# Una introducción a Python




P. J. De Los Santos

# Contenido



# 1 Fundamentos del lenguaje

## 1.1 Python

Python es un lenguaje de programación, interpretado, de alto nivel y propósito general, además de ser un proyecto libre y de código abierto, con una comunidad enorme implicada en el desarrollo y mantenimiento de librerías que hacen posible el *multidominio* actual de Python.

Dada su concepción como lenguaje de propósito general, Python es utilizado en una diversidad de aplicaciones, desde desarrollo web, encriptación, análisis de datos, procesamiento de imágenes, aprendizaje automático, computación simbólica, etc.

Las características de este lenguaje le hacen propicio para el prototipado de aplicaciones, dado que es muy sencillo y rápido revisar y modificar el código desarrollado. Otra característica muy notable de Python es su sintaxis simple y fácil de aprender, lo cual ayuda al momento de introducirse en el desarrollo de algoritmos o el mundo propio de la programación de computadoras.

## 1.2 Instalando Python

En estos apuntes se utilizará la distribución Anaconda de Python, la cual contiene el intérprete y las librerías del *core*, pero además incluye la mayoría de librerías utilizadas para el desarrollo de aplicaciones de corte técnico-científico.

La descarga de Anaconda puede realizarla desde el sitio https://www.anaconda.com/download/ (https://www.anaconda.com/download/), selecciona el paquete de descarga conforme al sistema operativo (Windows, macOS o Linux) así como la arquitectura de su PC. La instalación suele ser muy sencilla, puede seguir las instrucciones dadas en el *How to install ANACONDA* de la misma página.

## 1.3 El notebook de Jupyter como una calculadora básica

Una vez instalado Anaconda puede testear la correcta instalación abriendo el Jupyter Notebook, la cual es una aplicación web en la cual puede escribir código, texto, ecuaciones, etc., y que básicamente es donde se desarrollaron estos apuntes. Puede buscar esta aplicación en la carpeta de instalación correspondiente.

A partir de este momento puede ingresar código Python en las celdas y teclear **Shift + Enter** para ejecutar la instrucción y la celda le devolverá lo que resulte de esto. Por ejemplo, si escribe un número cualesquiera y presiona la combinación indicada, la consola le devolverá justamente el mismo número:

```
In [1]: 1000
```

```
Out[1]: 1000
```

Puede ejecutar una simple suma aritmética:

```
In [2]: 100 + 200
```

```
Out[2]: 300
```

O una resta:

In [3]:
```
550 - 650
```

Out[3]:
```
-100
```

Naturalmente Python maneja sin complicaciones las cantidades negativas. Una multiplicación la realiza con el operador *:

In [4]:
```
50*25
```

Out[4]:
```
1250
```

Para las divisiones utiliza el operador /:

In [5]:
```
1/2
```

Out[5]:
```
0.5
```

Puede elevar a una potencia utilizando como operador el doble asterisco:

In [6]:
```
13**2
```

Out[6]:
```
169
```

Inclusive existe la posilidad de definir números complejos y realizar operaciones con ellos:

In [7]:
```
5 + 2j
```

Out[7]:
```
(5+2j)
```

In [8]:
```
(5 + 2j) - (10 + 7j)
```

Out[8]:
```
(-5-5j)
```

In [9]:
```
(5 + 2j)*(10 + 7j)
```

Out[9]:
```
(36+55j)
```

Puede ampliar la capacidad de las funcionalidades *built-in* de Python si importa alguna librería, como `math`, pero claro, eso será un tema a tratar con posterioridad.

## 1.4 Variables y tipos de datos

Al ser un lenguaje de alto nivel, Python dispone de los tipos de datos elementales en cualquier lenguaje de programación, pero además incluye estructuras de datos muy *avanzadas* y con altas prestaciones que facilitan en muchos aspectos la tarea del programador.

Python es un lenguaje de tipado dinámico en el que no hace falta declarar el tipo de dato que asignará a una variable, de igual manera una variable puede cambiar de tipo conforme la ejecución del programa, por ello se debe tener cuidado con la sintaxis para definir cada tipo de dato.

### 1.4.1 Variables

Las variables son referencias a los objetos de Python, son creadas por asignación mediante el signo `=`, por ejemplo:

In [10]:
```
a = 2
b = 10
a + b
```

Out[10]:
```
12
```

El nombre de una variable puede constar de una combinación de caracteres alfanúmericos y el guión bajo, siempre y cuando el primer caracter no sea un dígito. Además, en Python los nombres de variables son *case sensitive*, es decir, se distingue entre mayúsculas y minúsculas.

In [11]:
```
D = 177.8
d = 95
```

In [12]:
```
print(D)
```

```
177.8
```

In [13]:
```
print(d)
```

```
95
```

Existen algunas palabras reservadas del lenguaje que no puede utilizar como nombre de variable, puede verificar cuáles son estas palabras tecleando lo siguiente:

In [14]:
```
import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally',
'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while',
'with', 'yield']
```

### 1.4.2 Enteros ( `int` )

Los enteros son un tipo de dato básico en cualquier lenguaje de programación. En Python para definir un valor entero se debe colocar el número sin ningun punto decimal, por ejemplo:

In [30]:
```
a = 1
type(a)
```

Out[30]:
```
int
```

De manera explícita se puede definir un valor entero utilizando la función `int` :

In [71]:
```
m = 5.0
n = int(5.0)
type(m), type(n)
```

Out[71]:
```
(float, int)
```

Observe que cuando colocamos un punto decimal, automáticamente la cantidad deja de ser un entero y pasa a ser un flotante.

### 1.4.3 De coma flotante ( `float` )

Los valores de coma flotante son cantidades numéricas que incluyen a todos los reales. Para que Python reconozca un valor numérico como de tipo `float` se debe adicionar el punto decimal o bien utilizar la función `float` para hacer la indicación de manera explícita, por ejemplo:

In [72]:
```
w = 5.3
x = 10.0
y = 9.
z = float(8)
type(w), type(x), type(y), type(z)
```

Out[72]:
```
(float, float, float, float)
```

### 1.4.4 Booleanos

Las variables booleanas sólo pueden adoptar dos valores: verdadero ( `True` ) o falso ( `False` ). Un valor booleano se puede definir directamente a partir de las constantes `True` y `False` :

In [73]:
```
a = True
b = False
type(a), type(b)
```

Out[73]:
```
(bool, bool)
```

O bien a partir de otros objetos Python al aplicar la función `bool` :

In [74]:
```
bool("hola")
```

Out[74]:
```
True
```

In [75]:
```
bool([])
```

Out[75]:
```
False
```

In [76]:
```
bool(0)
```

Out[76]:
```
False
```

In [77]:
```
bool(10)
```

Out[77]:
```
True
```

En general, la función `bool` devolverá un `False` cuando se tienen objetos nulos o vacíos, en cualquier otro caso devolverá el valor `True` .

### 1.4.5 Cadenas de caracteres ( `str` )

Las cadenas de caracteres (denominadas habitualmente y de manera indistinta como *strings* es un tipo de dato que contiene una secuencia de símbolos, mismos que pueden ser alfanúmericos hasta cualquier otro símbolo propio de un sistema de escritura. En Python los strings se definen entre comillas dobles o simples:

In [78]:
```
"esta es una cadena de caracteres"
```

Out[78]:
```
'esta es una cadena de caracteres'
```

In [79]:
```
'esta también'
```

Out[79]:
```
'esta también'
```

Puede concatenar dos strings utilizando el operador `+` :

In [80]:
```
"Hola" + "mundo"
```

Out[80]:
```
'Holamundo'
```

Notará que Python por sí mismo no sabe que estamos uniendo dos palabras y que entre ellas debería haber un espacio para su correcta lectura, evidentemente este tipo de cuestiones son las que el programador debe tomar en cuenta al escribir un código.

Una cadena de caracteres es lo que en Python se conoce como *iterable*, es decir, una secuencia de elementos agrupados a los cuales se puede acceder de manera individual mediante indexación. Por ejemplo, sea `nombre` una cadena de caracteres dada por:

In [81]:
```
nombre="Catalina"
```

Puede acceder a cada una de las letras que componen dicha cadena mediante la notación `iter[pos]` , donde `iter` es el nombre del iterable y `pos` la posición en que se encuentra el caracter al cual se desea acceder, siendo 0 para la primera letra, 1 para la segunda y así de manera consecutiva. Por ejemplo:

In [82]:
```
nombre[0]
```

Out[82]:
```
'C'
```

In [83]:
```
nombre[4]
```

Out[83]:
```
'l'
```

In [84]:
```
nombre[2]
```

Out[84]:
```
't'
```

Al último elemento, sin importar la longitud de la cadena, se accede con el índice -1:

In [85]:
```
nombre[-1]
```

Out[85]:
```
'a'
```

### 1.4.6 Listas

Las listas son estructuras de datos que pueden almacenar cualquier otro tipo de dato, inclusive una lista puede contener otra lista, además, la cantidad de elementos de una lista se puede modificar removiendo o añadiendo elementos. Para definir una lista se utilizan los corchetes, dentro de estos se colocan todos los elementos separados por comas:

In [86]:
```
calificaciones = [10,9,8,7.5,9]
nombres = ["Ana","Juan","Sofía","Pablo","Tania"]
mezcla = [True, 10.5, "abc", [0,1,1]]
```

Las listas son *iterables* y por tanto se puede acceder a sus elementos mediante indexación:

In [87]:
```
nombres[2]
```

Out[87]:
```
'Sofía'
```

In [88]:
```
nombres[-1]
```

Out[88]:
```
'Tania'
```

Se tiene la posibilidad de agregar elementos a una lista mediante el método `append` :

In [89]:
```
nombres.append("Antonio")
nombres.append("Ximena")
print(nombres)
```

```
['Ana', 'Juan', 'Sofía', 'Pablo', 'Tania', 'Antonio', 'Ximena']
```

El método `remove` elimina un elemento de una lista:

In [90]:
```
nombres.remove("Ana")
print(nombres)
```

```
['Juan', 'Sofía', 'Pablo', 'Tania', 'Antonio', 'Ximena']
```

Sí el valor pasado al método `remove` no existe, Python devolverá un `ValueError` :

In [91]:
```
nombres.remove("Jorge")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-91-d983d2559e2f> in <module>()
----> 1 nombres.remove("Jorge")

ValueError: list.remove(x): x not in list
```

### 1.4.7 Tuplas

Las tuplas son secuencias de elementos similares a las listas, la diferencia principal es que las tuplas no pueden ser modificadas directamente, es decir, una tupla no dispone de los métodos como `append` o `insert` que modifican los elementos de una lista.

Para definir una tupla, los elementos se separan con comas y se encierran entre paréntesis.

In [92]:
```
colores=("Azul","Verde","Rojo","Amarillo","Blanco","Negro","Gris")
```

Las tuplas al ser *iterables* pueden accederse mediante la notación de corchetes e índice.

In [93]:
```
colores[0]
```

Out[93]:
```
'Azul'
```

In [94]:
```
colores[-1]
```

Out[94]:
```
'Gris'
```

In [95]:
```
colores[3]
```

Out[95]:
```
'Amarillo'
```

Si intentamos modificar alguno de los elementos de la tupla Python nos devolverá un `TypeError` :

In [96]:
```
colores[0] = "Café"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-96-3502c7127536> in <module>()
----> 1 colores[0] = "Café"

TypeError: 'tuple' object does not support item assignment
```

### 1.4.8 Diccionarios

Los diccionarios son estructuras que contienen una colección de elementos de la forma `clave: valor` separados por comas y encerrados entre llaves. Las claves deben ser objetos inmutables y los valores pueden ser de cualquier tipo. Necesariamente las claves deben ser únicas en cada diccionario, no así los valores.

Vamos a definir un diccionario llamado `edades` en el cual cada clave será un nombre y el valor una edad:

In [97]:
```
edades = {"Ana": 25, "David": 18, "Lucas": 35, "Ximena": 30, "Ale": 20}
```

Puede acceder a cada valor de un diccionario mediante su clave, por ejemplo, si quisieramos obtener la edad de la clave `Lucas` se tendría que escribir:

In [104]:
```
edades["Lucas"]
```

Out[104]:
```
35
```

## 1.5 Operadores relacionales y lógicos

Los operadores relacionales (o de comparación) nos permite efectuar comparaciones entre objetos de Python. El resultado de una comparación es un valor booleano `True` o `False` , dependiendo la naturaleza de la comparación.

A continuación se ejemplifican los operadores relacionales que podemos utilizar en Python:

In [105]:
```
# "igual a"
1 == 1
```

Out[105]: True

In [106]:
```
# "diferente a"
"a" != "a"
```

Out[106]: False

In [107]:
```
# mayor que
10 > 5
```

Out[107]: True

In [108]:
```
# menor que
5 < 1
```

Out[108]: False

In [109]:
```
# mayor o igual que
30 >= 30
```

Out[109]: True

In [110]:
```
# menor o igual que
20 <= 10
```

Out[110]: False

Hay que tener cuidado y verificar que al hacer comparaciones los objetos implicados sean compatibles. Cuando los objetos no son comparables Python devolverá un `TypeError`, por ejemplo:

In [113]:
```
"a" > 10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-113-7a3a9a917f79> in <module>()
----> 1 "a" > 10

TypeError: '>' not supported between instances of 'str' and 'int'
```

In [114]:
```
[0,4] < (1,2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-114-4dd688af572f> in <module>()
----> 1 [0,4] < (1,2)

TypeError: '<' not supported between instances of 'list' and 'tuple'
```

También podemos *encadenar* comparaciones con cadenas del tipo `a < b < c`, donde ese `<` puede ser cualquier operador relacional, por ejemplo:

In [115]:
```
1 < 2 < 3
```

Out[115]: True

In [116]:
```
10 > 10 > 3
```

Out[116]: False

In [117]:
```
3 == 3 >= 2
```

Out[117]: True

Los **operadores lógicos** nos sirven para realizar operaciones de lógica booleana entre valores de tipo `bool`. En Python podemos utilizar los operadores lógicos `and`, `or` y `not`, observe los siguientes ejemplos:

In [118]:
```
True and True
```

Out[118]: True

In [119]:
```
True and False
```

Out[119]: False

In [120]:
```
True or False
```

Out[120]: True

In [121]:
```
not True
```

Out[121]: False

In [122]:
```
(1 == 1) and (2 > 1)
```

Out[122]: True

In [123]:
```
(0 == 0) or (10 >=20) and (1 > 0)
```

Out[123]: True

In [124]:
```
not( (2 > 3) and (5==5) )
```

Out[124]: True

## 1.6 Las funciones `print` e `input`

In [130]:
```
print("Hola mundo")
```

```
Hola mundo
```

In [131]:
```
print("hola", "mundo", 1, 2, 3)
```

```
hola mundo 1 2 3
```

In [132]:
```
print(20,30,"python", sep=",")
```

```
20,30,python
```

In [133]:
```
print(20,30,"python", end="\n\n")
print("hola")
```

```
20 30 python

hola
```

In [134]:
```
print(30*"-")
```

```
------------------------------
```

In [135]:
```
a = 10
b = 30
print("a + b = {a} + {b} = {c}".format(a=a,b=b,c=a+b))
```

```
a + b = 10 + 30 = 40
```

In [136]:
```
a = 32479
b = 54988
print("a + b = {a:0.3e} + {b:0.2f} = {c}".format(a=a,b=b,c=a+b))
```

```
a + b = 3.248e+04 + 54988.00 = 87467
```

In [139]:
```
nombre = input("Ingrese su nombre: ")
print("Hola", nombre, "bienvenido")
```

```
Ingrese su nombre: Jorge
Hola Jorge bienvenido
```

In [141]:
```
n = float( input("Ingrese un número: ") )
print("2 *",n,"=", 2*n)
```

```
Ingrese un número: 10
2 * 10.0 = 20.0
```

## 1.7 Control de flujo

### 1.7.1 Condicional `if-elif-else`

El condicional `if-elif-else` es una estructura de control que sirve para tomar decisiones en el flujo del programa. La sintaxis para `if-elif-else` es:

```
if cond1:
    # hacer algo
elif cond2:
    # hacer otra cosa
    .
    .
    .
elif condn:
    # hacer algo más
else:
    # hacer algo por default
```

Donde `cond1, cond2, ... condn` son valores lógicos que resultan de una comparación. Esta estructura se evalúa secuencialmente hasta encontrar una condición que se cumpla, si ninguna lo hace, entonces se ejecuta la instrucción colocada en el caso por default `else` .

In [144]:
```
n = 1111

if (-1)**(n) > 0:
    print("El número es par")
else:
    print("El número es impar")
```

```
El número es impar
```

In [145]:
```
nota = 7

if 9.5 <= nota <= 10:
    print("Excelente")
elif 8.5 <= nota < 9.5:
    print("Muy bien")
elif 7.5 <= nota < 8.5:
    print("Bien")
elif 7.0 <= nota < 7.5:
    print("Regular")
else:
    print("No aprobado")
```

```
Regular
```

### 1.7.2 Bucle `for`

El **bucle for** es una estructura de control de naturaleza repetitiva, en la cual se conocen *a priori* el número de iteraciones a realizar. En lenguajes como C++ o Java, el ciclo `for` necesita de una variable de ciclo de tipo entero que irá incrementándose en cada iteración. En Python, la cuestión es un poco diferente, el ciclo `for` *recorre* un *iterable* y en la k-ésima iteración la variable de ciclo *adopta* el valor del elemento en la k-ésima posición del iterable.

De manera general, la sintaxis de `for` es:

```
for var in iterable:
    # Hacer algo ...
```

Donde `var` es la **variable de ciclo** e `iterable` la secuencia de valores que deberá iterarse. Es necesario remarcar la importancia de los dos puntos al final de esta primera línea y en indentar el bloque de código subsecuente que definirá el cuerpo del ciclo for.

Como primer ejemplo vamos a recorrer una lista de números y mostrarlos por consola:

In [147]:
```
numeros = [18,50,90,-20,100,80,37]
for n in numeros:
    print(n)
```

```
18
50
90
-20
100
80
37
```

Observe que en cada iteración la variable de ciclo `n` adopta el valor de cada uno de los elementos de la lista `numeros` .

Como ya se mencionó, en Python la variable de ciclo no necesariamente adopta valores numéricos enteros secuenciales, si no valores dentro de una secuencia. Esta secuencia podría ser también una cadena de caracteres, por ejemplo:

In [148]:
```
palabra = "Python"
for letra in palabra:
    print(letra)
```

```
P
y
t
h
o
n
```

Dentro de un ciclo for podemos colocar cualesquiera otra instrucción de control de flujo. Un caso muy común es el de incluir otro ciclo for, algo que habitualmente se denota como **ciclos anidados**. Por ejemplo, supongamos que se requieren mostrar por consola todos los elementos de algunas listas contenidas dentro de otra lista principal, en ese caso se hace necesario primero iterar sobre la lista principal y enseguida hacerlo sobre las listas contenidas, por ejemplo:

In [150]:
```
matriz = [[-5,2,0], [9,5,6], [1,7,15]]
for fila in matriz:
    for elemento in fila:
        print(elemento)
```

```
-5
2
0
9
5
6
1
7
15
```

### 1.7.3 Bucle `while`

## 1.8 Funciones

Las funciones son *porciones de código* que nos sirven para modularizar nuestros programas y evitar en muchos casos la repetitividad de código. De manera general una función recibe algunos valores de entrada, los *procesa* y devuelve algunos valores de salida (o bien modifica algunas variables).

### 1.8.1 Funciones nativas de Python ( `built-in` )

Python dispone de algunas funciones nativas que se *cargan* automáticamente cuando se inicia el intérprete. Por ejemplo la función `max` devuelve el mayor valor numérico de una lista de números:

In [151]:
```
max([10,35,5,110,48,30,112,98,87])
```

Out[151]: 112

También existe una función `min` , análoga a `max` :

In [152]:
```
min([10,35,5,110,48,30,112,98,87])
```

Out[152]: 5

Otro ejemplo de función nativa es `bin` , la cual dado un número en base 10 devuelve una cadena con la representación en base 2.

In [153]:
```
bin(10)
```

Out[153]: '0b1010'

Naturalmente, el valor devuelto por una función se puede asignar a una variable y posteriormente ser utilizado:

In [154]:
```
a = max([10,5,8])
b = min([10,5,8])
h = (a - b)/10
print(h)
```

```
0.5
```

Hay funciones que no devuelven como tal un valor, si no que pueden modificar directamente alguna variable global o simplemente mostrar algo en la salida estándar como el caso de `print` .

Tendremos también funciones que aceptan más de un argumento, por ejemplo a la función `round` podemos pasarle dos argumentos: un número real y la cantidad de lugares decimales a considerar para el redondeo.

In [155]:
```
round(3.141592653589793, 6)
```

Out[155]: 3.141593

In [156]:
```
round(3.141592653589793, 2)
```

Out[156]:
```
3.14
```

### 1.8.2 Funciones definidas por el usuario

Además de las funciones nativas de Python, es posible definir nuestras propias funciones. En Python, de manera general, una función se define siguiendo la estructura mostrada a continuación:

```
def nombre_fun(arg1, arg2, ..., argN):
    # Cuerpo de la función
    # .
    # .
    # .
    return val1, val2, ..., valN
```

Donde `def` es una palabra que debe anteceder siempre a la definición de una función, `nombre_fun` es el nombre que se asignará a la función, entre paréntesis y separados por comas se colocan los nombres de los argumentos de entrada, los dos puntos se colocan después de cerrar el paréntesis e indican que ahí termina el *encabezado* de la función y comenzará el *cuerpo* de la misma, aquí se colocarán todas las instrucciones que deberán realizarse; la palabra reservada `return` sirve para indicar los valores a devolver, mismos que se colocarán separados por comas.

Vamos a definir una función llamada `saluda`, la cual recibe un nombre (string) y devuelve un saludo (string) formado mediante concatenación:

In [157]:
```
def saluda(nombre):
    s = "Hola " + nombre + ", bienvenido."
    return s
```

In [158]:
```
print(saluda("Jorge"))
```

```
Hola Jorge, bienvenido.
```

Lo único que hace la función anterior es tomar un *string* como argumento y unirlo a algunas cadenas ya establecidas dentro de la función.

Veamos ahora cómo definir una función que recibe como argumento un entero y devuelve un valor lógico que indica si este es par.

In [159]:
```
def espar(n):
    if n%2 == 0:
        s = True
    else:
        s = False
    return s
```

In [160]:
```
print(espar(2))
print(espar(5))
print(espar(10))
```

```
True
False
True
```

Naturalmente, las funciones pueden recibir más de un argumento. Por ejemplo:

In [161]:
```
def mayor(a,b):
    m = a
    if a < b:
        m = b
    return m
```

In [162]:
```
print( mayor(50,30) )
print( mayor(1100,3050) )
```

```
50
3050
```

La función `mayor` recibe dos valores numéricos y determina cuál es el mayor de ambos mediante una comparación con la sentencia `if`.

¿Pueden las funciones en Python devolver más de un valor? ¡Claro! Hace falta nada más separar con comas los valores a devolver.

In [163]:
```
def calcula_rectangulo(b,h):
    A = b*h
    P = 2*b + 2*h
    return A, P
```

In [164]:
```
print( calcula_rectangulo(10,5) )
print( calcula_rectangulo(50,15) )
```

```
(50, 30)
(750, 130)
```

También es posible guardar/asignar los valores devueltos por la función en variables:

In [165]:
```
area1, perimetro1 = calcula_rectangulo(100, 20)
print("Área: {0}\nPerímetro: {1}".format(area1, perimetro1))
```

```
Área: 2000
Perímetro: 240
```

### 1.8.3 Funciones con una cantidad de parámetros indeterminada

En ocasiones el número de parámetros que deberá recibir una función no puede ser algo fijo. Las definiciones de función en Python tienen la flexibilidad de poder recibir una cantidad variable de argumentos de entrada.

Para ejemplificar esto, vamos a crear una función llamada `promedio` que calcule el promedio de una cierta cantidad de números pasados como argumentos:

In [166]:
```
def promedio(*numeros):
    suma = 0
    k = 0
    for n in numeros:
        suma += n
        k += 1
    return suma/k
```

In [167]:
```
print(promedio(10,5))
print(promedio(10,50,40,80,20,100))
print(promedio(5,15,10,5))
```

```
7.5
50.0
8.75
```

Observe que lo único que hacemos es que al nombre del parámetro le anteponemos un asterisco, esto le indica a Python que la cantidad de argumentos de entrada es indeterminada, en principio. Claro está, que el manejo posterior de la información es algo que el programador debe tener en cuenta. Dentro del cuerpo de la función se debe considerar que el parámetro `numeros` será una tupla cuya cantidad de elementos dependerá de la cantidad de argumentos ingresados.

### 1.8.4 Funciones y los argumentos con nombre

Una función en Python se puede *mandar a llamar* pasándo los argumentos de manera posicional, es decir, en el orden que fueron definidos en la función, o bien, haciendo uso del nombre del parámetro correspondiente al argumento que se introduce, por ejemplo:

In [168]:
```
def cuenta_cuantas(frase, letra):
    k = 0
    for car in frase:
        if car is letra:
            k += 1
    return k
```

In [169]:
```
print( cuenta_cuantas("hola mundo", "o") )
print( cuenta_cuantas(frase="hola mundo", letra="o") )
print( cuenta_cuantas(letra="o", frase="hola mundo") )
```

```
2
2
2
```

La función `cuenta_cuantas` devuelve el número de presencias de una determinada letra en una frase. Observe las tres formas en que la *ejecutamos*, todas son equivalentes. En la primera se pasan los argumentos de forma posicional, en la segunda y tercera se utilizan los argumentos con nombres, note que en este caso el orden en que los argumentos son pasados, es indistinto.

En la definición de funciones es posible también especificar que se pasarán ciertos argumentos con nombre sin necesidad de escribirlos de manera explícita. Observe la siguiente función:

In [170]:
```
def muestra_puntos(**personas):
    for persona in personas.items():
        print(persona[0] + " tiene " + str(persona[1]) + " puntos")
```

In [171]:
```
muestra_puntos(Jorge=8, Paty=10)
print(30*"=")
muestra_puntos(Ana=6, Carlos=9, Victor=4, Daniela=8)
```

```
Jorge tiene 8 puntos
Paty tiene 10 puntos
==============================
Ana tiene 6 puntos
Carlos tiene 9 puntos
Victor tiene 4 puntos
Daniela tiene 8 puntos
```

Vea que la definición de la función `muestra_puntos` incluye un parámetro llamado `**personas` , esos dos asteriscos antes del nombre del parámetro, indican

que no se tiene predeterminado el número de argumentos que se pasarán, pero además, indica que cada argumento a introducir deberá ser un argumento con nombre. Dentro del cuerpo de la función el parámetro `**personas` es un diccionario cuyas claves son los nombres de los argumentos y los valores corresponden a cada valor asignado al argumento.

---

## 1.9 Ejercicios

1. En las siguientes opciones se muestran operaciones aritméticas entre diversos objetos de Python. Verifique si es posible realizarlas e indique el resultado, de no ser así describa el por qué.
   A. `1 + 2`
   B. `1.3 + 2.5`
   C. `"1" + 2`
   D. `"1" + "2"`
   E. `[1,2,3] + [10,20]`
   F. `{"a":10, "b":5} + {"h":2, "i":4}`
2. Observe el siguiente código e identifique y explique el error:

```
for a,b in [1,2,5,3,8,7]:
    print(a)
```

3. El siguiente código debería imprimir la longitud de cada palabra contenida en la lista `palabras`. Identifique el error.

```
palabras = ["Carro", "Sol", "Mesa", "Dinosaurio", "Girasol", "Silla"]
for palabra in palabras:
    print(len(palabras))
```

4. Implemente un programa que determine si un número dado es par o impar.
5. Escriba un programa que cuente el número de vocales en una frase. Tome en cuenta que las vocales podrían ser tanto mayúsculas como minúsculas.
6. Escriba un programa que aproxime, mediante la suma de Riemann, el área bajo la curva de la función $f(x) = x^2 + 3x$ en el intervalo $0 \leq x \leq 10$.

# 1 Matplotlib: visualización gráfica

## Contenido



## 1.1 De Matplotlib, Jupyter y las gráficas mostradas

En todo este capítulo se asumirá que las siguientes líneas de código han sido ejecutadas, previamente, para cada porción de código:

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

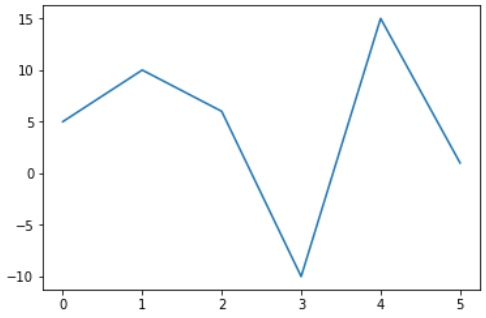
Sí usted ejecuta Python/Matplotlib dentro de un entorno diferente a Jupyter, deberá evitar colocar la instrucción `%matplotlib inline` y en su lugar colocar la instrucción `plt.show()` al final de cada código, para que se le muestren las gráficas correspondientes.

## 1.2 Una primera aproximación

Una de las características de Matplotlib es la facilidad con la que se puede comenzar a trazar gráficas, vea el siguiente código:

In [2]:
```
plt.plot([5,10,6,-10,15,1])
```

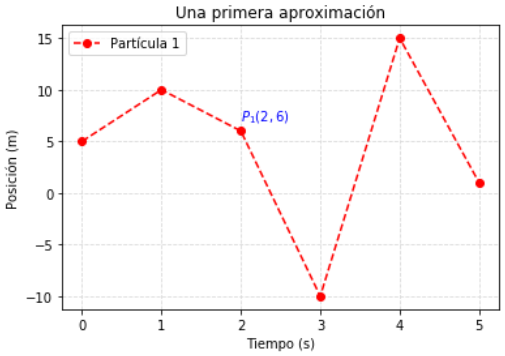
Out[2]:
```
[<matplotlib.lines.Line2D at 0x1e01a67ccc0>]
```




El código anterior produce la gráfica mostrada en la figura. Como puede observar son solamente tres líneas de instrucciones, la primera sirve para importar el módulo `pyplot` de Matplotlib, el cual contiene muchas de las funciones útiles para el trazo de gráficas; la segunda línea ejecuta la función `plot` pasando como argumento una lista de valores numéricos, y finalmente la tercera línea se encarga de mostrar el elemento gráfico resultante en una ventana.

Un resultado un poco más *trabajado* se obtiene con el siguiente código:

In [5]:
```
plt.plot([0,1,2,3,4,5], [5,10,6,-10,15,1], 'r--o', label="Partícula 1")
plt.xlabel("Tiempo (s)")
plt.ylabel("Posición (m)")
plt.title("Una primera aproximación")
plt.text(2,7,"$ P_1 (2,6) $", color="b")
plt.legend()
plt.grid(ls="--", color="#dadada")
```



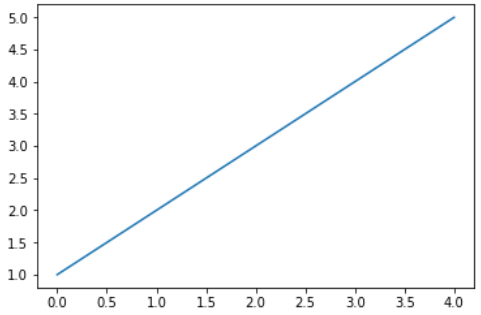

## 1.3 La función `plot`

La función `plot` está contenida en el módulo `pyplot` y básicamente con esta se produce cualquier gráfica bidimensional en coordenadas rectangulares. Esta función soporta varias maneras de ejecutarla dependiendo la cantidad de argumentos que se pasen.

La forma más básica de la función `plot` es pasarle un sólo argumento, por ejemplo:

In [6]:
```
plt.plot([1,2,3,4,5])
```

Out[6]:
```
[<matplotlib.lines.Line2D at 0x1e01a8dbcf8>]
```

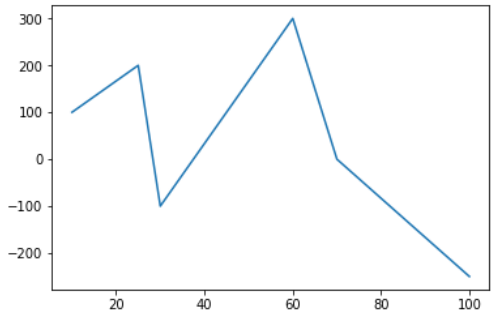



Al pasarle un sólo argumento, este se toma como los valores de la coordenada vertical, y se asume que la horizontal varía de 0 a `N-1`, donde `N` es el número de elementos contenidos en la lista de valores que se introducen.

La sintaxis más habitual es introducir dos argumentos, donde el primero contiene una lista `X` que define los valores de la coordenada horizontal, y el segundo una lista `Y` correspondiente a los valores de la coordenada vertical, por ejemplo:

In [7]:
```
plt.plot([10,25,30,60,70,100], [100,200,-100,300,0,-250])
```

Out[7]:
```
[<matplotlib.lines.Line2D at 0x1e01a935390>]
```




### 1.3.1 Graficando funciones matemáticas

En matemáticas una función es una relación que asigna elementos de un conjunto de manera unívoca a otro conjunto. Usualmente una función matemática se puede representar mediante una gráfica en coordenadas cartesianas, colocando uno de los conjuntos en el eje horizontal y el otro en el vertical.

Utilizando Python, y de manera específica la librería NumPy, se pueden evaluar las funciones matemáticas en un intervalo determinado y en una cantidad finita de puntos. Por ejemplo, suponga que se requieren calcular todos los pares coordenados correspondientes a la función $y = \cos x$ en el intervalo $0 \leq x \leq 5$, en Python se tendría que definir como:

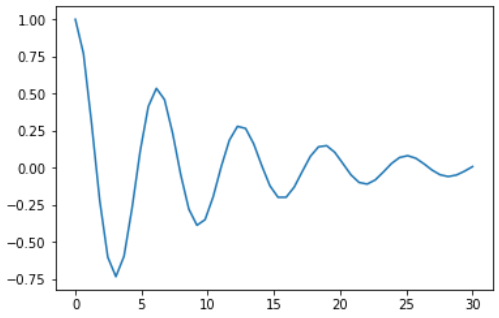
In [8]:
```
x = np.linspace(0,5)
y = np.cos(x)
```

Las variable `x` es un arreglo de NumPy que contiene 50 valores linealmente equiespaciados entre 0 y 5, la variable `y` es también un arreglo de NumPy que resulta de aplicar la función coseno a cada valor de `x`.

De manera similar a lo anterior se procederá a definir y graficar la función $y = e^{-0.1x}\cos x$ en el intervalo $0 \leq x \leq 30$:

In [9]:
```
x = np.linspace(0, 30)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x,y)
```

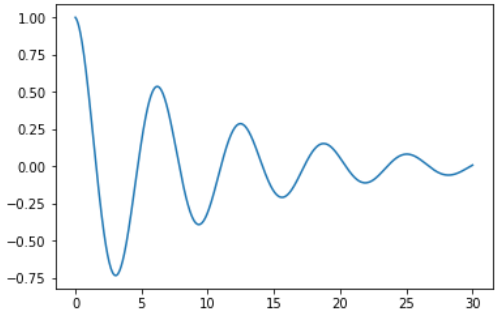
Out[9]:
```
[<matplotlib.lines.Line2D at 0x1e01a987518>]
```




La cantidad de puntos a evaluar es una cuestión muy importante, ya que de esto depende la correcta visualización del comportamiento de una función. Naturalmente, entre más puntos evaluados mejor será la apreciación que se tenga de la curva en cuestión, pero implica un mayor gasto de memoría para guardar y evaluar todos los datos. Enseguida se muestra la misma función graficada en el mismo intervalo pero con 1000 y 5 puntos evaluados de manera respectiva, notará la diferencia entre los casos, es evidente que en el caso de los 5 puntos *se pierde* muchísima información.

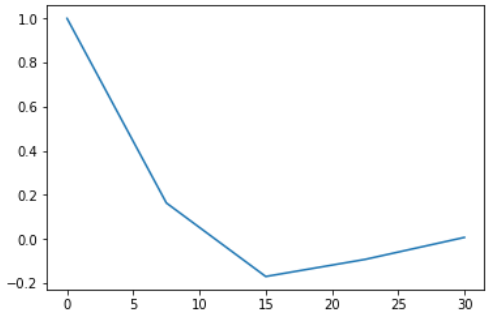
In [10]:
```
# Con 1000 puntos evaluados
x = np.linspace(0, 30, 1000)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x,y)
```

Out[10]:
```
[<matplotlib.lines.Line2D at 0x1e01a9e7400>]
```




In [11]:
```
# Con 5 puntos evaluados
x = np.linspace(0, 30, 5)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x,y)
```

Out[11]:
```
[<matplotlib.lines.Line2D at 0x1e01aa476d8>]
```



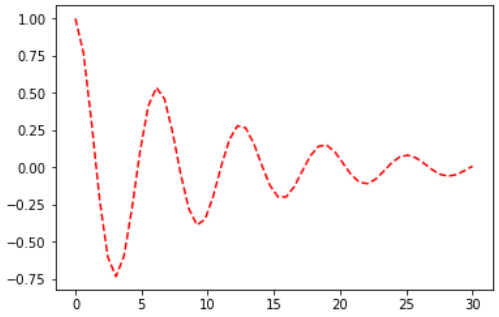
### 1.3.2 Modificando el color, estilos y grosor de línea

La función `plot` acepta argumentos adicionales que sirven para modificar y controlar características de la línea que se grafica.

Se puede pasar un tercer argumento que contenga una combinación de color y estilo de línea. Por ejemplo:

In [12]:
```
x = np.linspace(0, 30)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x, y, "r--")
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x1e01aaa72b0>]
```

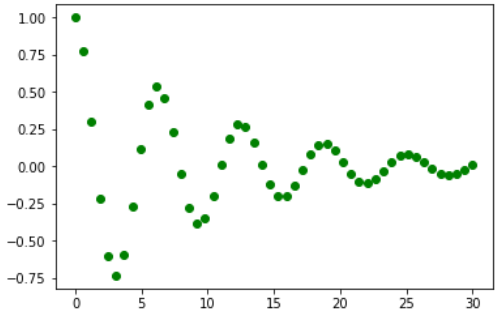



El código anterior genera una gráfica con una línea en color rojo ( `r` ) y un estilo de línea discontinua ( `--` ).

Si en lugar del string `--` se coloca `go` , se obtiene una gráfica como la mostrada enseguida, podrá inferir que `g` refiere al color verde (green) y `o` justamente al uso de esta como símbolo para representar cada punto.

In [13]:
```
x = np.linspace(0, 30)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x, y, "go")
```

Out[13]:
```
[<matplotlib.lines.Line2D at 0x1e01a871908>]
```

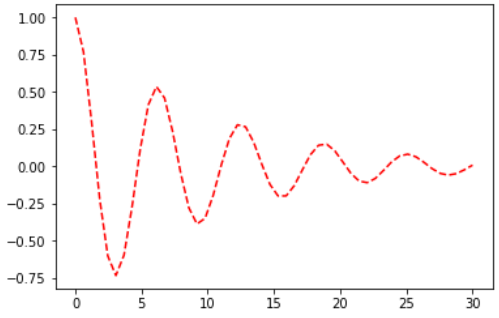



En https://matplotlib.org/api/markers_api.html (https://matplotlib.org/api/markers_api.html) se muestra una tabla con los símbolos (markers) disponibles para utilizar en la función `plot` . En https://matplotlib.org/api/colors_api.html (https://matplotlib.org/api/colors_api.html) puede consultar información respecto a los colores que puede abreviar mediante un sólo caracter.

Además de la forma anterior, también es posible especificar el color y estilo de línea utilizando *keyword arguments*, por ejemplo:

In [14]:
```
x = np.linspace(0, 30)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x, y, linestyle="--", color="r")
```

Out[14]:
```
[<matplotlib.lines.Line2D at 0x1e01a7479e8>]
```

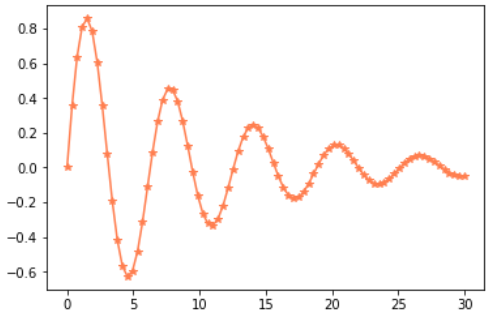



En ambos casos se especifica un cierto estilo de línea y color, con la diferencia notoria de la sintaxis.

Utilizar *keyword arguments* es una manera más general, puesto que la definición con strings no funciona para los casos en que se requieren colores que no se pueden especificar con un sólo caracter, por ejemplo, Matplotlib dispone de un color llamado `coral` y este no puede ser invocado mediante un sólo caracter, hace falta escribir todo el nombre.

In [15]:
```
x = np.linspace(0, 30, 80)
y = np.exp(-0.1*x)*np.sin(x)
plt.plot(x, y, linestyle="-", color="coral", marker="*")
```

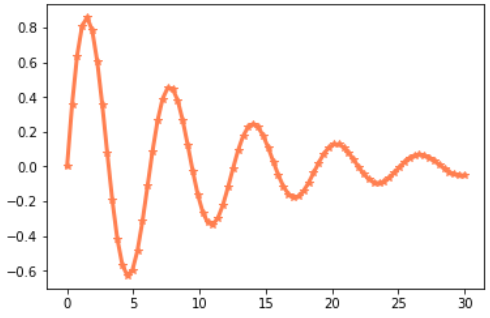
Out[15]:
```
[<matplotlib.lines.Line2D at 0x1e01a85ec18>]
```



El ancho de línea se puede controlar mediante el *keyword argument* `linewidth` , por ejemplo;

In [16]:
```
plt.plot(x, y, linestyle="-", color="coral", marker="*", linewidth=3)
```

Out[16]:
```
[<matplotlib.lines.Line2D at 0x1e01ab34908>]
```



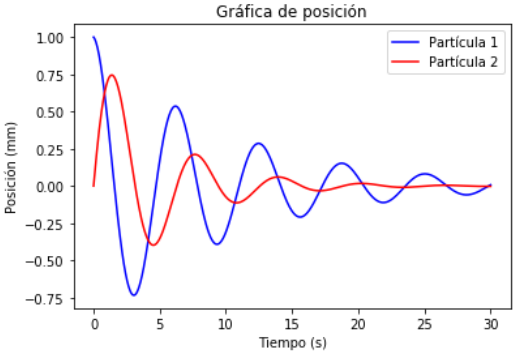
### 1.3.3 Título de gráfica, etiquetas de ejes y nombres de curvas

Por su naturaleza las gráficas nos sirven para presentar y/o visualizar información de ciertos datos, para lo cual se hace necesario especificar información descriptiva de lo que se muestra. Es muy común que se agreguen etiquetas a los ejes horizontal y vertical, así como el nombre de gráfica. Además, si se está graficando más de una curva, se hace necesario especificar a qué refiere cada una de ellas.

Por ejemplo, observe el siguiente código y la gráfica producida:

In [17]:
```
x = np.linspace(0, 30, 500)
y1 = np.exp(-0.1*x)*np.cos(x)
y2 = np.exp(-0.2*x)*np.sin(x)
plt.plot(x, y1, "b-", label="Partícula 1")
plt.plot(x, y2, "r-", label="Partícula 2")
plt.xlabel("Tiempo (s)")
plt.ylabel("Posición (mm)")
plt.title("Gráfica de posición")
plt.legend()
```

Out[17]:
```
<matplotlib.legend.Legend at 0x1e01ab93eb8>
```



La instrucción `xlabel` coloca una etiqueta al eje horizontal, de manera similar `ylabel` lo hace para el eje vertical. Con `title` adicionamos un título a la gráfica. La instrucción `legend` sirve para colocar el recuadro con *nombre* asignado a cada curva mediante el *keyword argument* `label` .

### 1.3.4 Anotaciones

Con anotaciones nos referimos a cualesquiera texto que se coloque dentro del *Axes* de Matplotlib. Usualmente utilizadas para indicar ciertas características partículares en una gráfica, o bien alguna nota informativa al respecto. La función base para realizar este tipo de tareas es `text` . La sintaxis más simple de `text` es:

```
plt.text(px, py, texto)
```

Donde `px` y `py` denotan las coordenadas en donde se colocará la anotación indicado en `texto` . Veamos un ejemplo:

In [18]:
```
x = np.linspace(0, 30, 100)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x, y, "m")
plt.xlabel("Tiempo (s)")
plt.ylabel("Posición (mm)")
plt.title("Gráfica de posición")
plt.text(10, 0.5, "Algo informativo")
```

Out[18]:
```
Text(10,0.5,'Algo informativo')
```


Note que únicamente colocamos el texto *Algo informativo* dentro del gráfico, de manera más específica en las coordenadas (10,0.5).

Al texto colocado podemos darle formato y ajustarlo a nuestros requerimientos, para ello a la función `text` se le pueden incluir los *keyword arguments* descritos en https://matplotlib.org/users/text_props.html (https://matplotlib.org/users/text_props.html). Por ejemplo:

In [19]:
```
x = np.linspace(0, 30, 200)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x, y, "m")
plt.xlabel("Tiempo (s)")
plt.ylabel("Posición (mm)")
plt.title("Gráfica de posición")
plt.text(10, 0.5, "Algo informativo", fontsize=16, color="r",
         name="Times New Roman")
```

Out[19]:
```
Text(10,0.5,'Algo informativo')
```


Observe que lo único que se cambió fueron algunas propiedades del texto, tales como el tamaño de la fuente con `fontsize` , el color de fuente con `color` y el tipo de fuente con `name` , con este último se debe tener cuidado, dado que el nombre de la fuente indicada debe estar instalada en la PC que se ejecuta.

## 1.4 Gráficas en coordenadas polares

Las coordenadas polares o sistema de coordenadas polares son un sistema de coordenadas bidimensional en el que cada punto del plano se determina por una distancia y un ángulo * (https://es.wikipedia.org/wiki/Coordenadas_polares) . Habitualmente las funciones en coordenadas polares tienen la forma $r = f(\theta)$.

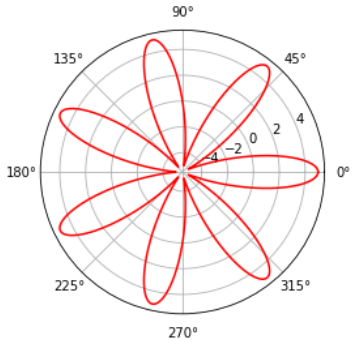
En Matplotlib se dispone de la función `polar`, la cual traza una gráfica en coordenadas polares, dados como argumentos tanto la variable independiente $\theta$ como la función $r$. Enseguida vamos a ver cómo graficar la tan conocida rosa polar, cuya ecuación general está dada por:

$$r = a\cos(k\theta + \phi_0)$$

Implementando esto en Python, se tiene:

In [20]:
```
theta = np.linspace(0, 2*np.pi, 1000)
a,k,phi0 = 5,7,0
r = a*np.cos(k*theta + phi0)
plt.polar(theta, r, "r")
```

Out[20]:
```
[<matplotlib.lines.Line2D at 0x1e01bc90fd0>]
```




Observe que la función `polar` funciona de manera bastante similar a `plot`, de hecho se le pueden pasar los mismos *keyword arguments* para personalizar el gráfico resultante.

## 1.5 Gráficas de barras

## 1.6 Gráficas de pastel

Las gráficas de barras nos sirven para representar porcentajes y proporciones. En Python podemos utilizar la librería Matplotlib para desarrollar este tipo de gráficas.

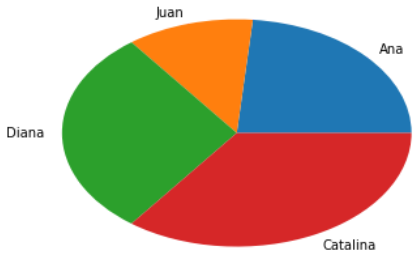
Matplotlib dispone de la función `pie`, cuya sintaxis depende del grado de personalización y control que se requiera sobre la gráfica de pastel a dibujar.

Para ejemplificar el uso de esta función vamos a suponer que se tienen los siguientes datos sobre algunas personas que tienen cierta cantidad de manzanas en *su poder*:

| Nombre | Manzanas |
|---|---|
| Ana | 20 |
| Juan | 10 |
| Diana | 25 |
| Catalina | 30 |

Para representar el porcentaje del total del cual dispone cada uno, podemos trazar una gráfica de pastel. Para ello realizamos lo siguiente:

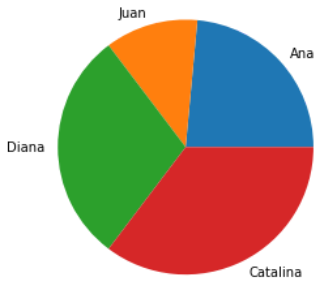
In [21]:
```
manzanas = [20,10,25,30]
nombres = ["Ana","Juan","Diana","Catalina"]
plt.pie(manzanas, labels=nombres);
```




Observe que lo primero que hacemos es importar la librería matplotlib, enseguida, en utilizando listas definimos los nombres y el número de manzanas correspondientes. Luego, la función `pie` acepta un primer argumento que contiene los valores absolutos de cada ítem, además, de un *keyword argument* `labels` que contiene las etiquetas correspondientes.

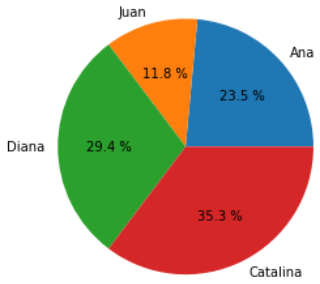
El aspecto *achatado* de la gráfica se puede solucionar utilizando la función `axis`.

In [22]:
```
manzanas = [20,10,25,30]
nombres = ["Ana","Juan","Diana","Catalina"]
plt.pie(manzanas, labels=nombres)
plt.axis("equal");
```



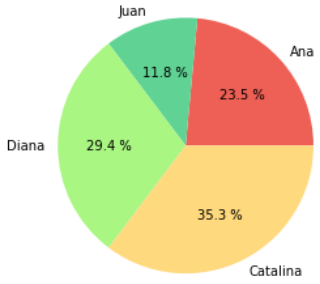

El porcentaje correspondiente a cada ítem se puede indicar mediante el argumento `autopct` :

In [23]:
```
manzanas = [20,10,25,30]
nombres = ["Ana","Juan","Diana","Catalina"]
plt.pie(manzanas, labels=nombres, autopct="%0.1f %%")
plt.axis("equal");
```



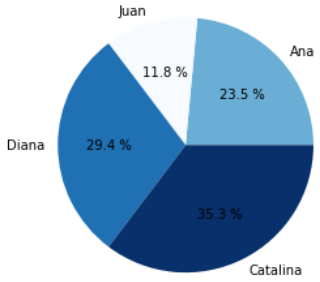

Los combinación de colores se puede especificar de manera manual, pasando una lista de color en formato hexadecimal o RGB.

In [24]:
```
manzanas = [20,10,25,30]
nombres = ["Ana","Juan","Diana","Catalina"]
colores = ["#EE6055","#60D394","#AAF683","#FFD97D","#FF9B85"]
plt.pie(manzanas, labels=nombres, autopct="%0.1f %%", colors=colores)
plt.axis("equal");
```



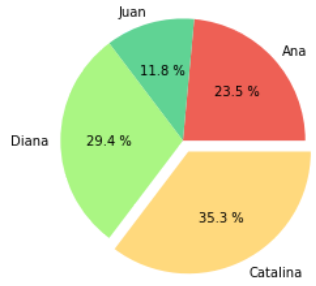

Los colores también se pueden determinar y autocalcular utilizando un mapa de color específico. Enseguida se muestra un ejemplo donde la variación es sobre colores en tonos azules.

In [25]:
```
from matplotlib import cm
from matplotlib import colors

manzanas = [20,10,25,30]
nombres = ["Ana","Juan","Diana","Catalina"]

normdata = colors.Normalize(min(manzanas), max(manzanas))
colormap = cm.get_cmap("Blues")
colores =colormap(normdata(manzanas))

plt.pie(manzanas, labels=nombres, autopct="%0.1f %%", colors=colores)
plt.axis("equal");
```




Es posible también segmentar o separar del bloque una o más de las *rebanadas* de la gráfica de pastel. Para ello se debe pasar una lista o tupla con valores entre 0 y n que indican el desfase respecto al centro, 0 indica ningún desfase y n un desfase equivalente a n*r, donde r es el radio de la gráfica de pastel.

In [26]:
```
manzanas = [20,10,25,30]
nombres = ["Ana","Juan","Diana","Catalina"]
colores = ["#EE6055","#60D394","#AAF683","#FFD97D","#FF9B85"]
desfase = (0, 0, 0, 0.1)
plt.pie(manzanas, labels=nombres, autopct="%0.1f %%", colors=colores, explode=desfase)
plt.axis("equal");
```




## 1.7 Gráficas de curvas paramétricas en el espacio

Una función vectorial de la forma:

$$\vec{r}(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix}$$

Se dice que es una función parámetrica, siendo $t$ en este caso el parámetro correspondiente. Una función vectorial de este tipo tiene una curva en el espacio asociada como representación gráfica. Es muy común trabajar con este tipo de expresiones en el análisis cinemático de partículas.

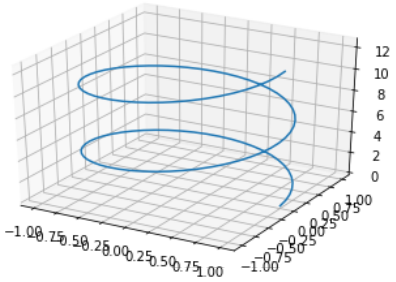
Supongamos que queremos graficar la función vectorial:

$$\vec{r}(t) = \begin{bmatrix} \cos(t) \\ \sin(t) \\ t \end{bmatrix}$$

En el intervalo $0 \leq t \leq 4\pi$. Para ello en Python haríamos lo siguiente:

In [27]:
```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

t = np.linspace(0, 4*np.pi, 100)
x = np.cos(t)
y = np.sin(t)
z = t

ax.plot(x, y, z)
```

Out[27]:
```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x1e01bded6d8>]
```

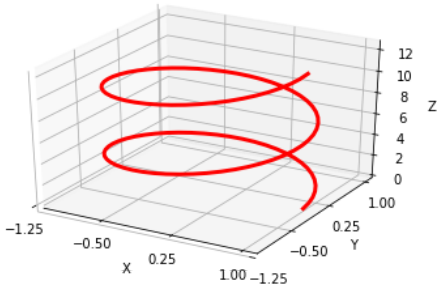


Ahora explicamos lo referente al código anterior. Observe que en la primera línea importamos la clase `Axes3D` del módulo `mpl_toolkits.maplot3d` , esto nos sirve para poder trabajar con gráficas tridimensionales. Luego, definimos un objeto de la clase `Figure` y lo asignamos a la variable `fig` , al objeto `fig` le añadimos un *Axes* mediante el método `add_subplot` , indicando que en dicho *axes* se utilizarán las proyecciones espaciales mediante el *keyword argument* `projection` . Las siguientes cuatro líneas definen las ecuaciones parámetricas. Y finalmente, con el método `plot` del objeto `ax` trazamos la gráfica de la curva tridimensional, note que en este caso el método `plot` , recibe al menos tres argumentos: las coordenadas en x, y, z.

Al igual que en las otros tipos de gráficos, podemos también manipular las características. Vea por ejemplo el siguiente código:

In [28]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

t = np.linspace(0, 4*np.pi, 100)
x = np.cos(t)
y = np.sin(t)
z = t

ax.plot(x, y, z, color="r", linewidth=3)
xticks = ax.get_xticks()
yticks = ax.get_yticks()
ax.set_xticks(xticks[::3])
ax.set_yticks(yticks[::3])
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
```

Out[28]:
```
Text(0.5,0,'Z')
```



## 1.8 Gráficas de superficies

In [ ]:
```
```

# Contenido



# 1 Sympy: un sistema de álgebra computacional

SymPy es un sistema de álgebra computacional (CAS) escrito en Python puro, está desarrollado con un enfoque en la extensibilidad y facilidad de uso, tanto a través de aplicaciones interactivas como programáticas. Estas características le han permitido a SymPy convertirse en una librería de cómputo simbólico muy popular en el ecosistema científico de Python.

En todo este capítulo se asumirá que se ha importado la librería SymPy, con las siguientes instrucciones:

In [1]:
```
from sympy import *
```

Adicionalmente dentro del entorno de Jupyter podemos ejecutar la siguiente instrucción que permite *renderizar* las expresiones resultantes.

In [2]:
```
init_printing()
```

## 1.1 Variables simbólicas

Las variables simbólicas son el *alma* de SymPy, todas las operaciones de álgebra simbólica se basan en estas. A una variable simbólica se le asigna un símbolo que la representa mediante la función `symbols`:

In [4]:
```
x = symbols("x")
print(type(x))
```

```
<class 'sympy.core.symbol.Symbol'>
```

Con la función `symbols` se define una nueva variable simbólica que se guarda en `x`, se verifica que se crea un objeto de la clase `Symbol`. Con la variable `x` ya definida se puede comenzar a formar expresiones algebraicas y manipular matemáticamente.

In [5]:
```
x + 2
```

Out[5]: $x + 2$

In [6]:
```
x**2 - 2*x - 10
```

Out[6]: $x^2 - 2x - 10$

In [7]:
```
sqrt(x) - sin(x)
```

Out[7]: $\sqrt{x} - \sin\left(x\right)$

Puede definir múltiples variables separando por comas cada representación simbólica:

In [8]:
```
p,q,r = symbols("p,q,r")
p**q + r/q
```

Out[8]: $p^{q} + \frac{r}{q}$

Además de la forma anterior, también es posible tener variables simbólicas disponibles si se importan del módulo `abc` de SymPy:

In [9]:
```
from sympy.abc import a,b,c,d
```

In [10]:
```
(a + b)*(c + d)
```

Out[10]: $\left(a + b\right)\left(c + d\right)$

## 1.2 Manipulación algebraica

SymPy es una poderosa herramienta de manipulación y simplificación algebraica, en lo subsiguiente se revisarán algunos casos elementales y se describirá el uso de las herramientas (funciones) que proporciona.

En primera instancia se definen algunas variables simbólicas a utilizar:

In [11]:
```
x,y,z = symbols("x,y,z")
a,b,c,d,k,m,n = symbols("a,b,c,d,k,m,n")
```

Para las expresiones algebraicas formadas en SymPy por default se *evalúan* y simplifican los términos semejantes. Vea los siguientes casos:

In [12]:
```
x**2 + 5*x**3 - 10*x**2 + 5*x - 10*(x + 1)
```

Out[12]: $5x^3 - 9x^2 - 5x - 10$

In [13]:
```
a*b + c*d + x/y - 7*a*b
```

Out[13]: $-6ab + cd + \frac{x}{y}$

Naturalmente para simplificaciones y operaciones un poco menos obvias, habrá que especificarle lo que se requiere.

Una de las operaciones elementales de simplificación en álgebra es la factorización, SymPy dispone de la función `factor`, la cual toma una expresión algebraica y la factoriza conforme sea posible. Por ejemplo, suponiendo que se tiene la expresión $ab + ac$, es sencillo identificar que se puede factorizar como $a(b + c)$, SymPy hace lo mismo sin sobresalto alguno.

In [14]:
```
factor(a*b + a*c)
```

Out[14]: $a\,(b + c)$

De igual forma se sabe que una expresión de la forma $ac + ad + bc + bd$ se puede factorizar como $(a + b)(d + c)$.

In [15]:
```
factor( a*c + a*d + b*c + b*d )
```

Out[15]: $(a + b)\,(c + d)$

Así, para un binomio al cuadrado se sabe que, $(a + b)(a + b)$ es la factorización de una expresión del tipo $a^2 + 2ab + b^2$.

In [16]:
```
factor(a**2 + 2*a*b + b**2)
```

Out[16]: $(a + b)^2$

¿Se puede hacer el proceso *inverso*, es decir, dados dos factores obtener su expresión *expandida*? Por supuesto, para ello se hace uso de la función `expand`.

In [17]:
```
expand( (x-7)*(x-3) )
```

Out[17]: $x^2 - 10x + 21$

SymPy puede manejar cualquier cantidad de factores y devolver la expresión que resulta de realizar la multiplicación algebraica, sin estar limitada siquiera por la cantidad de variables simbólicas.

In [18]:
```
expand( (x + y)*(x - y) )
```

Out[18]: $x^2 - y^2$

In [19]:
```
expand( (x + y)*(x+y) )
```

Out[19]: $x^2 + 2xy + y^2$

In [20]:
```
expand( (x + y)**2 )
```

Out[20]: $x^2 + 2xy + y^2$

In [21]:
```
expand( (x + y)**3 )
```

Out[21]: $x^3 + 3x^2y + 3xy^2 + y^3$

In [22]:
```
expand( a*(m + n)**2 )
```

Out[22]: $am^2 + 2amn + an^2$

De manera general, si requiere simplificar una expresión algebraica SymPy dispone de una función más o menos universal que funcionará en la mayoría de los casos: `simplify`. Por ejemplo, suponiendo que se tiene la siguiente expresión algebraica racional y se requiere reducir lo más posible:

$$\frac{x^2 - 3x}{x^2 + 3x}$$

Se puede notar que tanto en el numerador como en el denominador se puede factorizar $x$, lo cual conduce a:

$$\frac{x(x-3)}{x(x+3)} = \frac{x-3}{x+3}$$

Con SymPy:

In [25]:
```
simplify( (x**2 - 3*x)/(x**2 + 3*x) )
```

Out[25]: $\frac{x-3}{x+3}$

Lo mismo para una expresión que involucre funciones trigonométricas, por ejemplo, cualquiera que haya cursado matemáticas del nivel secundario, al menos, sabe que $\sin^2 x + \cos^2 x = 1$, y también *lo sabe* SymPy:

In [26]:
```
simplify( sin(x)**2 + cos(x)**2 )
```

Out[26]: $1$

Pero también *sabe* que $\cos x \cos y - \sin x \sin y = \cos(x+y)$.

In [23]:
```
simplify( cos(x)*cos(y) - sin(x)*sin(y) )
```

Out[23]: $\cos\left(x + y\right)$

O bien:

In [25]:
```
simplify( (1+tan(x)**2)/(1+cot(x)**2) )
```

Out[25]: $\tan^{2}\left(x\right)$

Habitualmente para la manipulación de expresiones que contienen funciones trigonométricas se suele utilizar la función `trigsimp` que en muchos casos lo que devuelva coincidirá con `simplify`. Sin embargo, si en lugar de reducir una expresión trigonométrica se requiere expandirla, como en el caso del coseno o seno de la suma de dos ángulos, probablemente por intuición se utilizaría `expand`, pero aquí no funciona como puede verificar.

In [26]:
```
expand( sin(x + y) )
```

Out[26]: $\sin\left(x + y\right)$

En este caso puede utilizar la función `expand_trig`, la cual maneja de mejor manera las manipulaciones trigonométricas:

In [28]:
```
expand_trig( sin(x+y) )
```

Out[28]: $\sin\left(x\right)\cos\left(y\right) + \sin\left(y\right)\cos\left(x\right)$

## 1.3 Resolviendo ecuaciones e inecuaciones

SymPy dispone de la función `solve`, la cual resuelve desde ecuaciones polinomiales, sistemas de ecuaciones lineales, inecuaciones, hasta sistemas de ecuaciones no lineales. La sintaxis de `solve` es polimórfica y en general depende de lo que se requiera resolver, tendiendo siempre a que sea posible especificar el mínimo número de parámetros.

En las siguientes subsecciones se describen algunos casos, se asumirá en lo subsiguiente que además de importar SymPy se definieron las siguientes variables simbólicas:

In [29]:
```
x,y,z = symbols("x,y,z")
a,b,c = symbols("a,b,c")
```

### 1.3.1 Ecuaciones polinómicas

La sintaxis más elemental de `solve` es pasando un sólo argumento, el cual se espera sea una expresión algebraica y se considera que esta estará igualada a cero. Por ejemplo para resolver la ecuación lineal $x - 3 = 0$:

In [30]:
```
solve( x - 3 )
```

Out[30]: $[3]$

Para una ecuación cuadrática $x^2 + 2x + 2 = 0$:

In [31]:
```
solve(x**2 + 2*x + 2)
```

Out[31]: $[-1 - i, \quad -1 + i]$

En este caso la ecuación tiene soluciones complejas, la unidad imaginaria en SymPy se especifica mediante el símbolo `I`.

Si se quisiera resolver la ecuación cuadrática en su forma general, por intuición y lo que sabemos hasta ahora, haríamos:

In [32]:
```
solve(a*x**2 + b*x + c)
```

Out[32]:
$$\left[\left\{a : -\frac{1}{x^2}(bx + c)\right\}\right]$$

Note que el problema está en que al haber más de una variable simbólica, SymPy no sabe con respecto a qué variable debe resolver y toma la primera en orden alfabético. Para especificar explícitamente respecto a qué variable resolver, se puede indicar mediante un segundo argumento:

In [39]:
```
solve(a*x**2 + b*x + c, x)
```

Out[39]:
$$\left[\frac{1}{2a}\left(-b + \sqrt{-4ac + b^2}\right), \quad -\frac{1}{2a}\left(b + \sqrt{-4ac + b^2}\right)\right]$$

Puede verificar que devuelve, en efecto, la tan conocida fórmula general.

### 1.3.2 Ecuaciones trigonométricas

## 1.4 Cálculo

### 1.4.1 Límites

Para calcular límites matemáticos SymPy dispone de la función `limit`, la cual requiere al menos tres argumentos de entrada: la función, variable y valor al que tiende. Por ejemplo, si se quiere calcular el siguiente límite:

$$\lim_{x \to 0} \frac{\sin x}{x}$$

Tendría que teclearse lo siguiente:

In [33]:
```
limit(sin(x)/x, x, 0)
```

Out[33]:
$$1$$

Para calcular limites laterales debe pasarse un cuarto argumento, por ejemplo:

In [34]:
```
limit(1/(x-5), x, 5, "-")
```

Out[34]:
$$-\infty$$

In [35]:
```
limit(1/(x-5), x, 5, "+")
```

Out[35]:
$$\infty$$

El símbolo `+` denota el cálculo de un límite lateral por la derecha y el símbolo `-` un límite lateral por la izquierda.

### 1.4.2 Derivadas

Las derivadas en Python se calculan utilizando la función `diff`, misma que en su forma más simple espera al menos dos argumentos: una expresión algebraica y una variable respecto a la cual derivar. Por ejemplo:

In [36]:
```
diff(exp(x)*cos(x), x)
```

Out[36]:
$$-e^x \sin(x) + e^x \cos(x)$$

Es posible también especificar una derivada de orden superior mediante un tercer argumento:

In [37]:
```
diff(5*x**2 + 3*x - 10, x, 2)
```

Out[37]:
$$10$$

La instrucción anterior calcula la segunda derivada de la función $f(x) = 5x^2 + 3x - 10$.

### 1.4.3 Integrales

Para calcular integrales vamos a utilizar la función `integrate`, la cual acepta al menos dos argumentos: la función a integrar y la expresión respecto a la cual se integra, por ejemplo:

In [38]:
```
integrate(x**2 + 3*x - 7, x)
```

Out[38]: $\frac{x^3}{3} + \frac{3x^2}{2} - 7x$

Esa instrucción calcula la integral:

$$\int \left(x^2 + 3x - 7\right)\, dx = \frac{x^3}{3} + \frac{3x^2}{2} - 7x + C$$

Note que la expresión algebraica devuelta por Python no contiene la constante de integración, por default SymPy no la considera. Sí en algún caso específico necesita referir a la constante de integración puede adicionarla manualmente.

Las integrales definidas se pueden calcular si el segundo argumento se hace una tupla de la forma `(variable, a, b)`, donde `a` y `b` indican el límite inferior y superior a evaluar en la integral:

Por ejemplo, para evaluar:

$$\int_0^{\pi} \sin x\, dx$$

In [41]:
```
integrate(sin(x), (x, 0, pi))
```

Out[41]: 2

O para:

$$\int_{-5}^{5} z^2\, dz$$

In [43]:
```
integrate(z**2, (z,-5,5))
```

Out[43]: $\frac{250}{3}$

## 1.5 Vectores

Un vector denota una cantidad física que tiene magnitud y orientación en un determinado sistema de referencia. En SymPy los vectores se pueden definir mediante la clase `Matrix` del módulo `matrices`, de tal manera que en primera instancia haría falta importar dicho módulo, para esto hacemos:

In [44]:
```
from sympy.matrices import Matrix
```

Ahora vamos a definir dos vectores $\vec{u}$ y $\vec{v}$:

$$\vec{u} = \begin{bmatrix} 2 \\ 1 \\ -5 \end{bmatrix} \quad ; \quad \vec{v} = \begin{bmatrix} 4 \\ -1 \\ 3 \end{bmatrix}$$

In [46]:
```
u = Matrix([2,1,-5])
v = Matrix([4,-1,3])
```

Una suma y resta vectorial se pueden ejecutar sin muchas complicaciones, mediante los operadores aritméticos ya conocidos.

In [47]:
```
u + v
```

Out[47]: $\begin{bmatrix} 6 \\ 0 \\ -2 \end{bmatrix}$

In [48]:
```
v - u
```

Out[48]: $\begin{bmatrix} 2 \\ -2 \\ 8 \end{bmatrix}$

La magnitud de un vector se puede calcular utilizando el método `norm`.

In [49]:
```
u.norm()
```

Out[49]: $\sqrt{30}$

In [50]:
```
v.norm()
```

Out[50]: $\sqrt{26}$

Recuerde que si requiere ver las expresiones resultantes como fracciones decimales debe usar `evalf`.

El producto escalar de dos vectores puede calcularlo utilizando el método `dot`, por ejemplo $\vec{u} \cdot \vec{v}$ lo puede especificar como:

In [51]:
```
u.dot(v)
```

Out[51]: $-8$

Para calcular el producto vectorial utilice el método `cross`, por ejemplo $\vec{u} \times \vec{v}$:

In [52]:
```
u.cross(v)
```

Out[52]: $\begin{bmatrix} -2 \\ -26 \\ -6 \end{bmatrix}$

Recuerde que el producto vectorial no es conmutativo, por tanto, $\vec{v} \times \vec{u}$ resultará en un vector diferente al obtenido anteriormente, como puede verificar enseguida:

In [53]:
```
v.cross(u)
```

Out[53]: $\begin{bmatrix} 2 \\ 26 \\ 6 \end{bmatrix}$

## 1.6 Matrices

Las matrices son arreglos rectangulares de números o cantidades simbólicas. En SymPy, se definen utilizando la clase `Matrix`, pasándole como argumentos una lista de listas, donde cada sublista corresponde a una fila de la matriz.

Por ejemplo, vamos a definir las matrices $A$, $B$ y $C$, dadas por:

$$A = \begin{bmatrix} 20 & 50 & 100 \\ 10 & 35 & 200 \\ -30 & 20 & 80 \end{bmatrix} \qquad B = \begin{bmatrix} 12 & 26 & 45 \\ 3 & -15 & 18 \\ 54 & 20 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} 5 & 9 \\ 2 & 3 \\ -10 & 8 \end{bmatrix}$$

Primero, importamos la clase `Matrix` del módulo `matrices`:

In [54]:
```
from sympy.matrices import Matrix
```

Luego escribiríamos:

In [55]:
```
A = Matrix([[20,50,100], [10,35,200], [-30,20,80]])
B = Matrix([[12,26,45],[3,-15,18],[54,20,0]])
C = Matrix([[5,9], [2,3], [-10,8]])
```

De manera muy sencilla podríamos realizar operaciones matriciales básicas, por ejemplo, una suma:

In [56]:
```
A + B
```

Out[56]: $\begin{bmatrix} 32 & 76 & 145 \\ 13 & 20 & 218 \\ 24 & 40 & 80 \end{bmatrix}$

Naturalmente, y como es esperable, SymPy *conoce* y aplica las reglas de álgebra de matrices, observe:

In [57]:
```
A + C
```

```
---------------------------------------------------------------------------
ShapeError                                Traceback (most recent call last)
<ipython-input-57-c3cb66996588> in <module>()
----> 1 A + C

~\Anaconda3\lib\site-packages\sympy\core\decorators.py in binary_op_wrapper(self, other)
    130                     else:
    131                         return f(self)
--> 132             return func(self, other)
    133         return binary_op_wrapper
    134     return priority_decorator

~\Anaconda3\lib\site-packages\sympy\matrices\common.py in __add__(self, other)
   1949             if self.shape != other.shape:
   1950                 raise ShapeError("Matrix size mismatch: %s + %s" % (
-> 1951                     self.shape, other.shape))
   1952
   1953         # honest sympy matrices defer to their class's routine

ShapeError: Matrix size mismatch: (3, 3) + (3, 2)
```

SymPy es muy explícito en ese tipo de situaciones y nos imprime un mensaje de error suficientemente descriptivo. De igual forma que es válido realizar el producto $AC$, pero no $CA$:

In [58]:
```
A*C
```

Out[58]:
$$\begin{bmatrix} -800 & 1130 \\ -1880 & 1795 \\ -910 & 430 \end{bmatrix}$$

In [59]:
```
C*A
```

```
---------------------------------------------------------------------------
ShapeError                                Traceback (most recent call last)
<ipython-input-59-cbae4c009e52> in <module>()
----> 1 C*A

~\Anaconda3\lib\site-packages\sympy\core\decorators.py in binary_op_wrapper(self, other)
    130                     else:
    131                         return f(self)
--> 132             return func(self, other)
    133         return binary_op_wrapper
    134     return priority_decorator

~\Anaconda3\lib\site-packages\sympy\matrices\common.py in __mul__(self, other)
   2006             if self.shape[1] != other.shape[0]:
   2007                 raise ShapeError("Matrix size mismatch: %s * %s." % (
-> 2008                     self.shape, other.shape))
   2009
   2010         # honest sympy matrices defer to their class's routine

ShapeError: Matrix size mismatch: (3, 2) * (3, 3).
```

El **determinante** de una matriz podemos calcularlo mediante la función `det`:

In [60]:
```
det(B)
```

Out[60]: $60102$

O bien, mediante el propio método `det`:

In [61]:
```
B.det()
```

Out[61]: $60102$

La **matriz inversa** podemos calcularla utilizando el método `inv`:

In [62]:
```
A.inv()
```

Out[62]:
$$\begin{bmatrix} \frac{6}{1195} & \frac{2}{239} & -\frac{13}{478} \\ \frac{34}{1195} & -\frac{23}{1195} & \frac{3}{239} \\ -\frac{5}{956} & \frac{19}{2390} & -\frac{1}{1195} \end{bmatrix}$$

In [63]:
```
A.inv()*A
```

Out[63]:
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

La **transpuesta** de una matriz se puede obtener accediendo al atributo `T`

In [64]:
```
C.T
```

Out[64]:
$$\begin{bmatrix} 5 & 2 & -10 \\ 9 & 3 & 8 \end{bmatrix}$$

## 1.7 Ejercicios

Utilice Python/SymPy para resolver los siguientes ejercicios:

1. Resuelva las siguientes ecuaciones (para la variable $x$):
   A. $x + 3x = 10$
   B. $2x^2 - 10x + 5 = 0$
   C. $\frac{a}{x} + bx - 8 = 0$
   D. $\cos x + \sin x = 10$
   E. $kx - 10 = x^2$
2. Calcule las siguientes derivadas:
   A. $\frac{d}{dx}(\cos x)$
   B. $\frac{d}{dt}\left(at^2 - 2\tan t - \frac{10}{t}\right)$
   C. $\frac{d}{dz}\left(z^5 - 10e^{-z}\right)$
3. Calcule las siguientes integrales indefinidas:
   A. $\int \cos x \, dx$
   B. $\int \left(x^3 - 8x\right) \, dx$
   C. $\int \left(e^{-3x} \sin x\right) \, dx$
   D. $\int \left(s^2 + ks - m\right) \, ds$
4. Calcule las siguientes integrales definidas:
   A. $\int_a^b \sin x \, dx$
   B. $\int_{-10}^{5} x^2 + 10x \, dx$
   C. $\int_0^2 10e^{-z} \, dz$
5. Resuelva la siguiente ecuación diferencial:
$$\frac{dS}{dr} = kS$$
6. Escriba una función que dados como entrada dos vectores, determine sí estos son paralelos.
7. Desarrolle una función llamada `calcula_fuerza_resultante` que reciba como datos de entrada un conjunto de vectores fuerza y devuelva el vector de fuerza resultante.
8. Escriba un función llamada `calcula_momento_resultante` que reciba como datos de entrada un conjunto de vectores fuerza y un conjunto de vectores de posición del punto de aplicación de la fuerza con respecto a un cierto punto. Se deberá devolver el momento total producido por las fuerzas con respecto al punto.

In [ ]:
```

```