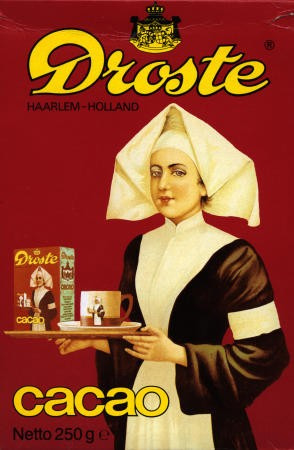
**11.2 Intro a la Recursión**

Para esta sección y las dos siguientes hay un [video](https://youtu.be/C0o-KuY-BNs) donde introducimos el tema de recursión y vemos algunos ejemplos.

**La recursión y cómo puede ser que funcione**

Estamos acostumbrados a escribir funciones que llaman a otras funciones. Pero lo cierto es que nada impide que en Python (y en muchos otros lenguajes) una función se llame a sí misma. Y lo más interesante es que esta propiedad, que se llama *recursión*, permite en muchos casos encontrar soluciones muy elegantes para determinados problemas.

En materias de matemática se estudian los razonamientos por inducción para probar propiedades de números enteros y la recursión no es más que una generalización de la inducción a más estructuras: las listas, las cadenas de caracteres, las funciones, etc.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/cartel.png)

A continuación estudiaremos diversas situaciones en las cuales aparece la recursión, veremos cómo es que esto puede funcionar, algunas situaciones en las que es conveniente utilizarla y otras situaciones en las que no.

**Una función recursiva matemática**

Es muy común tener definiciones inductivas de operaciones. Un caso paradigmático es la del factorial. Recordemos que el factorial de un número entero positivo n es el producto de todos los números entre 1 y n. La definición usual, inductiva, es la siguiente:

1! = 1

n! = n \* (n-1)! si n>1

Este tipo de definición se traduce naturalmente en una función en Python:

def factorial(n):

"""Precondición: n entero positivo

Devuelve: n!"""

if n == 1:

return 1

return n \* factorial(n - 1)

Ésta es la ejecución del factorial para n = 1 y para n = 3.

>>> factorial(1)

1

>>> factorial(3)

6

El sentido de la instrucción n \* factorial(n - 1) es exactamente el mismo que el de la definición inductiva: para calcular el factorial de n se debe multiplicar n por el factorial de n-1.

Dos piezas fundamentales para garantizar el funcionamiento de este programa son:

* Que se defina un *caso base* (en este caso la indicación *no recursiva* de cómo calcular factorial(1)).
* Que el argumento de la función respete la precondición de que n debe ser un entero mayor o igual que 1.

No es increíble que esto pueda funcionar adecuadamente en un lenguaje de programación?

**Ejercicio 11.1:**

Para poder analizar qué sucede a cada paso de la ejecución de la función, utilizaremos una versión más detallada del mismo código, en la que el resultado de cada paso se asigna a una variable.

def factorial(n):

if n == 1:

r = 1

return r

f = factorial(n-1)

r = n \* f

return r

Esta porción de código funciona exactamente igual que la anterior, pero nos permite ponerles nombres a los resultados intermedios de cada operación para poder estudiar qué sucede a cada paso. Analizá con tu debugger la ejecución de factorial(3).

**Algoritmos recursivos y algoritmos iterativos**

Llamaremos *algoritmos recursivos* a aquellos que realizan llamadas recursivas para llegar al resultado, y *algoritmos iterativos* a aquellos que llegan a un resultado a través de una iteración mediante un ciclo definido o indefinido.

Todo algoritmo recursivo puede expresarse como iterativo y viceversa. Sin embargo, según las condiciones del problema a resolver podrá ser preferible utilizar la solución recursiva o la iterativa.

Una posible implementación iterativa de la función factorial vista anteriormente sería:

def factorial(n):

"""Precondición: n entero positivo

Devuelve: n!"""

fact = 1

for num in range(n, 1, -1):

fact \*= num

return fact

Se puede ver que en este caso no es necesario incluir un caso base, ya que el mismo ciclo incluye una condición de corte, pero que sí es necesario incluir un acumulador, que en el caso recursivo no era necesario.

Por otro lado, si hiciéramos el seguimiento de esta función, como se hizo para la versión recursiva, veríamos que la pila de ejecución siempre tiene un único marco, en el cual se van modificando los valores de num y fact.

Es por esto que, en general, las versiones recursivas de los algoritmos utilizan más memoria (ya que la pila de ejecución se guarda en memoria) pero suelen ser más elegantes.

**Un ejemplo de recursión elegante**

Consideremos ahora otro problema que puede ser resuelto de forma elegante mediante un algoritmo recursivo.

La función potencia(b, n) que vimos cuando hablamos de invariantes en la [Sección 7.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/05_Especificacion_y_Documentacion.md#invariantes-de-ciclo), realiza n iteraciones para poder obtener el valor de b^n.

Sin embargo, es posible optimizarla teniendo en cuenta los siguientes hechos:

b^n = b^(n/2) \* b^(n/2) si n es par, y

b^n = b^((n-1)/2) \* b^((n-1)/2) \* b si n es impar.

Estas ecuaciones nos permiten diseñar un algoritmo muchísimo más eficiente. Esta situación guarda cierta analogía con el problema de la búsqueda en una lista ordenada. La idea es, en un paso, reducir *el tamaño* del problema a la mitad.

Antes de programar cualquier función recursiva es necesario decidir cuál será el *caso base* y cuál el *paso recursivo*. Para esta función, tomaremos n=0 como el caso base (devolveremos 1). El paso recursivo tendrá dos partes, correspondientes a los dos posibles grupos de valores de n.

def potencia(b,n):

"""Precondición: n >= 0

Devuelve: b^n."""

if n <= 0:

# caso base

return 1

if n % 2 == 0:

# caso n par

p = potencia(b, n // 2)

return p \* p

else:

# caso n impar

p = potencia(b, (n - 1) // 2)

return p \* p \* b

El uso de la variable p en este caso no es optativo, ya que es una de las ventajas principales de esta implementación: se aprovecha el resultado calculado en lugar de tener que calcularlo dos veces. Vemos que este código funciona correctamente:

>>> potencia(2, 10)

1024

>>> potencia(3, 3)

27

>>> potencia(5, 0)

1

El orden de las llamadas, haciendo un seguimiento simplificado de la función será:

potencia(2, 10)

potencia(2, 5)

potencia(2, 2)

potencia(2, 1)

potencia(2, 0)

return 1

return 1 \* 1 \* 2

return 2 \* 2

return 4 \* 4 \* 2

return 32 \* 32

return 1024

Se puede ver, entonces, que para calcular 2^10 se realizaron 5 llamadas a potencia, mientras que en la implementación más sencilla se realizaban 10 iteraciones. Y esta optimización será cada vez más importante a medida que aumenta n: por ejemplo para n = 100 se realizarán 8 llamadas recursivas, y para n = 1000 11 llamadas.

Es posible transformar este algoritmo recursivo en un algoritmo iterativo. Para ello es necesario *simular* la pila de llamadas a funciones mediante una pila que almacene los valores que sean necesarios. En este caso, lo que apilaremos será si el valor de n es par o no.

def potencia(b, n):

"""Precondición: n >= 0

Devuelve: b^n."""

pila = []

while n > 0:

if n % 2 == 0:

pila.append(True)

n //= 2

else:

pila.append(False)

n = (n - 1) // 2

p = 1

while pila:

es\_par = pila.pop()

if es\_par:

p \*= p

else:

p \*= p \* b

return p

Como se puede ver, este código es mucho más complejo que la versión recursiva. Esto se debe a que utilizando recursión el uso de la pila de llamadas a funciones oculta el proceso de apilado y desapilado y permite concentrarse en la parte importante del algoritmo.

**Un ejemplo de recursión poco eficiente**

Del ejemplo anterior se podría deducir que siempre es mejor utilizar algoritmos recursivos; sin embargo --como dijimos antes-- cada situación merece ser analizada por separado.

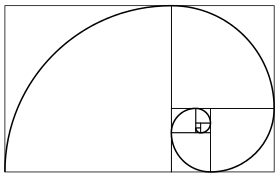
Un ejemplo clásico en el cual la recursión tiene un resultado muy poco eficiente es el de los números de Fibonacci. La sucesión de Fibonacci está definida por la siguiente relación:

F(0) = 0

F(1) = 1

F(n) = F(n - 1) + F(n - 2) si n > 1

Los primeros números de esta sucesión son: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. La sucesión tiene numerosas aplicaciones en computación y matemática y también aparece en configuraciones biológicas, como en las flores de girasoles, en la configuración de los ananás o las piñas de las coníferas, en la reproducción de conejos, etc. La siguiente imagen muestra su uso para aproximar una espiral áurea:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/280px-Fibonacci_spiral_34.bmp)

Dada la definición recursiva de la sucesión, puede resultar muy tentador escribir una función que calcule en valor de fib(n) de la siguiente forma:

def fib(n):

"""Precondición: n >= 0.

Devuelve: el número de Fibonacci número n."""

if n == 0:

res = 0

elif n == 1:

res = 1

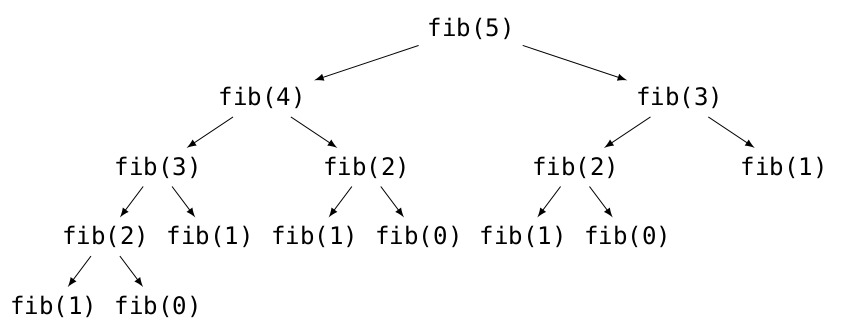
else:

res = fib(n - 1) + fib(n - 2)

return res

Si bien esta implementación es muy sencilla y elegante, también es extremadamente poco eficiente: para calcular fib(n - 1) es necesario calcular fib(n - 2), que luego volverá a ser calculado para obtener el valor fib(n).

Por ejemplo, una simple llamada a fib(5), generaría recursivamente todas las llamadas ilustradas en el siguiente gráfico. Puede verse que muchas de estas llamadas están repetidas, generando un total de 15 llamadas a la función fib, sólo para devolver el valor F(5).

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/fib5.jpg)

En este caso, será mucho más conveniente utilizar una versión iterativa, que vaya almacenando los valores de las dos variables anteriores a medida que los va calculando.

def fib(n):

"""Precondición: n >= 0.

Devuelve: el número de Fibonacci número n."""

if n == 0 or n == 1:

return n

ant2 = 0

ant1 = 1

for i in range(2, n + 1):

fibn = ant1 + ant2

ant2 = ant1

ant1 = fibn

return fibn

Vemos que el caso base es el mismo para ambos algoritmos, pero que en el caso iterativo se calcula el número de Fibonacci de forma incremental, de modo que para obtener el valor de fib(n) se harán n-1 iteraciones.

*En resumen*: vimos que un algoritmo recursivo **no** es necesariamente mejor que uno iterativo, ni viceversa. En cada situación es conveniente analizar cuál algoritmo provee una solución más clara y eficiente.

**11.3 Diseño de algoritmos recursivos**

Hasta el momento vimos que hay muchas funciones matemáticas que se definen o que pueden desarrollarse de forma recursiva, pero puede aplicarse recursividad a muchos problemas que no sean explicitamente recursivos. Diseñar un algoritmo recursivo es un proceso sistematizable.

En general, en el proceso para plantear un algoritmo recursivo, necesitamos resolver estos tres problemas:

1. **Caso base:** Necesitamos definir uno o más casos base de acuerdo a nuestro problema. Como regla general tratamos de pensar como caso base a las condiciones sobre las cuales es más fácil resolver nuestro problema. Por ejemplo, si estuviéramos trabajando sobre listas o cadenas probablemente sepamos la respuesta a nuestro problema en el caso de una secuencia vacía; si estuviéramos trabajando sobre conjuntos de elementos probablemente la respuesta sea evidente para los conjuntos de un solo elemento.
2. **Caso recursivo** o caso general: Este es el caso que va a efectuar la llamada recursiva. La idea de este caso es reducir el problema a un problema más sencillo, del cual se hará cargo la llamada recursiva, y luego poder ensamblar la solución al problema original. Ampliaremos esto más adelante.
3. **Convergencia:** Necesitamos que la reducción que se haga en el caso recursivo converja hacia los casos base, de modo que la recursión alguna vez termine. Esto es, si dijimos que el caso base se resolvía cuando teníamos una lista vacía, las operaciones del caso recursivo tienen que reducir reiteradamente la lista hasta que la misma quede vacía.

Si podemos hacer estas tres cosas, tendremos un algoritmo recursivo para nuestro problema.

**Un primer diseño recursivo**

Supongamos que queremos programar una función sumar(lista) que determine en forma recursiva la suma de una secuencia lista de números.

Como caso base debemos elegir un caso sencillo de verificar. El caso más sencillo de verificar es uno en el que ni siquiera necesitamos computar algo: Si la lista está vacía es evidente que la suma da cero.

Nuestro caso base será algo así como:

if len(lista) == 0:

return 0

Queremos diseñar un paso recursivo que realice *una reducción* de manera que dada cualquier lista, la aplicación sucesiva de la reducción seleccionada converja al caso base. Hay muchas maneras de reducir una lista para lograr esto; para este caso vamos a proponer una muy sencilla: sacar el primer elemento. Si cada llamada recursiva saca el primer elemento, tarde o temprano covergeremos a una lista vacía.

Nuestra llamada recursiva podría ser algo así como:

sumar(lista[1:])

Lo más complejo ahora es pensar el caso general.

Dijimos que íbamos a retirar un elemento de la lista por vez y hacer una llamada recursiva. Olvidémonos por un momento de la recursividad e imaginemos que *ya* tenemos una función sumar2 que sabe sumar los elementos de una lista y que lo hace bien. Intentemos resolver el problema inverso: si agregamos un elemento x al principio de la lista (obteniendo [x] + lista), ¿podemos calcular la suma de la nueva lista? ¿Podemos resolver el problema más grande con la solución al problema más pequeño? La solución es sencilla: La suma de la lista ampliada será x más la suma de la lista original (que podemos calcular como sumar2(lista)).

Es decir, la solución al problema este que planteamos sería así:

def sumar(lista):

"""Precondición: len(lista) >= 1.

Devuelve: La suma de los elementos en la lista."""

return lista[0] + sumar2(lista[1:])

Podemos ver que si tuviéramos implementada sumar2 entonces sumar funcionaría bien. Volvamos ahora a recursividad: Si sabemos resolver el caso general en función a la solución del caso simplificado de la llamada recursiva, si existen casos base que corten la recursión y si además la recursión converge hacia los casos base tenemos resuelto el problema completo. La función que asumimos que funcionaba *es la misma* que acabamos de implementar.

Cuando diseñamos una función recursiva tenemos que dar este *salto de fé*: asumir que la función del paso recursivo ya funciona; nosotros lo que vamos a implementar es una función que logra concatenar el resultado del subproblema y ensamblarlo con nuestro problema mayor. Si hacemos esto bien entonces todo funciona.

Finalmente nuestra primera función recursiva quedaría:

def sumar(lista):

"""Devuelve la suma de los elementos en la lista."""

res = 0

if len(lista) != 0:

res = lista[0] + sumar(lista[1:])

return res

**Recursión de cola**

Dentro de los problemas recursivos no siempre es inmediato establecer cómo se va a propagar la información entre las llamadas recursivas. Es decir, cómo va a interactuar la solución de el o los subproblemas en la solución del problema general.

En todos los ejemplos presentados hasta el momento la información del resultado se propagó desde las hojas del árbol de llamadas (los casos base) hacia las funciones invocantes (mediante la instrucción return). Por ejemplo, para resolver el resultado de Fibonacci F(5) se utilizan únicamente los resultados computados por F(4) y F(3), y no se recibe ningún dato adicional de la función invocante (más allá del parámetro n=5). Esto no siempre es así, en algunos problemas sí se hace necesario propagar información "hacia abajo". Y en otros casos, si bien no es necesario, puede tener ventajas adicionales.

Por ejemplo, podríamos reescribir la función sumar() de esta forma:

def sumar(lista, suma = 0):

"""Devuelve la suma de los elementos en la lista."""

res = suma

if len(lista) != 0:

res = sumar(lista[1:], lista[0] + suma)

return res

Puede observarse que en esta implementación en vez de *esperar* a que se resuelva el cómputo de la parte recursiva para ensamblar la solución e ir resolviendo los cálculos parciales desde el final de la lista hacia el principio, le *pasamos* la solución parcial a la llamada recursiva. Finalmente el caso base devuelve la suma de los cálculos que se realizaron de principio a final y cada llamada recursiva devuelve este resultado.

No profundizaremos más en el tema, pero la particularidad de que lo último que se realice en el caso general sea la llamada recursiva (sin realizar ninguna operación adicional sobre el resultado de esta llamada) se conoce como *recursividad de cola*. La recursividad de cola es de interés porque implica muy poco esfuerzo reescribir una versión iterativa y no recursiva del algoritmo. Esto es inmediato: como lo último que se hace es la llamada recursiva entonces no hace falta seguir *recordando* el contexto de la llamada anterior cuando se hace la siguiente, entonces no es necesario utilizar la pila de ejecución. El código anterior puede reescribirse como

def sumar(lista):

"""Devuelve la suma de los elementos en la lista."""

suma = 0

while lista:

lista, suma = lista[1:], lista[0] + suma

return suma

tan solo reemplazando la recursión por un bucle y actualizando las variables según los parámetros de la llamada recursiva.

**Modificación de la firma**

La *firma* de una función es su nombre, más los parámetros que recibe, más los valores que devuelve. Para invocar una función cualquiera, es suficiente con saber cómo es su firma, y no es necesario saber cómo es la implementación interna. Ahora bien, si cambiamos la lista de parámetros o el tipo de dato del valor de retorno de la función, estamos cambiando su firma, y eso nos obliga a cambiar cualquier lugar del código que contenga alguna llamada a la función.

En el ejemplo de sumar implementada con recursividad de cola nos vimos obligados a modificar la firma de la función agregando el parámetro suma que no formaba parte del problema inicial. Pudimos hacerlo elegantemente utilizando un valor por omisión (suma=0), pero la firma de todos modos quedó confusa.

Hay casos en los que no podemos salvar un cambio en la firma. Por ejemplo, supongamos que queremos diseñar una función recursiva que calcule el promedio de una secuencia de números.

Como ya sabemos diseñar funciones recursivas, intuimos que el caso base será cuando la lista tenga un solo elemento y que la reduciremos sacando de a un elemento por vez. El cuerpo de nuestra función será algo así:

def promediar(lista):

if len(lista) == 1:

res = lista[0]

else:

res = promediar(lista[1:]) ???

return res

Ahora bien, con esto no alcanza para resolver el problema.

Para calcular un promedio necesitamos tanto calcular la suma como contar la cantidad de elementos. Entonces, una implementación recursiva va a estar computando dos valores cuando el resultado del problema es evidentemente uno solo. Si bien puede elaborarse una solución similar a la que ya ensayamos con sumar complicaría innecesariamente el código. Es preferible modificar la firma de la función.

Implementemos el problema resolviendo primero la llamada recursiva (en una función diferente que llamaremos promediar\_aux()) y luego ensamblando:

def promediar\_aux(lista):

suma = lista[0]

cantidad = 1

if len(lista) > 1:

suma\_resto, cantidad\_resto = promediar\_aux(lista[1:])

suma += suma\_resto

cantidad += cantidad\_resto

return suma, cantidad

Puede verse que esta función cumple con las reglas de diseño de recursividad que describimos antes. Con lo que no cumple esta función es con la firma natural de la función promediar() que queríamos diseñar, ya que promediar\_aux() devuelve dos cosas y no una.

Esto no invalida nuestra solución, pero la misma está incompleta. Lo que debemos hacer es implementar una función *wrapper* (envoltorio) que lo que haga es operar como *cara visible* para le usuarie de la función que hace realmente el trabajo. A esta función sí la vamos a llamar promediar, ya que va a cumplir con la firma deseada:

def promediar(lista):

"""Devuelve el promedio de los elementos de una lista de números."""

def promediar\_aux(lista):

suma = lista[0]

cantidad = 1

if len(lista) > 1:

suma\_resto, cantidad\_resto = promediar\_aux(lista[1:])

suma += suma\_resto

cantidad += cantidad\_resto

return suma, cantidad

suma, cantidad = promediar\_aux(lista)

return suma / cantidad

Notar que si bien la función visible promediar no es recursiva, sí lo es la función promediar\_aux que es la que realiza el trabajo, por lo que el conjunto se considera recursivo. Observá que estamos definiendo la función promediar\_aux **dentro** de la función promediar de manera que no resulte visible desde afuera (no la podés llamar desde afuera: así como hay *variables locales*, ésta es una *función local*).

Además de para adaptar la firma de la función recursiva, las funciones wrapper se suelen utilizar para simplificar el código de las funciones recursivas. Por ejemplo, si quisiéramos hacer validaciones de los parámetros, no querríamos que las mismas se reiteraran en cada iteración recursiva porque consumirían recursos innecesarios. Entonces las podemos resolver en la función wrapper, antes de empezar la recursión.

Por ejemplo, hace un rato implementamos la potencia en forma recursiva con la restricción n >= 0. Pero dado que b^n = (1/b)^(-n) podemos aprovechar el código implementado para resolver para cualquier n entero. Podríamos modificar el código de potencia() para incluir este caso, pero se reiteraría la comprobación en cada nivel de la recursión. Para este caso resulta más sencillo construir una función wrapper e incluir ahí todo lo que consideremos necesario. Habiendo renombrado la función original como \_potencia, nuestro wrapper sería:

def potencia(b, n):

"""Precondición: n es entero

Devuelve: b^n."""

if n < 0:

b = 1 / b

n = -n

return \_potencia(b, n)

**Limitaciones**

Si creamos una función sin *caso base*, obtendremos el equivalente recursivo de un bucle infinito.

Éste es un bucle infinito y corre para siempre.

i = 0

while i < 10:

suma = suma + i

El bucle recursivo infinito, sin embargo, termina agotando la memoria. Esto se debe a que cada llamada recursiva agrega un elemento a la pila de llamadas a funciones y la memoria de nuestras computadoras no es infinita.

En particular, en Python, para evitar que la memoria se termine, la pila de ejecución de funciones tiene un límite. Es decir, que si se ejecuta un código como el que sigue:

def inutil(n):

return inutil(n - 1)

Se obtendrá un resultado como el siguiente:

>>> inutil(1)

File "<stdin>", line 2, in inutil

File "<stdin>", line 2, in inutil

(...)

File "<stdin>", line 2, in inutil

RecursionError: maximum recursion depth exceeded

El límite por omisión es de 1000 llamadas recursivas. Es posible modificar el tamaño máximo de la pila de recursión mediante la instrucción sys.setrecursionlimit(n). Sin embargo, si se está alcanzando este límite suele ser una buena idea pensar si realmente el algoritmo recursivo es el que mejor resuelve el problema.

**Sabías que**

Existen algunos lenguajes *funcionales*, como Haskell, ML, o Scheme, en los cuales la recursión es la única forma de realizar un ciclo. Es decir, no existen construcciones while ni for.

Estos lenguajes cuentan con optimización de recursión de cola, una optimización para que cuando se identifique que la recursión es de cola, no se apile el estado de la función innecesariamente, evitando el consumo adicional de memoria mencionado anteriormente.

La ejecución de todas las funciones con recursión de cola vistas en esta unidad podría ser optimizada por el compilador o intérprete del lenguaje.

**Resumen**

* A medida que se realizan llamadas a funciones, el estado de cada función se almacena en la *pila de ejecución*.
* Esto permite que sea posible que una función se llame a sí misma, pero que las variables dentro de la función tomen distintos valores.
* La *recursión* es el proceso en el cual una función se invoca a sí misma. Este proceso permite crear un nuevo tipo de ciclos.
* Siempre que se escribe una función recursiva es importante considerar el *caso base* (el que detendrá la recursión) y el *caso recursivo* (el que realizará la llamada recursiva). Una función recursiva sin caso base es equivalente a un bucle infinito.
* Una función no es mejor ni peor por ser recursiva. En cada situación a resolver puede ser conveniente utilizar una solución recursiva o una iterativa. Para elegir una o la otra será necesario analizar las características de elegancia y eficiencia.
* Al diseñar funciones recursivas muchas veces puede ser útil implementar una función *wrapper*, por ejemplo para adaptar la firma de la función, validar parámetros, inicializar datos o manejar excepciones.

**11.4 Práctica de Recursión**

**Ejercicios**

**Ejercicio 11.2: Números triangulares**

Escribí una función que calcule recursivamente el n-ésimo número triangular (es decir, el número *1 + 2 + 3 + ... + n*).

Fijate que este ejercicio es un caso particular de la función sumar\_enteros(desde, hasta) que implementaste en el [Ejercicio 7.6](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/05_Especificacion_y_Documentacion.md#ejercicio-76-sumas). La implementación que hiciste en el primer inciso de ese ejercicio es una forma de reemplazar la recursión por un ciclo. La implementación que hiciste en el segundo inciso es mucho más eficiente.

**Ejercicio 11.3: Dígitos**

Escribí una función recursiva que reciba un número positivo, n, y devuelva la cantidad de dígitos que tiene.

**Ejercicio 11.4: Potencias**

Escribí una función recursiva que reciba 2 enteros, *n* y *b*, y devuelva True si *n* es potencia de *b*.

Ejemplos:

es\_potencia(8, 2) -> True

es\_potencia(64, 4) -> True

es\_potencia(70, 10) -> False

es\_potencia(1, 2) -> True

**Ejercicio 11.5: Subcadenas**

Escribí una funcion recursiva que reciba como parámetros dos cadenas *a* y *b*, y devuelva una lista con las posiciones en donde se encuentra *b* dentro de *a*.

Ejemplo:

posiciones\_de('Un tete a tete con Tete', 'te') -> [3, 5, 10, 12, 21]

**Ejercicio 11.6: Paridad**

Escribí dos funciones mutualmente recursivas par(n) e impar(n) que determinen la paridad del numero natural dado, usando solo que:

* 1 es impar.
* Un número mayor que uno es impar (resp. par) si su antecesor es par (resp. impar).

**Ejercicio 11.7: Máximo**

Escribí una funcion recursiva que encuentre el mayor elemento de una lista (sin usar max()).

**Ejercicio 11.8: Replicar**

Escribí una función recursiva para replicar los elementos de una lista una cantidad n de veces. Por ejemplo:

replicar([1, 3, 3, 7], 2) -> ([1, 1, 3, 3, 3, 3, 7, 7])

*Sugerencia: hacé la recursión en el largo de la lista.*

**Ejercicio 11.9: Pascal**

El [triángulo de Pascal](https://es.wikipedia.org/wiki/Tri%C3%A1ngulo_de_Pascal) es un arreglo triangular de números que se define de la siguiente manera: Las filas se enumeran desde *n = 0*, de arriba hacia abajo. Los valores de cada fila se enumeran desde *k = 0* (de izquierda a derecha). Los valores que se encuentran en los bordes del triángulo son todos unos. Cualquier otro valor se calcula sumando los dos valores contiguos de la fila de arriba.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/PascalTriangleAnimated2.gif)

Escribí la función recursiva pascal(n, k) que calcula el valor que se encuentra en la fila n y la columna k. Guardá tu función en el archivo larenga.py para entregar.

Ejemplo:

>>> pascal(5, 2)

10

**Ejercicio 11.10: Combinatorios**

Escribí una función recursiva que reciba una lista de caracteres únicos, y un número *k*, e imprima todas las posibles cadenas de longitud *k* formadas con los caracteres dados (permitiendo caracteres repetidos).

Ejemplo:

>>> combinaciones(['a', 'b', 'c'], 2)

aa ab ac ba bb bc ca cb cc

**Ejercicio 11.11: Búsqueda binaria**

Escribí una función recursiva que implemente la búsqueda binaria de un elemento e en una lista ordenada lista. La función debe devolver simplemente True o False indicando si el elemento está o no en la lista. Para esto completá el siguiente código:

def bbinaria\_rec(lista, e):

if len(lista) == 0:

res = False

elif len(lista) == 1:

res = lista[0] == e

else:

medio = len(lista)//2

# completar

return res

Guardá tu solución en el archivo bbin\_rec.py.

**Ejercicio 11.12: Envolviendo a Fibonacci**

Como vimos, la implementación recursiva inmediata del cálculo del número de Fibonacci (F(n) = F(n-1) + F\_(n-2), F(0) = 0, F(1)= 1) es ineficiente porque muchas de las ramas calculan reiteradamente los mismos valores.

Escribí una función fibonacci(n) que calcule el *n*-ésimo número de Fibonacci de forma recursiva pero que utilice un diccionario para almacenar los valores ya computados y no computarlos más de una vez.

*Observación*: Será necesario implementar una función *wrapper* (es decir, una función que envuelva a otra) para cumplir con la firma de la función pedida. Podés trabajar en un script en blanco o completar el siguiente código.

def fibonacci(n):

"""

Toma un entero positivo n y

devuelve el n-ésimo número de Fibonacci

donde F(0) = 0 y F(1) = 1.

"""

def fibonacci\_aux(n, dict\_fibo):

"""

Calcula el n-ésimo número de Fibonacci de forma recursiva

utilizando un diccionario para almacenar los valores ya computados.

dict\_fibo es un diccionario que guarda en la clave 'k' el valor de F(k)

"""

if n in dict\_fibo.keys():

F = dict\_fibo[n]

else:

?? # completar

return ?? # completar

dict\_fibo = {0:0, 1:1}

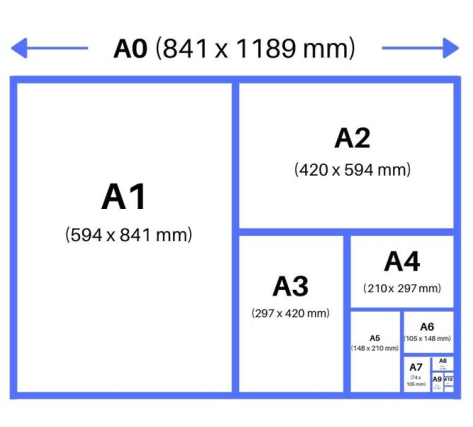
F, dict\_fibo = fibonacci\_aux(n, dict\_fibo)

return F

Guardala en el archivo fibonacci\_envuelto.py.

**Ejercicio 11.13: Hojas ISO y recursión**

La norma ISO 216 especifica tamaños de papel. Es el estándar que define el popular tamaño de papel A4 (210 mm de ancho y 297 mm de largo). Las hojas A0 miden 841 mm de ancho y 1189 mm de largo. A partir de la A0 las siguientes medidas, digamos la A(N+1), se definen doblando al medio la hoja A(N). Siempre se miden en milímetros con números enteros: entonces la hoja A1 mide 594 mm de ancho (y no 594.5) por 841 mm de largo.

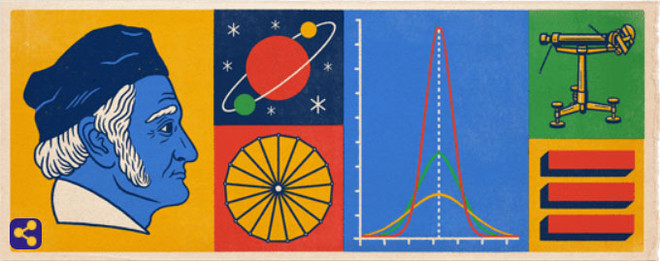
[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/AN.png)

Escribí una función recursiva que para una entrada N mayor que cero, devuelva el ancho y el largo de la hoja A(N) calculada recursivamente a partir de las medidas de la hoja A(N−1), usando la hoja A0 como caso base.

Guardala en el archivo hojas\_ISO.py.

**11.5 Regresión lineal**

En esta sección vamos a trabajar con **regresión lineal**, tema que introducimos en [este video](https://youtu.be/8tKruBXMDZY). No es una clase con todos los fundamentos del tema, sino un acercamiento práctico a las técnicas y sus formas de uso en Python. Para un desarrollo más profundo te recomendamos por ejemplo las notas de [Andrew Ng](http://cs229.stanford.edu/notes2020spring/cs229-notes1.pdf).

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/gauss.jpg)

**Regresión lineal simple**

Supongamos que queremos modelar la relación entre dos variables reales mediante un modelo lineal. Y que vamos a ajustar los parámetros de ese modelos a partir de ciertos valores conocidos (mediciones, digamos). Es decir que vamos a estar pensando que las variables tienen una relación lineal, Y = a\*X + b, donde X es la variable *explicativa* (sus componentes se denominan *independientes* o *regresores*), e Y es la variable *a explicar* (también denominada *dependiente* o *regresando*).

A partir de un conjunto de datos de tipo (x\_i, y\_i), planteamos el modelo Y = a\*X + b.

En general el modelo no va a ser exacto, es decir, no se va a complir que y\_i = a\*x\_i + b para los valores (x\_i, y\_i) (salvo que estén, justamente, todos los valores sobre una línea recta). En general, decíamos, vamos a tener que y\_i = a\*x\_i + b + r\_i donde, los valores r\_i, llamados *residuos*, representan las diferencias entre los valores de la recta en cada valor de x que tenemos y los valores de y asociados.

El problema de regresión lineal consiste en elegir los parámetros a, b de la recta (es decir, su pendiente y ordenada al origen), de manera que la recta sea la que *mejor se adapte* a los datos disponibles.

import numpy as np

import matplotlib.pyplot as plt

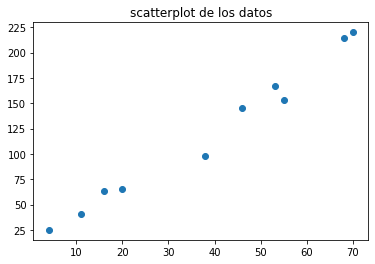
x = np.array([55.0, 38, 68, 70, 53, 46, 11, 16, 20, 4])

y = np.array([153.0, 98, 214, 220, 167, 145, 41, 63, 65, 25])

g = plt.scatter(x = x, y = y)

plt.title('scatterplot de los datos')

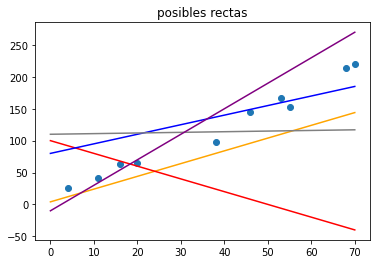
plt.show()

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/ej0_scatter.png)

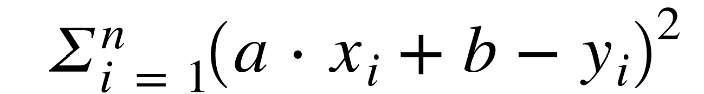
¿Qué quiere decir *mejor*? Vamos a considerar el criterio de cuadrados mínimos.

**Criterio de cuadrados mínimos**

Vamos a elegir como mejor recta a la que minimice los residuos. Más precisamente, vamos a elegir la recta de manera tal que la suma de los cuadrados de los residuos sea mínima.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/ej0_posiblesrectas.png)

Analíticamente, buscamos a, b tales que minimicen la siguiente suma de cuadrados:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/eq_suma_de_cuadrados.png)

Usar cuadrados mínimos tiene múltiples motivaciones que no podemos detallar adecuadamente acá. Solo mencionaremos dos hechos importantes relacionados con su frecuente elección:

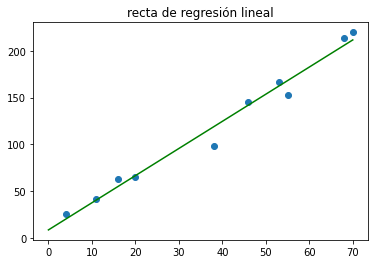
* Por un lado, minimizar el error cuadrático medio puede resolverse derivando la fórmula del error. Los que sepan algo de análisis matemático, recordarán que la derivada nos permite encontrar mínimos y que la derivada de una función cuadrática es una función lineal. Por lo tanto, encontrar la recta que mejor ajusta los datos se reduce a buscar el cero de una derivada que en el fondo se reduce a resolver un sistema lineal, algo que sabemos hacer muy bien y muy rápido. Si en lugar de minimizar la suma de los cuadrados de los residuos planteáramos, por ejemplo, minimizar la suma de los valores absolutos de los residuos no podríamos encontrar la recta que mejor ajusta tan fácilmente.
* Otro argumento muy fuerte, de naturaleza estadística en este caso, es que si uno considera que los residuos son por ejemplo errores de medición y que tienen una distribución normal (una gaussiana), entonces puede mostrarse que la recta que da el método de los cuadrados mínimos es *la recta de máxima verosimilitud*.

Estas cosas se explican muy bien en [el apunte de Andrew Ng](http://cs229.stanford.edu/notes2020spring/cs229-notes1.pdf) que citamos antes.

Recordá que en la [Sección 5.2](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/05_Random_Plt_Dbg/02_NumPy_Arrays.md#f%C3%B3rmulas-matem%C3%A1ticas) vimos que calcular el promedio de estos errores cuadráticos es muy sencillo en numpy. También podés usar la función mean\_squared\_error del módulo sklearn.metrics que trae muchas métricas muy útiles.

**Ejemplo: el modelo de cuadrados mínimos**

Para los datos que graficamos antes, ésta es *la mejor recta*, es decir, la que minimiza la suma de los cuadrados de los residuos. Vamos a decir que esta recta es **el ajuste lineal de los datos**.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/ej0_ajuste.png)

¿Cómo se encuentran estos coeficientes?

**Ajuste del modelo de cuadrados mínimos**

Como buscamos el mínimo de la expresión

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/eq_suma_de_cuadrados.png)

podemos derivar respecto de los parámetros `a, b` e igualar a cero para despejarlos. No es una cuenta díficil. El único cero que va a tener la derivada se corresponde con un mínimo (porque la recta se puede ajustar \*tan mal como uno quiera\*). De esta manera se obtienen las siguientes fórmulas para el ajuste:

def ajuste\_lineal\_simple(x,y):

a = sum(((x - x.mean())\*(y-y.mean()))) / sum(((x-x.mean())\*\*2))

b = y.mean() - a\*x.mean()

return a, b

**Ejemplo: datos sintéticos**

Veamos un ejemplo generado con datos sintéticos. Generamos 50 datos para la variable x, y determinamos a la variable y con una relación lineal más un error normal.

import numpy as np

N = 50

minx = 0

maxx = 500

x = np.random.uniform(minx, maxx, N)

r = np.random.normal(0, 25, N) # residuos simulados

y = 1.3\*x + r

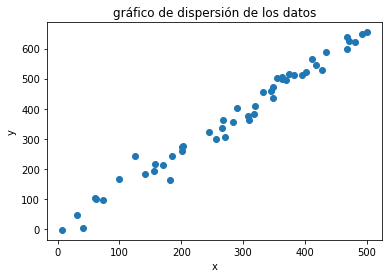
g = plt.scatter(x = x, y = y)

plt.title('gráfico de dispersión de los datos')

plt.xlabel('x')

plt.ylabel('y')

plt.show()

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/ejsint_scatter.png)

Ahora ajustamos con las fórmulas que vimos antes:

a, b = ajuste\_lineal\_simple(x, y)

grilla\_x = np.linspace(start = minx, stop = maxx, num = 1000)

grilla\_y = grilla\_x\*a + b

g = plt.scatter(x = x, y = y)

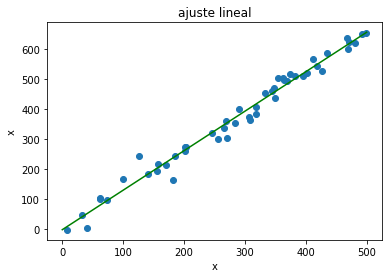
plt.title('y ajuste lineal')

plt.plot(grilla\_x, grilla\_y, c = 'green')

plt.xlabel('x')

plt.ylabel('y')

plt.show()

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/ejsint_ajuste.png)

**Ejercicio 11.14: precio\_alquiler ~ superficie**

Consideramos datos de precios (en miles de pesos) de alquiler mensual de departamentos en el barrio de Caballito, CABA, y sus superficies (en metros cuadrados). Queremos modelar el precio de alquiler a partir de la superficie para este barrio. A veces este modelo se nota con *precio\_alquiler ~ superficie*.

* Usando la función que definimos antes, ajustá los datos con una recta.
* Graficá los datos junto con la recta del ajuste.

superficie = np.array([150.0, 120.0, 170.0, 80.0])

alquiler = np.array([35.0, 29.6, 37.4, 21.0])

Una forma de cuantificar cuán bien ajusta la recta es considerar el promedio de los errores cuadráticos, llamado *error cuadrático medio*.

errores = alquiler - (a\*superficie + b)

print(errores)

print("ECM:", (errores\*\*2).mean())

* Calculá el error cuadrático medio del ajuste que hiciste recién.

Guardá tu código en el archivo alquiler.py para entregar.

**Ejemplo: relación cuadrática**

Veamos qué pasa si los datos guardan en realidad una relación cuadrática. Generemos aletoriamente variables independientes y dependientes con este tipo de relación.

np.random.seed(3141) # semilla para fijar la aleatoriedad

N=50

indep\_vars = np.random.uniform(size = N, low = 0, high = 10)

r = np.random.normal(size = N, loc = 0.0, scale = 8.0) # residuos

dep\_vars = 2 + 3\*indep\_vars + 2\*indep\_vars\*\*2 + r # relación cuadrática

Grafiquemos los datos obtenidos y, por comodidad, llamémoslos x e y.

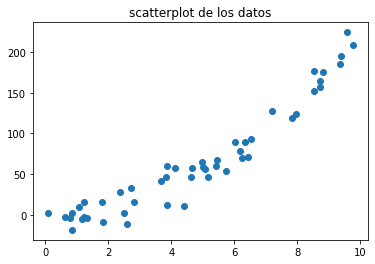
x = indep\_vars

y = dep\_vars

plt.scatter(x,y)

plt.title('scatterplot de los datos')

plt.show()

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/ejcuad_scatter.png)

Y ajustemos un modelo lineal (notado: *y ~ x*) a estos datos.

a, b = ajuste\_lineal\_simple(x, y)

grilla\_x = np.linspace(start = 0, stop = 10, num = 1000)

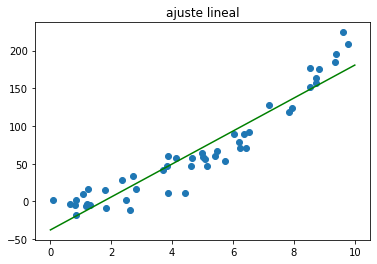
grilla\_y = grilla\_x\*a + b

g = plt.scatter(x = x , y = y)

plt.title('ajuste lineal')

plt.plot(grilla\_x, grilla\_y, c = 'green')

plt.show()

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/ejcuad_ajustelineal.png)

Veamos cuánto vale el error cuadrático medio.

errores = y - (x\*a + b)

print("ECM", (errores\*\*2).mean())

**Parte optativa:**

Ahora vamos a profundizar en algunos conceptos y a ver maneras alternativas de hacer las cosas. Lo que sigue es optativo.

**Ejemplo: precómputo de atributos adecuados**

Es natural pensar que aproximar una parábola con un modelo lineal no es lo más sensato. Un modelo alternativo es usar como variable explicativa x^2 en vez de x. El cómputo de xc = x^2 se realiza en un paso previo de forma que el modelo sigue siendo lineal (ahora lineal en x^2). Esto significa que el formalismo matemático para encontrar los coeficientes del nuevo modelo es el mismo que antes.

xc = x\*\*2

ap, bp = ajuste\_lineal\_simple(xc, y)

grilla\_y\_p = (grilla\_x\*\*2)\*ap + bp

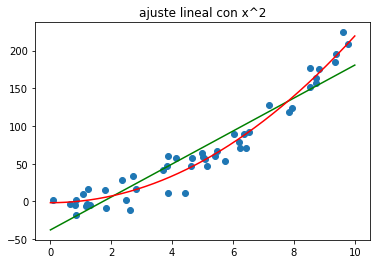
plt.scatter(x,y)

plt.plot(grilla\_x, grilla\_y, c = 'green')

plt.plot(grilla\_x, grilla\_y\_p, c = 'red')

plt.title('ajuste lineal con x^2')

plt.show()

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/ejcuad_ajusteconx2.png)

Y si queremos cuantificar el error en este modelo:

yhat = (x\*\*2)\*ap + bp # valores estimados

residuos = y - yhat # diferencia entre el valor original y el estimado

ecm = (residuos\*\*2).mean() # error cuadrático medio

print("ECM:", ecm)

Al usar x^2 en lugar de x mejora sustancialmente la bondad de ajuste del modelo (notado: *y ~ x^2*). Veremos próximamente que podemos usar ambas x y x^2 como variables explicativas y obtener un ajuste aún mejor de los datos.

**Raíz del error cuadrático medio**: Una alternativa al error cuadrático medio es su raíz cuadrada, conocida como *root mean squared error* (RMSE). La ventaja de esta medida de la bondad de ajuste de un modelo a los datos radica en que ésta se expresa en las misma unidades que la variable a explicar, y, mientras que el ECM (MSE) se expresa en *unidades al cuadrado*. Siendo la raíz una función monótona, minimizar una métrica o la otra es equivalente.

**Scikit-Learn**

La biblioteca [scikit-learn](https://scikit-learn.org/stable/) tiene herramientas muy útiles para el análisis de datos y el desarrollo de modelos de aprendizaje automático, aunque se mantiene relativamente alejada de la inferencia estadística. En particular, para regresión lineal tiene el módulo linear\_model, y en el siguiente ejemplo mostramos cómo puede usarse. Para les que estén habituades al lenguaje R, quizás les conviene usar la biblioteca [stastmodels](https://www.statsmodels.org/stable/regression.html) que tiene un funcionamiento más cercano.

Al igual que el modelo de clustering que usamos en el [Ejercicio 9.19](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/06_Teledeteccion.md#ejercicio-919-clasificaci%C3%B3n-autom%C3%A1tica) de teledetección, el objeto de tipo LinearRegression de sklearn.liearmodel también tiene un método fit() que permite ajustar el modelo a los datos y otro predict() que permite usar el modelo ajustado con nuevos datos.

Acá rehacemos el primer ejemplo que dimos ([Sección 11.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/05_Regresion_Lineal.md#ejemplo-el-modelo-de-cuadrados-m%C3%ADnimos)), usando pandas y el módulo linear\_model.

import pandas as pd

from sklearn import linear\_model

x = np.array([55.0, 38, 68, 70, 53, 46, 11, 16, 20, 4]) # mismos datos x, y

y = np.array([153.0, 98, 214, 220, 167, 145, 41, 63, 65, 25])

datosxy = pd.DataFrame({'x': x, 'y': y}) # paso los datos a un dataframe

ajus = linear\_model.LinearRegression() # llamo al modelo de regresión lineal

ajus.fit(datosxy[['x']], datosxy['y']) # ajusto el modelo

grilla\_x = np.linspace(start = 0, stop = 70, num = 1000)

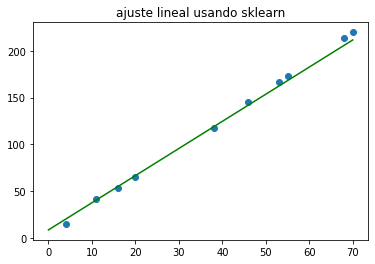
grilla\_y = ajus.predict(grilla\_x.reshape(-1,1))

datosxy.plot.scatter('x','y')

plt.title('ajuste lineal usando sklearn')

plt.plot(grilla\_x, grilla\_y, c = 'green')

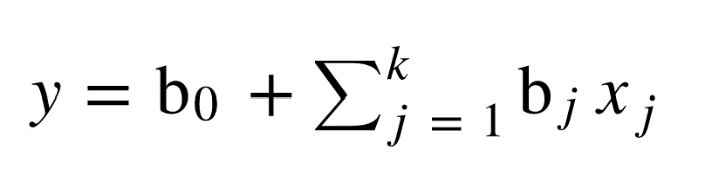
plt.show()

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/ejsk1_ajuste.png)

Usamos el método fit() para ajustar el modelo y el método predict() para obtener los valores de y de la recta. Fijate que al método fit le pasamos el Dataframe datosxy[['x']] y no la serie datosxy['x'] ya que el método está preparado para trabajar con regresiones múltiples (es decir, tenés muchos regresores).

**Regresión Lineal Múltiple**

La regresión lineal múltiple tiene un planteo similar, pero con más variables explicativas. El modelo es el siguiente.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/eq_reg_lin_multiple.png)

**Ejemplo: superficie y antigüedad**

Trabajamos nuevamente con los departamentos, ahora también conociendo su antigüedad, y la tomamos como otra variable explicativa. Ajustaremos un modelo que tenga en cuenta ambas variables, y lo notaremos: *precio\_alquiler ~ superficie + antigüedad*

superficie = np.array([150.0, 120.0, 170.0, 80.0])

alquiler = np.array([35.0, 29.6, 37.4, 21.0])

antigüedad = [50.0, 5.0, 25.0, 70.0]

data\_deptos = pd.DataFrame({'alquiler': alquiler, 'superficie': superficie, 'antigüedad': antigüedad})

X = pd.concat([data\_deptos.superficie,data\_deptos.antigüedad], axis = 1)

ajuste\_deptos = linear\_model.LinearRegression()

ajuste\_deptos.fit(X,data\_deptos.alquiler)

errores = data\_deptos.alquiler - (ajuste\_deptos.predict(X))

print(errores)

print("ECM:", (errores\*\*2).mean()) # error cuadrático medio

Usando los atributos intercept\_ y coef\_ de ajuste\_deptos escribí a mano la fórmula de la regresión múltiple obtenida y respondé las siguientes preguntas respecto al modelo obtenido:

* A mayor superficie, ¿aumenta o disminuye el precio?
* A mayor antigüedad, ¿aumenta o disminuye el precio?
* ¿Cuánto vale la ordenada al origen del modelo?

**Ejercicio 11.15: Peso específico**

Queremos estimar el peso específico de un metal (es decir, peso divido volumen, en unidades de g/cm³). Para esto, disponemos de barras de dicho metal, con base de 1cm² y largos diversos, y de una balanza que tiene pequeños errores de medición (desconocidos). Vamos a estimar el peso específico *R* del metal de la siguiente manera:

Sabemos que el volumen de una barra de largo m es mcm³, por lo que su peso debería ser R\*m. Queremos estimar R. Utilizando la balanza, tendremos los pesos aproximados de distintas barras, con ciertos errores de medición. Si ajustamos un modelo lineal a los datos de volumen y peso aproximado vamos a tener una buena aproximación para R (la pendiente de la recta).

Los datos de longitudes y pesos se encuentran en el archivo disponible acá.

* Cargá los datos directamente con el enlace usando el siguiente código.

import requests

import io

enlace = 'https://raw.githubusercontent.com/python-unsam/Programacion\_en\_Python\_UNSAM/master/Notas/11\_Recursion/longitudes\_y\_pesos.csv'

r = requests.get(enlace).content

data\_lyp = pd.read\_csv(io.StringIO(r.decode('utf-8')))

* Hacé una regresión lineal simple con sklearn, con variable explicativa longitud y variable explicada peso (*peso ~ longitud*).
* Estimá el peso específico del metal mirando el coeficiente obtenido.
* Graficá los datos junto con la recta del ajuste, y calculá el error cuadrático medio.
* Guardá el código en un archivo peso\_especifico.py.

*Cuidado:* por cómo planteamos el problema, estamos ajustando una recta con ordenada al origen igual a cero. Para esto tendrás que usar el parámetro fit\_intercept = False en la declaración de tu modelo.

**Ejercicio 11.16: Modelo cuadrático**

Volvamos ahora al ejemplo cuadrático de antes. La relación entre x (indep\_vars) e y (dep\_vars) estaba dada por y = 2 + 3\*x + 2\*x\*\*2 + r. Ya tratamos de ajustar regresiones simples tipo y = a\*x + b y y = a\*x^2 + b. Ajustemos ahora una regresión lineal múltiple, usando como regresores a x y a x^2.

Nos gustaría no generar datos aleatorios nuevamente sino usar los anteriores, ya generados, para poder comparar (los errores cuadráticos medios de) los tres modelos.

x = indep\_vars

xc = x\*\*2

y = dep\_vars

Para preparar los datos a usar como regresores (en este caso múltiple serán x y x^2) podés usar:

X = np.concatenate((x.reshape(-1,1),xc.reshape(-1,1)),axis=1)

Si te fijás, el array X tiene un shape de (50, 2). Esto se corresponde a cincuenta datos con dos atributos.

* Usá un objeto lm = linear\_model.LinearRegression() para comparar los ajustes obtenidos usando x como única variable regresora, xc (los cuadrados) como única variable regresora, o ambas en un modelo múltiple (notado: *y ~ x + x^2*). Imprimí para cada uno de los tres modelos, el error cuadrático medio y los coeficientes (ordenada al orígen y coeficientes de los regresores) obtenidos. ¿Qué modelo ajusta mejor? ¿Cuál da coeficientes más similares a los originales? ¿Qué pasaría si usáramos un modelo de grado tres o cuatro?
* Graficá los datos originales y los tres ajustes en un solo gráfico indicando adecuadamente los nombres de los modelos.

**Navaja de Ockham**

Al agregar covariables (regresores) a un modelo, el ajuste tiende a mejorar. Si ajusto un modelo con variables x1, x2, x3 para explicar una variable y no puedo obtener un peor ajuste que si lo ajusto usando solo las variables x1 y x2 ya que todo modelo con las dos variables es un caso particular del modelo con las tres (simplemente hay que poner el coeficiente de la tercera variable igual a cero). Por eso, en general, al agregar variables a un modelo, su error cuadrático disminuye. Sin embargo un modelo con mejor ajuste no es *necesariamente* mejor.

El principio metodológico conocido como la [navaja de Ockham](https://es.wikipedia.org/wiki/Navaja_de_Ockham) nos indica que de un conjunto de variables explicativas debe seleccionarse la combinación más reducida y simple posible.

Esto ayuda a evitar fenómenos como el **sobreajuste** que causa [problemas muy serios y a veces graciosos](https://twitter.com/electricfuture5/status/1309688641157906433).

**Ejercicio 11.17: Modelos polinomiales para una relación cuadrática**

Vimos en el [Ejercicio 11.16](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/05_Regresion_Lineal.md#ejercicio-1116-modelo-cuadr%C3%A1tico) que los datos de ese ejercicio se ajustan mejor con una regresión múltiple (usando x y x^2) que una regresión simple (basada un una sola variable). Te proponemos ahora que te fijes qué ocurre si seguimos aumentando el grado de las potencias de x que admitimos en la regresión múltiple (es decir, usar x, x^2,..., etc. hasta x^n). ¿Sigue bajando el error cuadrático medio? ¿Pueden considerarse *mejores* los modelos obtenidos?

Para n entre 1 y 8 realizá un ajuste con un polinomio de grado n (que tiene n+1 parámetros, por la ordenada al orígen) e imprimí una salida como esta:

-------------------------

Grado del polinomio: 1

Cantidad de parámetros: 2

ECM: 201.194

------------------------

Grado del polinomio: 2

Cantidad de parámetros: 3

ECM: 36.325

...

...

Te recomendamos usar la siguiente función pot() para generar las primeras potencias de x:

def pot(x,n):

X=x.reshape(-1,1)

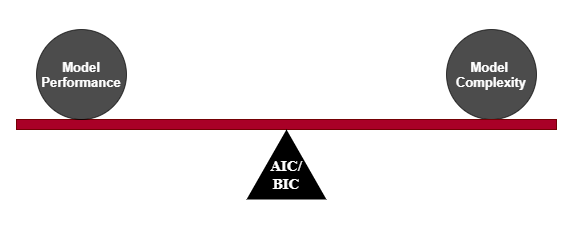
for i in range(n-1):

X=np.concatenate((X,(x\*\*(i+2)).reshape(-1,1)),axis=1)

return X

**Ejercicio 11.18: selección de modelos**

El [criterio de información de Akaike](https://es.wikipedia.org/wiki/Criterio_de_informaci%C3%B3n_de_Akaike) es una medida de la calidad relativa de un modelo estadístico, para un conjunto dado de datos. Como tal, el AIC proporciona un medio para la comparación de modelos. AIC maneja un trade-off entre la bondad de ajuste del modelo y la complejidad del mismo (medido en cantidad de parámetros).

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/AIC.png)

En el caso de la regresión lineal múltiple, puede computarse con la siguiente función:

def AIC(k, ecm, num\_params):

'''Calcula el AIC de una regresión lineal múltiple de 'num\_params' parámetros, ajustada sobre una muestra de 'k' elementos, y que da lugar a un error cuadrático medio 'ecm'.'''

aic = k \* np.log(ecm) + 2 \* num\_params

return aic

Agregá al código del ejercicio anterior el cómputo del AIC para cada modelo.

-------------------------

Grado del polinomio: 1

Cantidad de parámetros: 2

ECM: 201.194

AIC: 269.213

------------------------

Grado del polinomio: 2

Cantidad de parámetros: 3

ECM: 36.325

AIC: 185.626

...

...

Cuando complejizamos el modelo mejorando el error cuadrático medio, pero sin disminuir el AIC, es probable que el modelo se esté [sobreajustando](https://es.wikipedia.org/wiki/Sobreajuste) a los datos de entrenamiento.

Si seleccionamos el modelo ya no por su bondad de ajuste (ECM) sino buscando el mínimo AIC ¿Qué modelo queda seleccionado? Respodé esta pregunta usando el comando [np.argmin()](https://numpy.org/doc/stable/reference/generated/numpy.argmin.html) para encontrar el grado del polinomio que minimiza el AIC y comentá adecuadamente tu código. Guardalo en el archivo selección\_modelos.py para entregar.

**Ejercicio 11.19: Datos para la evaluación**

Otra alternativa para comparar modelos es evaluarlos en un conjunto de datos diferente al que usamos para entrenarlos. La próxima clase vamos a ver que sklearn tiene funciones que permiten partir automáticamente los datos en conjuntos de *entrenamiento* y *evalaución*. Por ahora supongamos que nos dan los siguientes datos frescos:

N=50

#genero datos para evaluar

x\_test = np.random.uniform(size = N, low = 0, high = 10)

r\_test = np.random.normal(size = N, loc = 0.0, scale = 8.0) # residuos

y\_test = 2 + 3\*x\_test + 2\*x\_test\*\*2 + r\_test # misma relación cuadrática

Evaluá los modelos que armaste antes usando el ECM sobre estos datos frescos. ¿Qué modelo da un mejor ajuste?

Estas técnicas de selección de modelos usando datos de entrenamiento y evaluación separados o usando el criterio de información de Akaike tratan de evitar usar modelos que se sobreajusten a los datos de entrenamiento. Este fenómeno, conocido como *overfitting*, puede causar problemas muy serios.

**Ejercicio 11.20: Altura y diámetro de árboles.**

Queremos comparar las formas de las siguientes especies de árboles en los parques de Buenos Aires:

* Jacarandá,
* Palo borracho rosado,
* Eucalipto, y
* Ceibo.

Vamos a trabajar nuevamente con el archivo de arbolado porteño en parques que tenés en el archivo '../Data/arbolado-en-espacios-verdes.csv'.

* Cargá los datos en un DataFrame data\_arbolado\_parques.
* Para cada especie, seleccioná los datos correspondientes, realizá un ajuste lineal (sin ordenada al orígen) de la altura dependiendo del diámetro a la altura del pecho. Realizá un scatterplot de los datos de la especie junto con la recta de regresión lineal.
* Realizá un gráfico comparando los cuatro modelos obtenidos.
* Guardá el código de este ejercicio en un archivo ajuste\_arboles.py.

*Observación*: Como podés ver en los scatterplots, para árboles más anchos hay mayor variabilidad de alturas que para árboles angostos. Esto implica que el modelo va a ser más sensible a datos de árboles anchos que a datos de árboles angostos. Esta caraceterística se llama *heterocedasticidad* y muchas veces es un problema para usar regresiones lineales. Por ejemplo, no es posible aplicar directamente tests de hipótesis a los resultados obtenidos.

Para explicarlo con un ejemplo: imaginá que tenemos tres pares de datos (*DAP* y *altura* para un árbol angosto, para un árbol mediano y para un árbol muy ancho) y supongamos que hay una relación lineal *real* que es la que estamos buscando estimar a partir de los datos. La altura del arbol angosto va a variar unos pocos centímetros respecto a este modelo ideal mientras que la altura del árbol ancho puede variar muchos metros. Esto hace que el *residuo* del árbol grueso respecto al modelo ideal sea mucho mayor que el residuo del árbol angosto y, por lo tanto, que su infuencia en los coeficientes estimados sea mayor también (recordemos que estamos minimizando la suma de estos residuos al cuadrado). Esto viola una de las hipótesis de la regresión lineal (la *homocedasticidad*) que dice que todos los residuos tienen la *misma* distribución.

En este caso contamos con una gran cantidad de datos y podemos aplicar de todas formas la regresión en el marco de un análisis exploratorio de los datos.

**Ejercicio 11.21: Gráficos de ajuste lineal con Seaborn**

* Seleccioná los datos correspondientes a las especies: Jacarandá, Palo borracho rosado, Eucalipto y Ceibo, todas en un mismo DataFrame, usando el siguiente filtro.

filtro = data\_arbolado\_parques['nombre\_com'].isin(esp\_selec)

* Explorá el comando de seaborn sns.regplot(), que ajusta el modelo lineal y lo grafica sin pasar por scikit learn. El parámetro order te permite hacer ajustes polinomiales. El ci se refiere al intervalo de confianza a sombrear.
* Para facilitar la comparación que hiciste en el ejercicio anterior, graficá todos los ajustes juntos usando:

g = sns.FacetGrid(datos\_selec\_p, col = 'nombre\_com')

g.map(sns.regplot, 'diametro', 'altura\_tot')

*Observación*: Nos quedaron afuera de esta clase temas importantes como sobreajuste (*Overfitting*), partición de los datos en conjuntos de entrenamiento y evaluación, validación cruzada, presencia de datos atípicos (outliers), tests de hipótesis, selección de modelos... No era nuestra idea dar estos contenidos sino mostrar un acercamiento práctico desde Python al problema de la regresión lineal.

**11.6 Cierre de la clase de Recursión y Regresión**

En esta clase vimos el concepto de recursión y lo ejercitaste. También estudiamos un poquito (o no tan poco) sobre el ajuste y uso de regresiones lineales en Python. Ya estamos cerrando este curso. La próxima clase será la última. En la clase de hoy te pedimos que entregues los siguientes archivos:

* El archivo larenga.py del [Ejercicio 11.9](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/05_EjerciciosRec.md#ejercicio-119-pascal).
* El archivo bbin\_rec.py del [Ejercicio 11.11](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/05_EjerciciosRec.md#ejercicio-1111-b%C3%BAsqueda-binaria).
* El archivo hojas\_ISO.py del [Ejercicio 11.13](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/05_EjerciciosRec.md#ejercicio-1113-hojas-iso-y-recursi%C3%B3n).
* El archivo alquiler.py del [Ejercicio 11.14](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/05_Regresion_Lineal.md#ejercicio-1114-precioalquiler-superficie).
* Opcional: El archivo seleccion\_modelos.py del [Ejercicio 11.18](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/05_Regresion_Lineal.md#ejercicio-1118-selecci%C3%B3n-de-modelos).

Como de costumbre, completá por favor [el formulario](https://docs.google.com/forms/d/1Ttqq8axr8wSFLcydLEO9YKzz3IvN0jdNTka-5Onm5lI) asociado a la clase y adjuntá los archivos correspondientes. Ésta es la última semana en la que podés participar de la corrección de pares con el [Ejercicio 11.13](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/11_Recursion/05_EjerciciosRec.md#ejercicio-1113-hojas-iso-y-recursi%C3%B3n).