 lista_de_valores = []
                 for fila in archivo_csv:
                     lista_de_palabras.append(fila[0])
                     lista_de_valores.append(int(fila[1]))
             valor_mas_grande = lista_de_valores[0]
```

```
        plt.bar(range(len(lista_de_palabras[:100])), lista_de_valores[:100])
        plt.axhline(valor_mas_grande, color='red', ls='--')
        plt.axhline(valor_mas_grande/2, color='red', ls='--')
        plt.axhline(valor_mas_grande/3, color='red', ls='--')
        plt.axhline(valor_mas_grande/4, color='red', ls='--')
        plt.title(titulo)
        plt.xlabel('Índice de la palabra')
        plt.ylabel('Frecuencia')
        plt.savefig(nombre_archivo, format='png')
        plt.close()

In [45]: imagenZipf('tabla_palabras_THUSSPAKEZARATHUSTRA.csv', 'Análisis de Zipf -T
```

Clase 5 - Sympy

March 25, 2017

1 Introducción

En esta última clase haremos una breve revisión de las posibilidades de cálculo simbólico que ofrece la librería `sympy`. Combinaremos estas posibilidades con el poder de graficación que viene con `numpy` y con `matplotlib`.

Importamos `sympy` con la siguiente convención:

```
In [2]: import sympy as sym
```

`Sympy` nos permite modificar la impresión de los resultados. Como estaremos trabajando con polinomios y funciones, recomendaría usar el siguiente comando:

```
In [3]: sym.init_printing(use_latex=True)
```

2 Manejo de expresiones en `sympy`

2.1 Símbolos

Las moléculas de `sympy` (es decir, los objetos más comunes y sobre los cuales se opera) son los símbolos.

```
In [5]: x = sym.Symbol('x')
        f = x**2 - 1
```

2.2 Simplificación

La función `sympy.simplify` simplifica expresiones simbólicas: cancela términos que se puedan cancelar, aplica identidades trigonométricas, entre otros.

```
In [9]: g = sym.simplify((x**2 + 2*x + 1)/(x+1)**2)
        print(g)
```

```
1
```

```
In [8]: sym.simplify(sym.cos(x)**2 + sym.sin(x)**2)
```

Out[8]:

$$1$$

Si no simplificamos, `sympy` toma lo que le pasemos de forma literal.

```
In [7]: h = (x**2 + 2*x + 1)/(x+1)**2
        print(h)
```

```
(x**2 + 2*x + 1)/(x + 1)**2
```

2.3 Factorización y expansión

Si tenemos una expresión, podemos factorizarla o expandirla usando `sympy.factor` y `sympy.expand`:

```
In [10]: f = x**2 - 1
```

```
In [11]: sym.factor(f)
```

Out[11]:

$$(x-1)(x+1)$$

```
In [12]: g = (x+1)*(x-2)**2
```

```
In [13]: sym.expand(g)
```

Out[13]:

$$x^3 - 3x^2 + 4$$

```
In [14]: h = sym.exp(x)*x**2 + sym.exp(2*x)
         sym.factor(h)
```

Out[14]:

$$\left(x^2 + e^x\right) e^x$$

2.4 Evaluación

Cuando tenemos una expresión podemos evaluarla o sustituir una variable por un valor.

```
In [15]: raizdetres = sym.sqrt(3)
         raizdetres
```

Out[15]:

$$\sqrt{3}$$

In [16]:
```
raizdetres.evalf()
```

Out[16]:

$$1.73205080756888$$

In [17]:
```
f = x**2 + 1
f.subs(x, 3)
```

Out[17]:

$$10$$

Podemos sustituir más de una variable, pero la sustitución no se hace de forma simultánea por defecto. Si necesitamos que la sustitución sea simultánea, pasamos el argumento `simultaneous=True`.

In [18]:
```
y = sym.Symbol('y')
```

In [19]:
```
g = x*y + x**2 - 1
print(g.subs([(x,y), (y,3)]))
print(g.subs([(x,y), (y,3)], simultaneous=True))
```

```
17
y**2 + 3*y - 1
```

3 Cálculo diferencial usando `sympy`

3.1 Derivadas

Con `sympy` podemos derivar (parcialmente) expresiones usando el comando `diff`:

In [20]:
```
f = x**2*sym.exp(x)
```

In [22]:
```
f.diff()
```

Out[22]:

$$x^2 e^x + 2xe^x$$

In [23]:
```
f.diff(x,2)
```

Out[23]:

$$\left(x^2 + 4x + 2\right) e^x$$

Podemos especificar con respecto a qué variable queremos derivar, también:

In [24]:
```
g = x*y**2 + 2*x*y
```

In [26]:
```
g.diff(y)
```

Out[26]:

$$2xy + 2x$$

3.2 Ejercicio: Aproximando funciones con polinomios usando el teorema de Taylor.

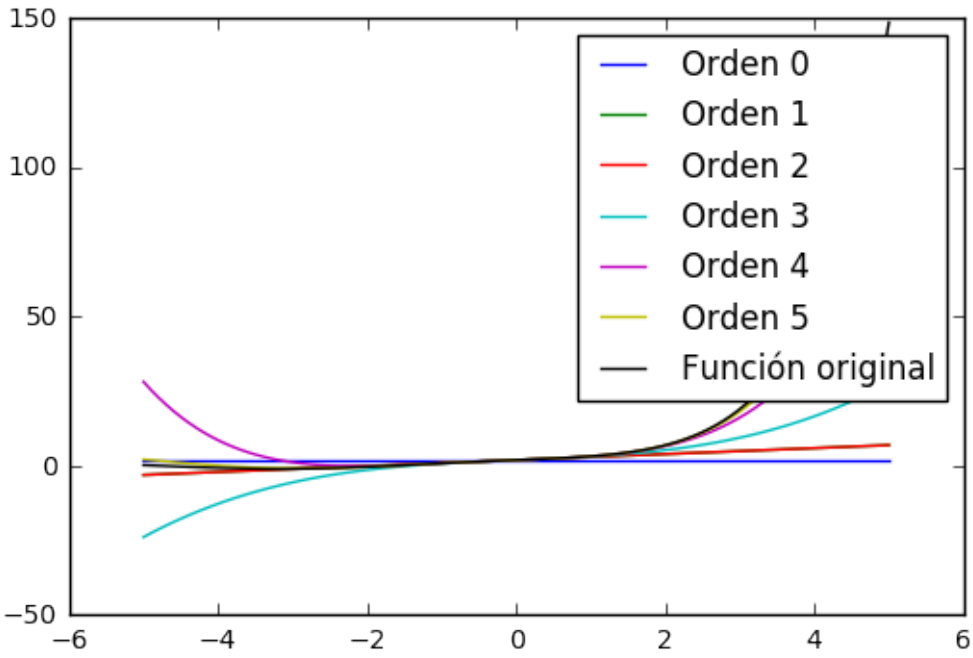
Recordemos que una función analítica se puede escribir como su serie de Taylor. Si una función f es analítica,

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \cdots$$

Implementemos una función que aproxime una función a través de su polinomio de Taylor hasta cierto grado n en cierto punto $a \in \mathbb{R}$, y que grafique el resultado y el polinomio para cada grado en un intervalo centrado en a.

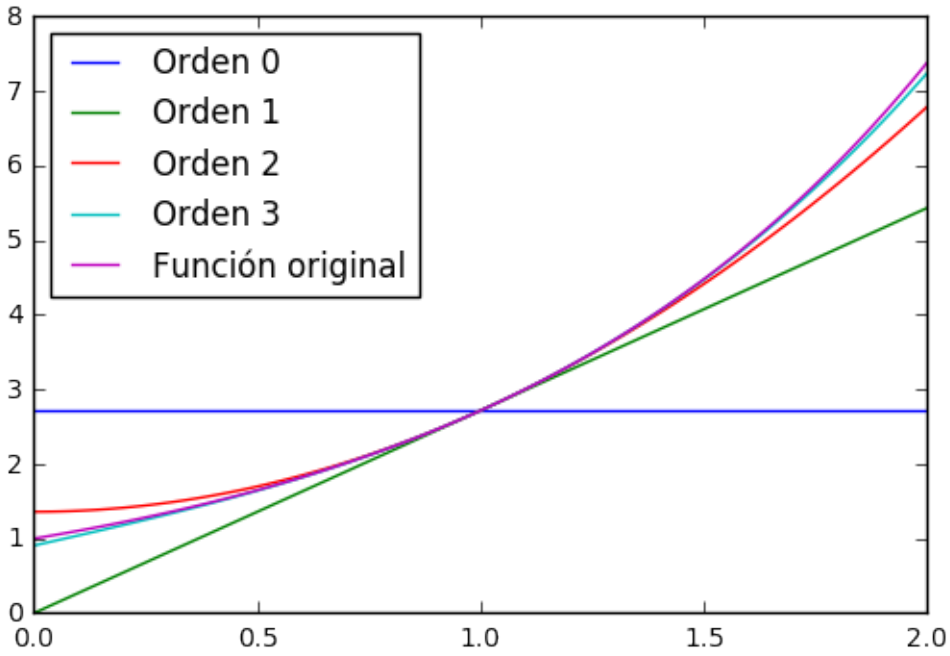
Dividamos esta tarea en subtareas: 1. Dado cierto orden $n \in \mathbb{N}$ y una función f, construir una lista con todas las derivadas de f (desde 0 hasta n). 2. Iteramos sobre esta lista, evaluando y sumando a un acumulable. En cada iteración, graficamos el polinomio aproximante. 3. Graficamos la función original.

```
In [27]: # 1.
         f = sym.exp(x) + sym.cos(x)
         n = 5
         a = 0
         lista_de_derivadas = [f.diff(x,k) for k in range(n+1)]

In [28]: from math import factorial

In [29]: import numpy as np
         import matplotlib.pyplot as plt

In [30]: dominio = np.linspace(0,1,100)
         Y = [f.subs(x, valor) for valor in dominio]

In [31]: # 2.
         polinomio = 0
         dominio = np.linspace(a-5, a+5, 100)
         plt.figure()
         for k, derivada in enumerate(lista_de_derivadas):
             polinomio += (derivada.subs(x, a)/factorial(k))*(x-a)**k
             Y = [polinomio.subs(x, valor) for valor in dominio]
             plt.plot(dominio, Y, label='Orden {}'.format(k))

In [32]: Yoriginal = [f.subs(x, valor) for valor in dominio]
         plt.plot(dominio, Yoriginal, label='Función original')
         plt.legend()

Out[32]: <matplotlib.legend.Legend at 0x7f04cef6c978>

In [33]: plt.show()
```



Con todo esto, podemos juntar los resultados en una función y podemos pulir la gráfica cerrando más el dominio y acotando lo que aparece en el eje y:

```
In [34]: def taylorаprox(f, a, n, legendflag = True):
             lista_de_derivadas = [f.diff(x,k) for k in range(n+1)]
             polinomio = 0
             dominio = np.linspace(a-1, a+1, 100)
             Yoriginal = [f.subs(x, valor) for valor in dominio]
             minval = min(Yoriginal)
             maxval = max(Yoriginal)
             plt.figure()
             for k, derivada in enumerate(lista_de_derivadas):
                 polinomio += (derivada.subs(x, a)/factorial(k))*(x-a)**k
                 Y = [polinomio.subs(x, valor) for valor in dominio]
                 plt.plot(dominio, Y, label='Orden {}'.format(k))
             plt.plot(dominio, Yoriginal, label='Función original')
             plt.ylim(int(minval)-1, int(maxval)+1)
             if legendflag:
                 plt.legend(loc='best')
             plt.show()
             return polinomio

In [35]: taylorаprox(sym.exp(x), 1, 3)
```



Out[35]:

$$\frac{e}{6}(x-1)^3 + \frac{e}{2}(x-1)^2 + e(x-1) + e$$

3.3 Integración

Así como derivadas, `sympy` permite integrar con `integrate`:

```
In [36]: f = sym.cos(x)
```

```
In [37]: f.integrate()
```

Out[37]:

$$\sin(x)$$

Podemos hacer también integrales definidas:

```
In [38]: f.integrate((x,0, sym.pi/2))
```

Out[38]:

$$1$$

4 Ecuaciones lineales y no lineales en sympy

Si quieremos escribir una ecuación en sympy usamos `sympy.Eq`:

```
In [39]: ecuacion = sym.Eq(x**2, 1)

In [40]: ecuacion
```

Out[40]:

$$x^2 = 1$$

Para solucionar una ecuación, usamos los comandos `sympy.solve` y `sympy.solveset`.

```
In [41]: sym.solve(ecuacion)
```

Out[41]:

$$[-1, \quad 1]$$

```
In [42]: ecuacion2 = sym.Eq(x-1)

In [43]: sym.solveset(ecuacion, x)
```

Out[43]:

$$\{-1, 1\}$$

5 Matrices en sympy

`sympy` tiene la clase `sympy.Matrix` y, con ésta, muchos métodos asociados.

```
In [44]: M = sym.Matrix([(0,1),(1,0)])

In [45]: M
```

Out[45]:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

```
In [46]: M.diagonalize() # Saca P y D tales que M = P^-1 * D * P con D diagonal.
```

Out[46]:

$$\left(\begin{bmatrix} -1 & 1 \\ 1 & 1 \end{bmatrix}, \quad \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \right)$$

```
In [47]: M.det() # Determinante
```

```
Out[47]:
```

$$-1$$

```
In [48]: M.T # Transpuesta
```

```
Out[48]:
```

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

```
In [49]: M[0,1] # Entrar a los elementos (empezando en 0)
```

```
Out[49]:
```

$$1$$

```
In [50]: M.inv() # Inversa.
```

```
Out[50]:
```

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

```
In [51]: N = sym.Matrix([(1,1),(2,2)])
```

```
In [52]: N.nullspace()
```

```
Out[52]:
```

$$\left[\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right]$$

```
In [53]: N.columnspace()
```

```
Out[53]:
```

$$\left[\begin{bmatrix} 1 \\ 2 \end{bmatrix}\right]$$

5.1 Ejercicio: construir la matriz de diferencias finitas.

Al intentar solucionar un problema de valor en la frontera con diferencias finitas, es común encontrarse con la siguiente matriz:

$$\begin{bmatrix} 2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & 2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 2 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 2 & 1 \\ 0 & 0 & \cdots & 0 & 1 & 2 \end{bmatrix}_{n \times n}$$

Construyámosla usando sympy. El ejercico consiste en escribir una función que pida el orden n y devuelva la matriz.

Primero, realicemos el proceso para un n fijo:

```
In [54]: n = 5
         M = sym.zeros(n)
         M
```

Out[54]:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
In [55]: for i in range(n):
             for j in range(n):
                 if i == j:
                     M[i,j] = 2
                 elif i == j-1:
                     M[i,j] = 1
                 elif i-1 == j:
                     M[i,j] = 1
```

```
In [56]: M
```

Out[56]:

$$\begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 & 2 \end{bmatrix}$$

Ya tenemos la idea lista, ahora:

```
In [57]: def matrizespecial(n):
             M = sym.zeros(n)
             for i in range(n):
                 for j in range(n):
                     if i == j:
                         M[i,j] = 2
                     elif i == j-1:
                         M[i,j] = 1
                     elif i-1 == j:
                         M[i,j] = 1
             return M
```

```
In [58]: matrizespecial(8)
```

Out[58]:

$$\begin{bmatrix} 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \end{bmatrix}$$