

# `python` para el pueblo

Federico Bosio      Santiago Curutchet

19 de septiembre de 2020

En el año 2018, decidimos dictar un curso extracurricular de programación orientado al procesamiento de señales, en vista de la necesidad que los mismos estudiantes de *Ingeniería Acústica*, alias *Ingeniería de Sonido*, han manifestado en diversas ocasiones. La construcción del curso, su temario y la organización del mismo derivaron en la confección de apuntes que son útiles para cualquiera con conocimientos elementales de programación en búsqueda de aplicarlos a proyectos concretos, relacionados con la Ingeniería en general.

Con el tiempo, los apuntes han sido extendidos a punto tal de tornar en libro. “**Python para el pueblo**” es un libro accesible sobre Procesamiento de Señales, Análisis Numérico y desarrollo complementario de interfaces gráficas mediante el lenguaje de programación Python. Este lenguaje es de código abierto, sigue la filosofía del software libre, es relativamente sencillo de utilizar, y tiene uso masivo.

La obra se trata de literatura sobre programación, a nuestro parecer, sin precedentes, debido a la especificidad de los temas abordados, y el enfoque con el que dichos temas fueron abordados. La obra incentiva y propone hacer. Da las herramientas suficientes para lograrlo. Por ello, las explicaciones van al grano, no se detienen en detalles teóricos profundos, y van acompañadas de ejemplos e ilustraciones. Con el estilo lacónico de la obra no se pretende bajo ningún concepto horrorizar a los puristas. Todo está como debe ser, con la longitud justa y necesaria. No falta nada. El objetivo no es cavilar ni elucubrar, sino dar herramientas con funciones concretas. Es un libro para estudiantes de Ingeniería, investigadores y curiosos en general, con cierta experiencia en programación.

Esperamos honestamente que agrade a los lectores.

# Índice general

<b>1. El lenguaje python</b>	<b>6</b>
1.1. El intérprete . . . . .	7
1.2. Expresiones y variables . . . . .	8
1.3. El editor de código . . . . .	11
1.4. Estilo . . . . .	12
1.5. Programa ramificado . . . . .	13
1.6. Ciclos iterativos e ingreso de datos . . . . .	15
1.7. Conversión y obtención de tipos . . . . .	17
1.8. Variables no escalares . . . . .	18
1.8.1. Cadenas de caracteres ( <code>str</code> ) . . . . .	18
1.8.2. Listas ( <code>list</code> ) . . . . .	20
1.9. El ciclo <code>for</code> . . . . .	24
1.10. Funciones . . . . .	24
1.10.1. Argumentos obligatorios y opcionales . . . . .	26
1.10.2. Múltiples valores de salida . . . . .	27
1.10.3. Recursión . . . . .	29
1.10.4. Funciones de alto orden . . . . .	31
1.11. Módulos de <code>python</code> . . . . .	31
1.11.1. Biblioteca estándar . . . . .	33
1.12. Comentario final . . . . .	33
<b>2. Procesamiento de audio</b>	<b>34</b>
2.1. El audio es vectorial . . . . .	35
2.2. ¿Qué es <code>numpy</code> ? . . . . .	35
2.2.1. Rutinas de creación de vectores . . . . .	36
2.2.2. Operaciones vectoriales . . . . .	37
2.2.3. Audio en estéreo . . . . .	39
2.2.4. Rutinas de creación de matrices . . . . .	40
2.2.5. Concatenación de matrices . . . . .	41
2.2.6. Operaciones matriciales . . . . .	41
2.2.7. Expansión de vectores y matrices . . . . .	42

2.3. Reproducción de audio . . . . .	45
2.4. Grabación de audio . . . . .	45
2.5. Efectos de audio en tiempo . . . . .	46
2.5.1. Eco . . . . .	46
2.5.2. Filtro peine . . . . .	47
2.5.3. Tremolo . . . . .	49
2.5.4. Distorsión armónica . . . . .	51
2.5.5. Reverberación . . . . .	52
<b>3. Gráficos y operaciones numéricas</b>	<b>54</b>
3.1. Interpolaciones y ajustes . . . . .	58
3.1.1. Polinomios de Lagrange . . . . .	58
3.1.2. Ajuste por cuadrados mínimos . . . . .	61
3.1.3. Ajuste de funciones potenciales . . . . .	64
3.2. Suavizado de datos . . . . .	65
3.2.1. Filtro de media móvil . . . . .	65
3.2.2. Filtro de Savitzky-Golay . . . . .	68
3.3. Área y área acumulada . . . . .	69
3.3.1. Cálculo de áreas . . . . .	69
3.3.2. Área acumulada . . . . .	70
3.4. Resolución de ecuaciones diferenciales . . . . .	73
3.4.1. La idea teórica: método de Euler . . . . .	73
3.4.2. Ecuaciones diferenciales en Python 3 . . . . .	74
<b>4. La transformada de Fourier</b>	<b>79</b>
4.1. El análisis de espectro . . . . .	79
4.2. La Transformada (discreta) de Fourier . . . . .	83
4.3. Filtros, Transformada Z y ecualización . . . . .	86
4.4. Filtros estándar . . . . .	87
<b>5. Orientación a objetos</b>	<b>94</b>
5.1. Métodos . . . . .	94
5.2. Clases . . . . .	95
5.2.1. Constructor, atributos y métodos . . . . .	95
5.3. Analogía industrial . . . . .	96
5.4. Herencia . . . . .	97
5.4.1. Sobrescritura . . . . .	98
5.4.2. Extensión . . . . .	99
5.5. Comentario final . . . . .	101

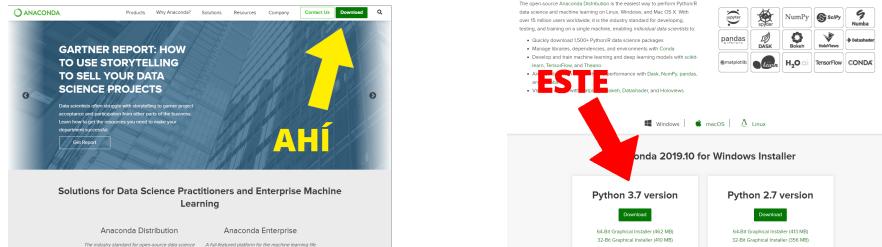
<b>ÍNDICE GENERAL</b>	<b>5</b>
<b>6. Interfaz gráfica de usuario</b>	<b>102</b>
6.1. La ventana . . . . .	102
6.2. Componentes . . . . .	103
6.3. Información compartida . . . . .	105
6.4. Componente personalizado . . . . .	108
6.5. Incrustación de gráficos . . . . .	113
6.5.1. Un gráfico interactivo . . . . .	114
6.6. Diseño visual . . . . .	116
6.6.1. Generador de tonos puros . . . . .	119
6.6.2. Conexión de señales entre componentes . . . . .	122
6.6.3. Lógica del generador . . . . .	122
6.7. Comentario final . . . . .	124
<b>7. Digitalización y audio en tiempo real</b>	<b>125</b>
7.1. El proceso de digitalización . . . . .	126
7.1.1. Muestreo . . . . .	126
7.1.2. Cuantificación . . . . .	127
7.1.3. Codificación . . . . .	128
7.2. Un ejemplo en Python . . . . .	129
7.3. PyAudio, <i>buffers</i> y <i>streams</i> . . . . .	133
7.4. Un analizador de espectro en tiempo real . . . . .	135
7.5. Modo <i>Callback</i> . . . . .	141
<b>A. Computadoras y algoritmos</b>	<b>144</b>
A.1. ¿Qué es una computadora? . . . . .	145
A.2. ¿Qué instrucciones recibe? . . . . .	146
A.3. ¿Cómo se dan las instrucciones? . . . . .	148

# Capítulo 1

## El lenguaje python

Preparamos nuestro entorno de trabajo antes de lanzarnos a programar. En este libro, utilizaremos **anaconda**, un conjunto de programas en donde viene todo incluido y funcionando: el lenguaje **python**, el editor **spyder**, el gestor de paquetes **conda** y muchísimas otras cosas más, que probablemente ni usemos, pero son *cool*. Veremos brevemente cómo instalar **anaconda**.

1. Abrimos <https://www.anaconda.com/> en un navegador web.
2. Pulsamos **Download** y elegimos la última versión (para el momento en el que se hizo este libro, es la 2019.10, con **python 3.7**).

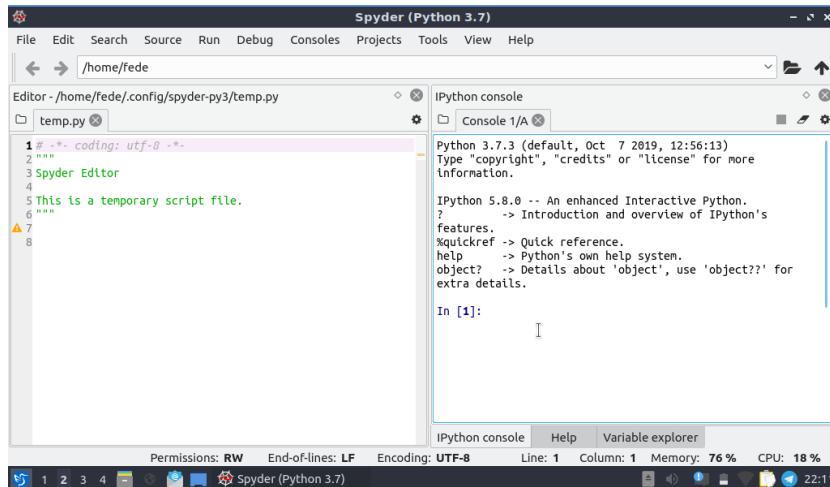


3. Ejecutamos el instalador descargado. **Importante:** Es conveniente instalar **anaconda** sólo para nuestro usuario. Debemos estar atentos para seleccionar esta opción en Windows y Mac OS.
4. El resto de las opciones puede dejarse por defecto.
5. Avanzamos hasta comenzar la instalación.
6. Esperamos un rato y listo.

Para los detalles, ver <https://docs.anaconda.com/anaconda/>.

## 1.1. El intérprete

Abrimos **spyder** y hacemos clic dentro de la Terminal (o presionamos la combinación de teclas **Ctrl+Shift+I**).



La Terminal contiene el intérprete de python.

Escribimos la frase Hola Python y pulsamos la tecla Enter.

```
In [1]: Hola Python
SyntaxError: invalid syntax
```

**El intérprete no está hecho para comprender el lenguaje natural**, por eso el error. Si en cambio encerramos la palabra entre comillas simples:

```
In [2]: 'Hola Python'
Out[2]: 'Hola Python'
```

el intérprete responde. Esto significa que **python admite palabras encerradas por comillas**. Las palabras entre comillas se llaman *cadenas* y se abrevian **str** por la palabra original en inglés (“string”). Se llaman así porque son *cadenas de caracteres* [1], o sea, “letras encadenadas”.

Podemos ingresar también números y hacer cuentas:

```
In [3]: (5+9-4*2) / 3
Out[3]: 2.0
```

Las operaciones matemáticas tienen la precedencia esperada:

$$(5 + 9 - 4 * 2) / 3 \longrightarrow (5 + 9 - 8) / 3 \longrightarrow (6) / 3 \longrightarrow 2.0$$

**Python usa el punto como separador decimal.** Con esto indica que el resultado no es un número entero.

Otra operación posible es la potenciación, cuyo operador es `**`.

```
In [4]: (3**2*(5**2-1))**(1/3)
Out[4]: 5.999999999999999
```

El resultado no es exactamente 6 porque la fracción  $\frac{1}{3}$  es, en realidad, un número decimal periódico. Tiene infinitas cifras decimales, pero **la computadora solamente almacena una cantidad finita de dígitos** de esta fracción [2], y calcula la potencia con esa cantidad finita.

```
In [5]: 1/3
Out[5]: 0.3333333333333333
```

Además de las operaciones básicas, python puede calcular el cociente y el resto de una división de **números enteros**. Se hacen, respectivamente, con los operadores `//` y `%`.

```
In [6]: 5*(13//5) + 13%5
Out[6]: 13
```

El resultado es una verificación de la conocida regla que vemos en la primaria para chequear el cálculo de la división: dividendo = divisor \* cociente + resto.

Si dividendo y divisor no son enteros, el operador `//` trunca el resultado.

```
In [7]: 13.2 // 5.7
Out[7]: 2.0
In [8]: 13.2 / 5.7
Out[8]: 2.31578947368421
```

## 1.2. Expresiones y variables

Las cadenas, los números y los operadores matemáticos conforman **expresiones**. Para cada `In`, el intérprete evalúa la expresión dada y lleva el resultado a `Out`, siempre que no surjan errores en el proceso. Podemos dar nombres a las expresiones.

```
In [9]: resultado = 1 + 2
```

Ahora la expresión `1 + 2` se llama `resultado`. Al nombre de una expresión se la conoce como **variable** [1]. El valor que resulta de evaluar la expresión `1 + 2` se guarda en la variable llamada `resultado`. Podemos ver dicho valor en la Terminal. Para ello, escribimos la palabra `resultado`, y pulsamos la tecla `Enter`.

```
In [10]: resultado
Out[10]: 3
```

*Aclaración importante:* el signo = no es una igualdad matemática, sino una asignación. La variable va a la izquierda, y la expresión a la derecha. Debemos respetar este orden al asignar:

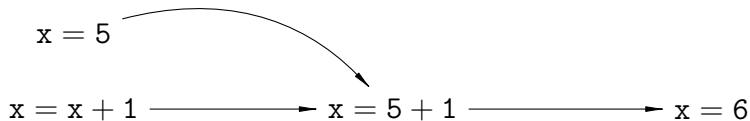
```
In [11]: 1 + 2 = resultado
SyntaxError: can't assign to operator
```

Es importante distinguir entre asignación e igualdad matemática. Consideremos, por ejemplo, la expresión  $x = x + 1$ . La igualdad es falsa, cualquiera sea el valor numérico de  $x$ . Pero acá el signo = es un operador de asignación:

```
In [12]: x = 5
In [13]: x = x + 1
In [14]: x
Out[14]: 6
```

¿Cómo es que funciona?

$x = 5$  Asigna el nombre  $x$  al número 5.  
 $x = x + 1$  Evalúa la expresión  $x + 1$  sustituyendo  $x$  por su valor.  
Asigna el nombre  $x$  al resultado obtenido.



Para el intérprete de python, la expresión  $x + 1$  es  $5 + 1$ , o sea, 6, porque la variable  $x$  era inicialmente 5. Por eso,  $x = x + 1$  resulta  $x = 6$ .

Evaluamos igualdades matemáticas utilizando dos signos igual:

```
In [15]: x == x + 1
Out[15]: False
```

Además de hacer cuentitas, python evalúa también *expresiones lógicas*. La palabra `False` indica que la igualdad no se cumple. Sin embargo, al evaluar la expresión lógica, el valor que resulta depende siempre de  $x$ .

```
In [16]: 2 * x == x + 1
Out[16]: False
In [17]: x = 1
In [18]: 2 * x == x + 1
Out[18]: True
```

El intérprete no resuelve las ecuaciones. Reemplaza el valor de `x` establecido en ambos miembros, evalúa las expresiones, y compara los resultados.

Las desigualdades se evalúan así:

```
In [19]: 3 < 4
Out[19]: True
In [20]: -2 > x
Out[20]: False
In [21]: 0 <= x
Out[21]: True
In [22]: 0 >= 0
Out[22]: True
In [23]: 9 != 8
Out[23]: True
```

Mientras `3 < 4` y `-2 > x` son equivalentes a su contraparte matemática,  $0 \leq x$  se representa `0 <= x`,  $0 \geq 0$  es `0 >= 0`, y  $9 \neq 8$  es `9 != 8`.

Operamos con expresiones lógicas usando `and`, `or` y `not`.

```
In [24]: True and True
Out[24]: True
In [25]: 1 < 2 and 3 > 4
Out[25]: False
In [26]: False or False
Out[26]: False
In [27]: x == 1 or False
Out[27]: True
In [28]: not True
Out[28]: False
```

Al asignar `True` o `False` a una variable, definimos una variable *booleana*, que se pronuncia “buleana” (si no les gusta como suena, le dicen “variable lógica” y listo). Por ejemplo, las asignaciones

```
In [29]: mi_variable1 = True
In [30]: mi_variable2 = -3 < -6 and x != 5
```

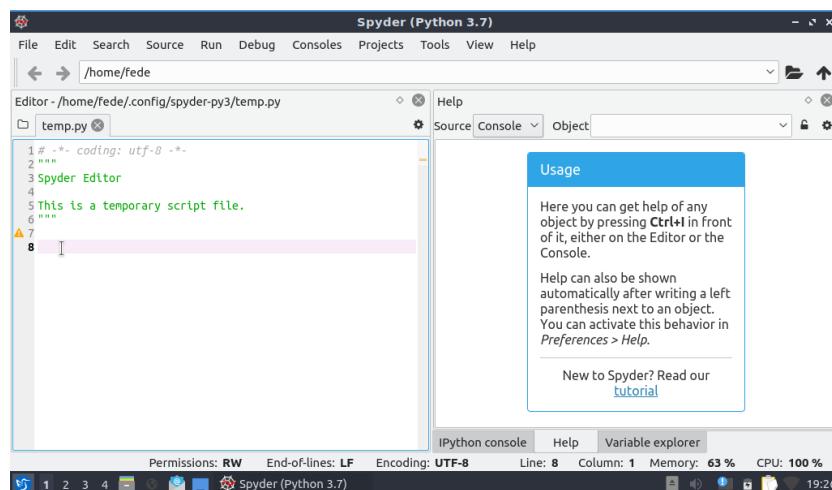
hacén que `mi_variable1` y `mi_variable2` sean booleanas.

Las variables numéricas y booleanas se llaman *escalares*. Son la mínima unidad del lenguaje, análogas a los átomos que conforman la materia. En python podemos definir también variables *no escalares*, como por ejemplo, las cadenas [1]. Siguiendo la analogía físicoquímica, las expresiones *no escalares* podrían ser, por ejemplo, moléculas y compuestos. Trataremos con estas expresiones en la sección 1.8, un poco más adelante.

### 1.3. El editor de código

La terminal interactiva de `spyder` es una buena herramienta para hacer pruebas rápidas. No obstante, cuando queremos evaluar expresiones en forma secuencial, a modo de instrucciones, trabajar de esta forma se vuelve inviable. Las instrucciones se almacenan entonces en un archivo, para ser ejecutadas después en cualquier momento por el intérprete [2]. El Editor de `spyder` nos permite elaborar y guardar un archivo de este tipo, que recibe varios nombres: “guión”, “script”, “código”, “programa”, etc. Todos los términos significan más o menos lo mismo, así que los intercambiaremos libremente a lo largo del libro.

A continuación, escribiremos una serie de instrucciones simples en el Editor para calcular el área de un rectángulo, a partir de un lado y su diagonal. Hacemos clic en el Editor de `spyder` (o sino, `Ctrl+Shift+E`).



Escribimos entonces en el Editor lo siguiente.

```
# Área de un rectángulo a partir de lado y diagonal
lado = 3
diagonal = 5
area = lado * (diagonal**2-lado**2)**(1/2)
area
```

Apretamos la tecla F5 para ejecutar el archivo (o sino desde el menú Run). Si no guardaron el archivo, aparecerá un cuadro de diálogo para hacerlo. Guárdenlo con el nombre que quieran. Si es la primera vez que ejecutan un código desde `spyder`, les aparecerá una ventana de configuración. Digan a todo eso que sí, debería funcionar con las opciones predeterminadas.

Aparentemente, al ejecutar el código, no pasa nada. Pero si vamos a la terminal (**Ctrl+Shift+I**) y vemos el valor de `area`...

```
In [31]: area
Out[31]: 12.0
```

...¡el código funciona efectivamente bien! Pero el intérprete no muestra el resultado. Mostramos el `area` durante la ejecución en la terminal con `print`:

```
# Área de un rectángulo a partir de lado y diagonal
lado = 3
diagonal = 5
area = lado * (diagonal**2-lado**2)**(1/2)
print(area) # <- cambió esta línea
```

Todo lo que sigue al símbolo `#` es ignorado por el intérprete: es lo que se llama *comentario*. Un programador incluye comentarios en un código para esclarecer su funcionamiento [2]. Además de los comentarios, hay otras formas de hacer el código más claro. Vemos algunas pautas a continuación.

## 1.4. Estilo

Con el siguiente código se pretende, también, calcular el área de un rectángulo a partir de un lado y su diagonal.

```
a= 3
b    = 5
print( a *(b** 2 -a)** (1/  2)  )
```

Los datos de entrada son iguales a los del código anterior pero, al ejecutar este código, la Terminal muestra un número distinto de 12.0.

Entonces, ¿cuál es el error? El error es haber escrito el código de forma tan inentendible en primer lugar.

El siguiente código es equivalente, pero mucho más piola.

```
# Área de un rectángulo a partir de lado y diagonal
lado = 3
diagonal = 5
area = lado * (diagonal**2-lado)**(1/2)
print(area)
```

Ustedes juzgarán si ahora resulta más sencillo encontrar el `**2` que falta (comparen con el código de la sección anterior).

**Es importante ser *claro* al elaborar un código:** las variables deben ser nombradas de acuerdo a lo que son, deben ponerse espacios alrededor del signo = al asignar variables, y los espacios deben señalar también las operaciones de menor precedencia. En el cálculo del **área**, por ejemplo, la operación de menor precedencia es la multiplicación.

Hay dos factores que motivan el orden y la prolijidad al programar.

1. El trabajo en equipo, en donde uno trabaja con código hecho por otras personas, y otros trabajan con los míos. Es más fácil comprendernos mutuamente si todos seguimos el mismo estilo de programación.
2. El código inentendible que hizo mi yo del pasado. Puede ser que hace un mes me creyera un genio, pero qué desordenado era.

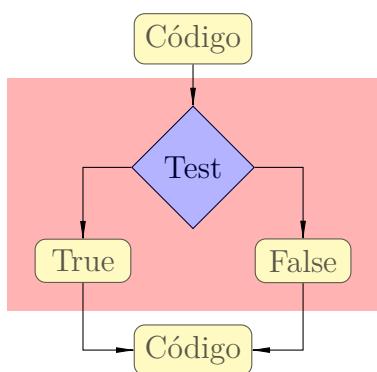
MORALEJA:

**Si vamos a programar, mejor hacerlo con estilo.**

Python tiene una *guía de estilo* llamada PEP 8, a la que pueden acceder desde una web dedicada: <https://pep8.org/>. Todos deberíamos leerla, aunque su longitud justifica el hecho de que rara vez la leamos completa.

## 1.5. Programa ramificado

Hasta el momento, escribimos **programas lineales**: absolutamente **todas las instrucciones son ejecutadas** en orden por el intérprete. La ejecución termina simplemente cuando se terminan las instrucciones. La cosa se pone interesante cuando **incorporamos la capacidad de la computadora para tomar decisiones**. Un programa en donde se aprovecha esta capacidad se denomina “branching program” [1], cosa que podríamos traducir como **programa ramificado**.



El dibujo es el diagrama de un programa ramificado, que tiene un **bloque condicional**. El primer Código se lee hasta llegar al Test. El Test tiene un valor booleano, que determina si se lee el código del bloque “True”, o el del bloque “False”. Una vez tomada la decisión, y terminada la ejecución del bloque seleccionado, el programa retoma su camino original, con el resto de Código que sigue.

En Python, implementamos el bloque condicional con `if` y `else`:

```
x = 3;

if x > 0:
    print('Es positivo.')
else:
    print('No es positivo.')

print('Este texto se muestra siempre.')
```

La ejecución del código deviene:

```
Es positivo.
Este texto se muestra siempre.
```

El intérprete *saltea* la línea de código `print('No es positivo')`, o sea, sólo ejecuta lo que está bajo el `if`, con sangría. *La sangría identifica en qué parte del bloque condicional estoy*, por ende, todas las que aparecen en el código son obligatorias. **Sin sangrías, el bloque condicional no anda.**

Si cambiamos `x` por cero o negativo, y pulsamos F5:

```
No es positivo.
Este texto se muestra siempre.
```

el intérprete se saltea la otra parte, y ejecuta lo que sigue al `else`.

Nuestro programa funciona, pero no diferencia números negativos del cero. Agreguemos un nuevo bloque condicional para que sí lo haga.

```
x = -7;

if x > 0:
    print('Es positivo.')
else:
    if x < 0:
        print('Es negativo.')
    else:
        print('Es cero, ¡no queda otra!')

print('Este texto está siempre, es nuestro amigo fiel.')
```

lo cual resulta

```
Es negativo.
Este texto está siempre, es nuestro amigo fiel.
```

Python, como muchos otros lenguajes, permite condensar ese tipo de condicionales anidados en uno solo:

```
if x > 0:
    print('Es positivo.')
else:
    if x < 0:
        print('Es negativo.')
    else:
        print('Es cero.')

if x > 0:
    print('Es positivo.')
elif x < 0:
    print('Es negativo.')
else:
    print('Es cero.)
```

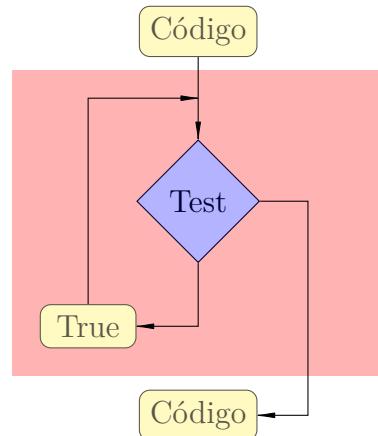
Ambos códigos hacen lo mismo. En definitiva:

```
elif = else + if
```

Un programa ramificado que sólo tiene bloques condicionales es un **programa de tiempo constante**: existe una *máxima cantidad de pasos* fija, independiente de los datos de entrada, que da el programa para finalizar. Surge naturalmente la pregunta: ¿qué pasa si desconozco a priori la cantidad de datos de entrada? Veremos a continuación cómo resolver este problema.

## 1.6. Ciclos iterativos e ingreso de datos

Una **iteración** consta de un Test que decide *repetir* o no el código del bloque “True”. El programa del dibujo ejecuta primero el Código y después evalúa el Test. Esta primera parte es idéntica a la del bloque condicional: si se satisface la condición dada por el Test, la ejecución sigue con el código del bloque “True”. Pero luego, *se vuelve a hacer el Test*. El programa seguirá pasando indefinidamente por el bloque “True”, hasta que deje de cumplirse la condición del Test. De ahí, se prosigue con la lectura del Código consecutivo.



Construiremos un programa para contar la cantidad de números positivos ingresados durante la ejecución del programa. Recibimos datos durante la ejecución con `input`, e implementamos un ciclo iterativo con `while`.

```

funcionando = True
positivos = 0

while funcionando:
    x = input('Dame un número: ')

    if x == '':
        funcionando = False
    else:
        x = float(x) # convierte a número

    if x > 0:
        positivos = positivos + 1

print('Conté', positivos, 'números positivos.')

```

Notemos una vez más la importancia de las **sangrías**, que separan diferentes ramas del código: iteraciones, condicionales y expresiones más simples.

Este programa pedirá números hasta que le demos una respuesta vacía, es decir, hasta que pulsamos **Enter** sin haber escrito ningún texto junto al mensaje ‘Dame un número:’.

El programa inicia **funcionando**, así que esta variable está en **True**, y como no tiene aún números ingresados, la cantidad de **positivos** es 0. El ciclo **while** hace el Test. Si el programa está **funcionando**, se pide un número con **input**, que se guarda en **x**.

```

IPython console
Console 1/A
Dame un número: 3.5
Dame un número: 9
Dame un número: 0
Dame un número: -1
Dame un número: 1
Dame un número: 56
Dame un número: 312809132.23
Dame un número: 1e-10
Dame un número:
Conté 8 números positivos.

In [3]: 

```

Si nuestra respuesta **x** está vacía, **funcionando** pasa a ser **False**. Cuando se vuelva a hacer el Test, dará **False**. El intérprete saltará el **while** y mostrará en la consola la cantidad de números positivos ingresados.

Si la respuesta no es vacía, *convertimos x a número con float* y chequeamos que sea positivo, en cuyo caso aumentamos la variable **positivos** en 1. La asignación del tipo **positivos = positivos + 1** es tan común, que **python** tiene una forma compacta de expresarlo:

**positivos = positivos + 1**      →      **positivos += 1**

## 1.7. Conversión y obtención de tipos

En la sección anterior, se ve cómo `float(x)` convierte `x` a un número. En realidad, `float` es uno de los tipos de variable numérica en `python`. Todo número con parte fraccionaria pertenece particularmente a esta clase. Si bien `x` era una cadena en el último ejemplo, no es necesario que lo sea siempre.

```
In [32]: float(327)
Out[32]: 327.0
In [33]: float(True)
Out[33]: 1.0
In [34]: float('inalterable')
ValueError: could not convert string to float:
'inalterable'
```

`float(valor)` convierte el valor a un número fraccionario , siempre que a `python` le resulte posible realizar dicha conversión.

Hemos trabajado también con otras clases de datos. Como `float`, cada una tiene un comando asociado, que se llama como su tipo. El comando convierte al tipo de variable en cuestión, cuando es posible. Usamos `type(expr)` para saber el tipo asociado al valor de una expresión.

```
In [35]: type(23.7)
Out[35]: float
In [36]: type(23)
Out[36]: int
In [37]: type(True)
Out[37]: bool
In [38]: type('Texto')
Out[38]: str
```

Entonces `int(valor)` convierte `valor` a un número entero, `bool(valor)` convierte `valor` a booleano, y `str(valor)` lo convierte a cadena.

```
In [39]: int(59.1)
Out[39]: 59
In [40]: int(True)
Out[40]: 1
In [41]: int('23')
Out[41]: 23
In [42]: bool(0)
Out[42]: False
In [43]: str(278 + 3.5)
Out[43]: '281.5'
```

## 1.8. Variables no escalares

Como ya se vio en la sección 1.2, página 10, `int`, `float` y `bool` son tipos *escalares*, mientras que `str` es *no escalar*. En esta sección daremos un pantallazo de lo que podemos hacer con los tipos no escalares `str` y `list`.

### 1.8.1. Cadenas de caracteres (`str`)

Las cadenas son texto encerrado por pares de comillas simples o dobles.

```
'Esto es una cadena.'      "Esto también."
```

En python, concatenamos cadenas “sumándolas”. Las repetimos mediante “multiplicación” por un número entero.

```
In [44]: 'todo ' + 'junto'
Out[44]: 'todo junto'
In [45]: contrato = 'se puede romper'
In [46]: '¿' + contrato + '?'
Out[46]: '¿se puede romper?'
In [47]: contrato += '.'
In [48]: contrato
Out[48]: 'se puede romper.'
In [49]: contrato = 'Que no ' + contrato
In [50]: contrato += ' '
In [51]: contrato * 2
Out[51]: 'Que no se puede romper. Que no se puede
romper.'
```



Los operadores `+` y `*` producen distintos resultados en base al tipo que tienen los operandos: se dice que son **operadores sobrecargados** [1].

número + número es la suma usual

`str + str` concatena

número \* número es el producto  
usual

`str * número` o `número * str`  
repite `str` tantas veces como  
indica el número

Accedemos a caracteres de la cadena con números entre corchetes.

```
In [52]: contrato[0]
Out[52]: 'Q'
In [53]: contrato[7]
Out[53]: 's'
In [54]: contrato[-2]
Out[54]: .'
```

Los números negativos cuentan desde el final de la cadena.

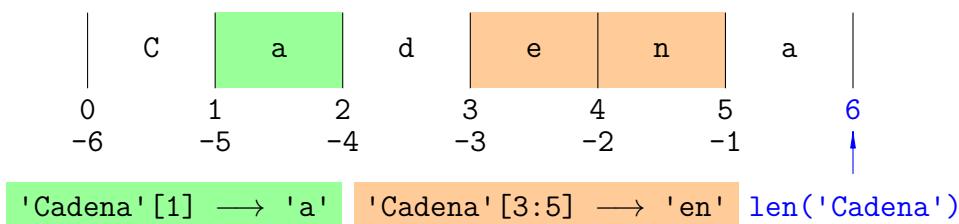
Extraemos caracteres consecutivos de la cadena con pares de números entre corchetes, separados por el signo de dos puntos.

```
In [55]: contrato[10:15]
Out[55]: 'puede'
In [56]: contrato[7:]
Out[56]: 'se puede romper. '
In [57]: contrato[:15]
Out[57]: 'Que no se puede'
In [58]: contrato[7:-9]
Out[58]: 'se puede'
```

Nuevamente, un negativo  $-i$  equivale a longitud de cadena  $- i$ . La longitud (cantidad de caracteres en la cadena) se obtiene con `len`.

```
In [59]: len(contrato)
Out[59]: 24
In [60]: contrato[-6] == contrato[len(contrato)-6]
Out[60]: True
```

RESUMEN:



No pueden modificarse partes de cadenas mediante asignación.

```
In [61]: contrato[16:] = 'comer'
TypeError: 'str' object does not support item assignment
```

Se dice que las cadenas son **inmutables**. Nunca modificamos cadenas, sino que creamos nuevas.

```
In [62]: rosca_prohibida = contrato[4:16] + 'comer'
In [63]: rosca_prohibida
Out[63]: 'no se puede comer'
```

La inmutabilidad de una variable no escalar es algo que no tiene por qué preocuparnos. Podemos “modificar” una cadena con el siguiente truco:

```
In [64]: frase = 'Se hace en un dique'
In [65]: frase = frase[:14] + 'to' + frase[16:]
In [66]: frase
Out[66]: 'Se hace en un toque'
```

Escribimos “modificar” entre comillas porque, técnicamente, no cambiamos la cadena original. Lo que hacemos es crear una nueva cadena a partir de la original y sobreescribir el nombre de la variable que la tenía asignada.

Por otro lado, la *mutabilidad* sí puede llegar a ser un *problema*, y por tal razón, se aborda especialmente en la sección 1.8.2, página 22.

Destacamos, por último, un operador interesante para tipos no escalares: el **operador in**. En particular, cuando el tipo de variables empleadas es **str**, el operador nos dice si una cadena está dentro de otra.

```
In [67]: 'palabra' in 'frase de cuatro palabras'
Out[67]: True
In [68]: 'x' in 'aeiou'
Out[68]: False
```

Como ya se adelantó, esto fue sólo un pantallazo sobre **str**. Pueden ver más funcionalidades y ejemplos en la documentación oficial (está en inglés):

<https://docs.python.org/3/tutorial/introduction.html#strings>.

### 1.8.2. Listas (**list**)

Volvamos al ejemplo del contador de números positivos introducido en la sección 1.6. El objetivo ahora no es simplemente obtener la cantidad de números positivos ingresados, sino que es *obtener los números “per se”*.

Los *guardamos* en **listas**. Se declaran con sus elementos entre corchetes, separados por coma. Dichos elementos pueden ser de cualquier tipo.

```
In [69]: primos = [2, 3, 5, 7, 11, 13, 17]
In [70]: palabras = ['comer', 'rezar', 'amar']
In [71]: estudiante = ['Fede', 'Bosio', 26, True]
```

En el último ejemplo, se ve que *los elementos pueden incluso tener tipos diferentes entre sí*. Un tipo no escalar como **list**, que contiene expresiones de diferentes tipos, se llama **tipo compuesto**.

Las operaciones entre listas son muy parecidas a las de **str**.

Concatenamos con + y repetimos elementos con \*

```
In [72]: primos + [19, 23]
Out[72]: [2, 3, 5, 7, 11, 13, 17, 19, 23]
In [73]: palabras += ['estudiar'] * 5
In [74]: palabras
Out[74]: ['comer', 'rezar', 'amar', 'estudiar',
           'estudiar', 'estudiar', 'estudiar', 'estudiar']
```

Con esto, ya podemos cambiar el contador del ejemplo de la sección 1.6 para que nos muestre los números positivos que vamos ingresando.

```
funcionando = True
positivos = [] # <- lista vacía

while funcionando:
    x = input('Dame un número: ')
    if x == '':
        funcionando = False
    else:
        x = float(x)

    if x > 0:
        positivos += [x] # agregamos x a la lista

print('Números positivos ingresados:')
print(positivos)
```

Los comentarios corresponden exactamente a los cambios efectuados en el código de la sección 1.6. Más específicamente:

1. la variable `positivos`, antes numérica, es ahora una `list`;
2. cuando el número ingresado por el usuario es positivo, se añade a la lista de `positivos`, en vez de incrementar una variable numérica que antes usábamos como contador.

Las variables de tipo `list` tienen los mismos operadores de acceso que las variables de tipo `str`:

- acceso a elementos con corchetes `[]`;
- acceso a sublistas con corchetes y dos puntos `[::]`.

Además, la cantidad de elementos de la lista se obtiene con la función `len`. Estas operaciones son idénticas a las empleadas con `str`.

```
In [75]: primos[4]
Out[75]: 11
In [76]: primos[:3]
Out[76]: [2, 3, 5]
In [77]: len(primos)
Out[77]: 7
In [78]: palabras[0]
Out[78]: 'comer'
In [79]: palabras[0][-1]
Out[79]: 'r'
```

El último ejemplo muestra un *doble acceso*: primero, al elemento 0 de la lista `palabras` (o sea, 'comer'), y segundo, al último elemento del valor obtenido (es decir, 'comer'[-1], que es 'r'). El *doble acceso* es una propiedad del tipo compuesto, y el *acceso* podría llegar a ser triple, cuádruple o, en general, *múltiple*, dado que podemos armar *listas dentro de otras listas*:

```
In [80]: personas = [estudiante, ['Juan', 'Pepe', 18, True], ['Arnaldo', 'Poroto', 67, False]]
In [81]: personas[0][1][2]
Out[81]: 's'
In [82]: matriz_rectangular = [[0]*3]*5 # es de 5x3
```

El funcionamiento del operador `in` se mantiene igual que para `str`.

```
In [83]: 7 in primos
Out[83]: True
In [84]: 'ranchear' in palabras
Out[84]: False
In [85]: ['Arnaldo', 'Poroto', 67, False] in personas
Out[85]: True
In [86]: 'Fede' in personas[0]
Out[86]: True
```

Una diferencia importante con los `str` es que las `list` son **mutables**.

```
In [87]: primos += [19, 22, 29]
In [88]: primos[-2] = 23
```

Jahora no hay error! Los elementos de las `list` pueden modificarse. En efecto:

```
In [89]: primos
Out[89]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Tabién podemos modificar sublistas completas.

```
In [90]: estudiante[:2] = ['Guara', 'Ni']
In [91]: estudiante[1:3] = []
In [92]: estudiante
Out[92]: ['Guara', True]
```

Con la *mutabilidad* de las listas podemos hacer cosas como esta:

```
In [93]: otras_palabras = palabras
In [94]: otras_palabras[-1] = 'ranchear'
In [95]: palabras
Out[95]: ['comer', 'rezar', 'amar', 'estudiar',
           'estudiar', 'estudiar', 'estudiar', 'ranchear']
```

O sea, modificar la lista `otras_palabras` produjo un efecto secundario: cambió la lista de `palabras` original. Esto sucede porque ambas listas tienen una **referencia compartida** en la memoria de la computadora [2].

Hay que tener cuidado con el comportamiento de las variables mutables. La *mutación accidental* puede llegar a ser difícil de detectar, y *para evitarla*, se usa `copy` [1], [2].

```
In [96]: palabras = otras_palabras.copy()
In [97]: palabras[-1] = 'estudiar'
In [98]: otras_palabras
Out[98]: ['comer', 'rezar', 'amar', 'estudiar',
           'estudiar', 'estudiar', 'estudiar', 'ranchear']
```

A `copy` se lo denomina *método*. Tanto `str` como `list` tienen métodos, y a lo largo del libro aparecerán algunos. La sección 5.1 tiene los rudimentos teóricos correspondientes.

Esta sección fue, nuevamente, un mero pantallazo. Para más detalles, ver

<https://docs.python.org/3/tutorial/introduction.html#lists>

El tipo de variable más importante es `list` porque hace de python lo que se llama un **lenguaje Turing-completo**. Sólo necesitamos `list`, `if` y `while` para implementar cualquier algoritmo existente.

**Programar en python es manejar `list`, `if` y `while`.**

**¡Todo lo demás es agregado!**

No obstante, los agregados son muy útiles para simplificar y reducir el código, así que los veremos en lo que resta del capítulo.

## 1.9. El ciclo `for`

Un “agregado” de python es el ciclo `for`. El `for` es muy común en los lenguajes de programación, por el siguiente motivo.

```
numero = 1
while numero < 20:
    print(numero**2)
    numero += 3

for numero in range(1, 20, 3):
    print(numero**2)
```

La ventaja es bastante obvia: reducimos el código a la mitad.

Notemos la presencia del operador `in` en el código del `for`. Esto se usa para que la variable `numero` tome todos los valores que están en el rango establecido por `range(1, 20, 3)`. Así, `1 <= numero < 20`, y por cada repetición del `print`, a `numero` se le suma 3.

Recordemos que `in` funciona con `str` y `list`. Usarlo en un `for` para recorrer los elementos de un `str` o un `list` es completamente legal:

```
for letra in 'Vertical':
    print(letra)

for palabra in ['Pablito', 'clavó', 'un', 'clavito']:
    print('La palabra', palabra, 'tiene', len(palabra),
          'letras.')
```

¿Para qué usar `while` entonces, si tengo `for` para recorrer rangos numéricos, cadenas y listas? Porque los datos a recorrer no siempre están definidos de antemano. En los ejemplos de las secciones 1.6 y 1.8.2, de hecho, el usuario los define al ejecutar el código. Esto impide usar `for` porque, inicialmente, *no hay datos para recorrer*. Recién tenemos todos los datos al final del `while`.

**Usamos `while` cuando la cantidad de datos de entrada es indefinida** (generalmente, cuando dicha cantidad depende del usuario).

**Caso contrario, usamos `for`, porque simplifica el código.**

## 1.10. Funciones

Aplicamos un **mismo procedimiento** a **expresiones diferentes**.

```
texto1 = 'Sacame las vocales'
texto2 = 'Ovni del Uritorco'
```

```

texto_nuevo = ''
for letra in texto1:
    if letra.lower() not in 'aeiou':
        texto_nuevo += letra
print(texto_nuevo)

texto_nuevo = ''
for letra in texto2:
    if letra.lower() not in 'aeiou':
        texto_nuevo += letra
print(texto_nuevo)

```

El código tiene partes repetidas: hay *redundancia*.

Eliminamos la redundancia usando funciones.

```

def quitar_vocales(texto):
    texto_nuevo = ''
    for letra in texto:
        if letra.lower() not in 'aeiou':
            texto_nuevo += letra

    return texto_nuevo

print(quitar_vocales('Sacame las vocales'))
print(quitar_vocales('Ovni del Uritorco'))

```

Así, ambos códigos tienen el mismo resultado:

```
Scm ls vcls
vn dl rtrc
```

pero el segundo evita “copy-paste”: se dice que la función *recicla* código.

ACLARACIONES: `cadena.lower()` pasa todos los caracteres de la cadena a minúscula. Así, `'A'.lower()` es `'a'`, `'LeTrAs'.lower()` es `'letras'`, etc. `lower()` es lo que se llama un *método* de `str`, y tiene la misma naturaleza que el método `copy`, usado para copiar una `list` en la sección 1.8.2. Veremos qué es un método con más detalle en la sección 5.1.

### 1.10.1. Argumentos obligatorios yopcionales

La función `quitar_vocales` puede ser generalizada de la siguiente forma.

```
def quitar_letras(texto, letras):
    letras = letras.lower()
    texto_nuevo = ''

    for letra in texto:
        if letra.lower() not in letras:
            texto_nuevo += letra

    return texto_nuevo

print(quitar_letras('Sacame las vocales', 'aeiou'))
print(quitar_letras('Ovni del Uritorco', 'cdeluv'))
```

Al ejecutar el código, resulta:

```
Scm ls vcls
Oni ritoro
```

La función `quitar_letras` tiene dos **argumentos obligatorios**: `texto` y `letras`. Los espera en ese orden. Una función puede tener también **argumentos opcionales**, los cuales traen un valor por defecto.

```
def quitar_letras(texto, letras, distingue_mayusculas=False):
    texto_nuevo = ''
    if not distingue_mayusculas:
        letras = letras.lower()
        texto = texto.lower()

    for letra in texto:
        if letra not in letras:
            texto_nuevo += letra

    return texto_nuevo

print(quitar_letras('Me hago Alto Guiso', 'aem'))
print(quitar_letras('Me hago Alto Guiso', 'aem', True))
```

Al ejecutar, se produce la salida siguiente.

```
hgo lto guiso
M hgo Alto Guiso
```

La línea `print(quitar_letras('Me hago Alto Guiso', 'aem', True))` también puede ser escrita como cualquiera de las siguientes

```
print(quitar_letras(letras='aem', distingue_mayusculas=True,
                    texto='Me hago Alto Guiso'))
print(quitar_letras(distingue_mayusculas=True,
                    texto='Me hago Alto Guiso', letras='aem'))
```

en definitiva, cuando especificamos argumentos con `=`, no hace falta seguir un orden específico. Hay otras formas combinadas para especificar argumentos, pero las vistas hasta ahora serán suficientes para nuestros usos. Al que le interesen los detalles, puede consultar <https://docs.python.org/3/tutorial/controlflow.html#defineining-functions>.

### 1.10.2. Múltiples valores de salida

En la sección 1.10.1 vimos que una función puede tener múltiples argumentos, o entradas. Si se quisiera construir una función con varias salidas, usando lo que vimos hasta ahora, podría usarse por ejemplo una `list` de la siguiente manera.

```
def mono_a_estereo(nombre_archivo):
    if '.' in nombre_archivo:
        raiz = nombre_archivo[:nombre_archivo.rindex('.')]
        sufijo = nombre_archivo[nombre_archivo.rindex('.'):]

    else:
        raiz = nombre_archivo
        sufijo = ''

    nombre_izq = raiz + '.L' + sufijo
    nombre_der = raiz + '.R' + sufijo

    return [nombre_izq, nombre_der]

nombres = mono_a_estereo('bailanta.wav')
izq = nombres[0]
der = nombres[1]
print(izq)
print(der)
```

cuya ejecución produce

```
bailanta.L.wav
bailanta.R.wav
```

es decir, la función toma una cadena que representa el nombre de un archivo, en el ejemplo es un archivo de audio, y devuelve dos cadenas, con .L y .R insertado en medio. La instrucción `nombre_archivo.rindex('.)')` devuelve el índice más grande del carácter '.' (punto). En otras palabras, dada una cadena `s` y un carácter `c`, `s.rindex(c)` devuelve el máximo número entero `i` para el cual `s[i]` es igual a `c`. Es otro método de `str`. Estudiaremos qué es un método con más detalle, en la sección 5.1. Por ahora, los curiosos deberán conformarse con el siguiente enlace:

<https://docs.python.org/3/library/stdtypes.html#string-methods>

En el ejemplo, hemos tenido que crear tres variables (`nombres`, `izq` y `der`) para guardar sólo dos valores (`izq` y `der`), producidos al llamar a la función `mono_a_estereo`. Hay un atajo sintáctico para estas situaciones.

```
def mono_a_estereo(nombre_archivo):
    if '.' in nombre_archivo:
        raiz = nombre_archivo[:nombre_archivo.rindex('.')]
        sufijo = nombre_archivo[nombre_archivo.rindex('.'):]

    else:
        raiz = nombre_archivo
        sufijo = ''

    nombre_izq = raiz + '.L' + sufijo
    nombre_der = raiz + '.R' + sufijo

    return nombre_izq, nombre_der

izq, der = mono_a_estereo('bailanta.wav')
print(izq)
print(der)
```

y esto produce la misma salida que antes.

Hemos realizado estos cambios:

1. eliminar los corchetes a la derecha de la instrucción `return`;
2. sustituir la variable `nombres`, bajo la definición de la función, por el par de variables `izq`, `der` separadas por comas.

Entonces, manejamos **múltiples argumentos de salida** de una función cuando **especificamos variables separadas por comas**. Esto aplica tanto al definir la función, como para obtener los valores producidos durante una llamada a dicha función.

En realidad, esto de los argumentos múltiples tiene como fundamento una capacidad de python denominada **asignación múltiple** [1], o **asignación de secuencias** [2]. Para python, los siguientes códigos son equivalentes.

```
x = 1           x, y = 1, 2
y = 2           x, y = y, x
temp = y
y = x
x = temp
```

### 1.10.3. Recursión

La sucesión de Fibonacci es uno de los ejemplos más famosos de sucesiones recursivas por la excesiva veneración que muestran por ella ciertos canales de YouTube y libros de divulgación. Su definición matemática no es otra que

$$f_0 = f_1 = 1, \quad f_n = f_{n-1} + f_{n-2} \quad \text{para } n \geq 2.$$

La sucesión es **recursiva** porque se define en términos de sí misma. Es fácil implementarla en python.

```
def fibonacci(n):
    if n <= 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))
print(fibonacci(35))
```

La función `fibonacci` se llama a sí misma. Toda función recursiva es así.

Si bien el primer valor (`fibonacci(10)`, que es 89) se muestra casi instantáneamente en la Terminal, el otro (14930352) se toma su tiempo para aparecer (a mi computadora le toma poco menos de 20 segundos, es bastante). Hay una forma iterativa que funciona bastante más rápido.

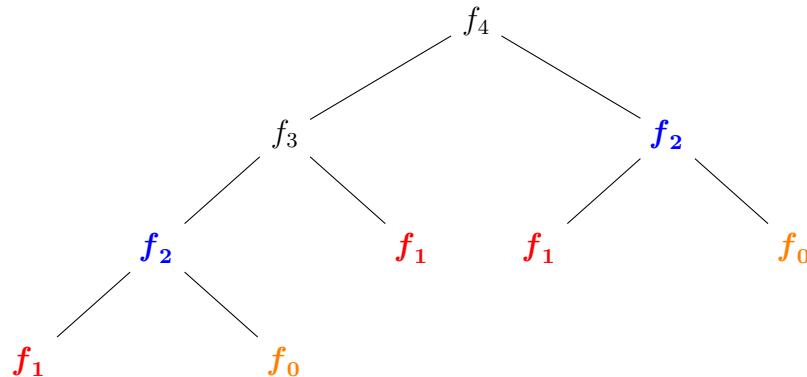
```
def fibo_iter(n):
    a = b = 1
    for i in range(n):
        a, b = b, a + b
    return b
```

Mi computadora tarda *menos de 60 microsegundos* en computar el valor de `fibo_iter(35)` ( $60 \mu\text{s}$  contra 20 s no tiene comparación!). La medición puede hacerse en la Terminal con el comando `%time`:

```
In [99]: %time fibo_iter(35)
CPU times: user 30 µs, sys: 3 µs, total: 33 µs
Wall time: 50.8 µs
Out[99]: 14930352
```

El comando `%time` sólo puede ser invocado desde la Terminal: no es una instrucción que podamos incluir en un programa. El intérprete de python en realidad no la entiende, sino que `%time` es un agregado que viene con `spyder`.

La diferencia colosal de tiempos se debe a la ineeficiencia de la forma recursiva implementada para calcular los términos de la sucesión de Fibonacci. Ejemplo, para `fibonacci(4)`, el proceso de cálculo es el siguiente.



Es decir, como  $f_4 = f_3 + f_2$ , entonces requerimos  $f_3$  y  $f_2$  para calcular  $f_4$ . Esa es la primera ramificación. Después, para calcular  $f_3$ , se requieren  $f_2$  y  $f_1$ , y así se va formando el árbol completo.

Notemos que hace falta llegar hasta el valor de `fibonacci(1)` **tres veces**, cuando la forma iterativa lo lee sólo una sola vez. De ahí que la forma recursiva sea más lenta que la iterativa.

Si bien la forma iterativa resultó ser muchísimo más rápida en este caso, existen otros en donde sólo se conoce una expresión en forma recursiva, e implementarla directamente toma menos tiempo y es más fácil que pensar una forma iterativa distinta de hacerlo. Tal es el caso de, por ejemplo, una función que devuelve las permutaciones de una cadena determinada.

### 1.10.4. Funciones de alto orden

Una función es de **alto orden** cuando **al menos uno de sus argumentos es otra función**. Se usan especialmente en diseño de interfaces gráficas y resolución numérica de ecuaciones diferenciales.

Ejemplo:

```
def derivada(funcion, punto, incremento=1e-4):
    return ((funcion(punto+incremento)-funcion(punto))
            / incremento)

def funcion_lineal(x):
    return 3*x + 2

print(derivada(funcion_lineal, 1))
```

devuelve

3.0000000000189

Más adelante veremos ejemplos concretos en donde se usan funciones de alto orden, particularmente en las secciones 3.4.2 y 6.2.

## 1.11. Módulos de python

Los códigos de esta sección pueden calcular la *carga másica* en un tanque de aireación, en base a datos como el caudal de agua a su entrada y concentraciones de microorganismos en el agua. Estos parámetros se usan para modelar los tanques de aireación, que forman parte de las plantas de *tratamiento de aguas residuales*, tema que concierne a la Ingeniería Ambiental.

Los siguientes archivos tienen una línea de código repetida.

Archivo <code>carga_a.py</code>	Archivo <code>carga_b.py</code>
<pre>caudal = 5000 / (24*60**2) sustrato = 220 biomasa = 2200 tiempo_residencia = 4 * 60**2  volumen = (tiempo_residencia            * caudal) carga_masica = (caudal *                  sustrato / (volumen*biomasa))</pre>	<pre>caudal = 5 / 1000 sustrato = 330 biomasa = 1800 profundidad = 5 area = 60  volumen = profundidad * area carga_masica = (caudal *                  sustrato / (volumen*biomasa))</pre>

Eliminamos esta redundancia creando un **módulo**.

Creamos un archivo `carga_modulo.py` en el mismo directorio que los otros dos, y definimos dentro la siguiente función.

```
def calcular_carga_masica(caudal, sustrato, volumen, biomasa):
    return caudal * sustrato / (volumen*biomasa)
```

Ahora, modificamos los archivos `carga_a.py` y `carga_b.py` así:

Archivo `carga_a.py`

---

```
import carga_modulo

caudal = 5000 / (24*60**2)
sustrato = 220
biomasa = 2200
tiempo_residencia = 4 * 60**2

volumen = tiempo_residencia * caudal
carga_masica = carga_modulo.calcular_carga_masica(
    caudal, sustrato, volumen, biomasa
)
```

Archivo `carga_b.py`

---

```
import carga_modulo

caudal = 5 / 1000
sustrato = 330
biomasa = 1800
profundidad = 5
area = 60

volumen = profundidad * area
carga_masica = carga_modulo.calcular_carga_masica(
    caudal, sustrato, volumen, biomasa
)
```

De esta manera, los archivos `carga_a.py` y `carga_b.py` comparten la función `calcular_carga_masica`, definida en el archivo `carga_modulo.py`.

Podemos prescindir del prefijo “`carga_modulo.`” para llamar a la función `calcular_carga_masica` usando

```
from carga_modulo import calcular_carga_masica
```

en lugar del simple `import carga_modulo`.

Si bien la redundancia en el código del ejemplo es mínima, la cosa puede ponerse difícil cuando aumenta el alcance del proyecto, y por ende, el tamaño del código. Es en estos casos que los módulos cobran importancia.

### 1.11.1. Biblioteca estándar

Siendo `python` un lenguaje de *propósito general*, ya cuenta con **módulos incorporados**. Algunos de ellos son `math`, `statistics` y `random` para matemática, estadística y generación de valores aleatorios, `calendar` y `date` para manipular fechas, `os` para manejar archivos y otras cuestiones del sistema operativo, entre muchos otros más. La lista exhaustiva está en

<https://docs.python.org/3/library/index.html>.

`Python` puede ser extendido también mediante *módulos no-estándar*. Afortunadamente, `anaconda` trae instalado consigo muchos módulos no-estándar para computación científica, que son los que usaremos en las secciones siguientes para tratar cuestiones específicas de ingeniería. Cuando la cosa se ponga muuuuy específica, instalaremos los módulos que hagan falta.

## 1.12. Comentario final

Este capítulo sólo fue un pantallazo general de las cosas más básicas que podemos hacer con `python`. Aquí hemos visto todo lo necesario para avanzar con las **aplicaciones** y llevar a cabo **proyectos que funcionan**. Al que quiera volverse un crack de `python` y la programación, recomendamos el tutorial oficial: <https://docs.python.org/3/tutorial/index.html>, junto con la bibliografía complementaria que aparece en las **Referencias**.

## Referencias

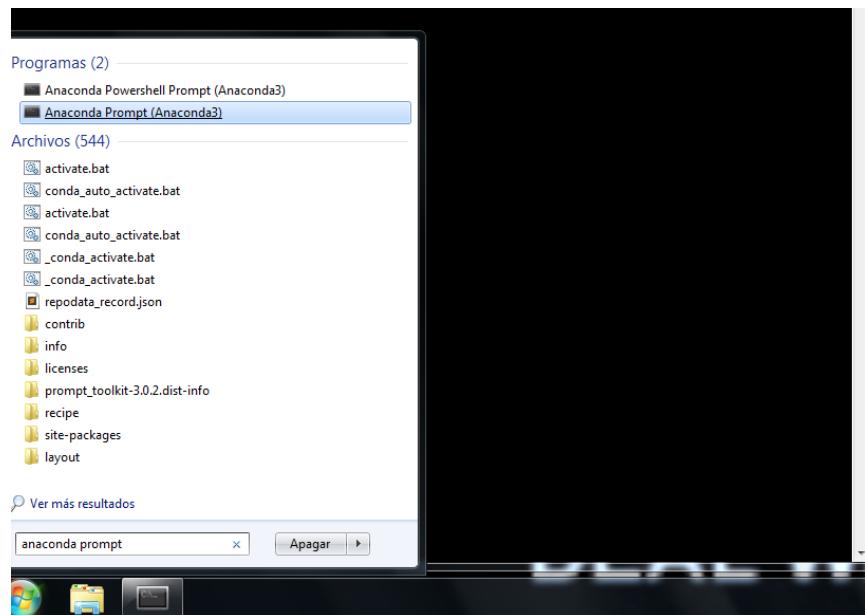
- [1] J. V. Guttag, *Introduction to computation and programming using Python*, 2.<sup>a</sup> ed. United States of America: Massachusetts Institute of Technology Press, 2016 (vid. págs. 7, 8, 10, 13, 18, 23, 29).
- [2] M. Lutz, *Learning Python*, 5.<sup>a</sup> ed. United States of America: O'Reilly Media, 2013 (vid. págs. 8, 11, 12, 23, 29).

# Capítulo 2

## Procesamiento de audio

En este capítulo usaremos los módulos `soundfile` y `sounddevice` para cargar y manipular archivos de audio. Estos módulos no vienen incluidos por defecto en `anaconda`. Sin embargo, instalarlos es realmente simple y rápido.

1. Abrir Anaconda Prompt;



2. escribir `pip install soundfile sounddevice`
3. pulsar `Enter`, esperar un ratito, y listo.

Les dije que era simple y rápido.

## 2.1. El audio es vectorial

Importemos un archivo de audio mono. Debemos ejecutar `python` en el mismo directorio en el cual está el archivo. Cambiamos desde `spyder` el directorio de trabajo con el `Explorador de archivos`. Si el audio se llama `audio.wav`, entonces ponemos en la Terminal de `spyder`

```
In [1]: import soundfile as sf
In [2]: x, fs = sf.read('audio.wav')
In [3]: print(x)
Out[3]: [0. 0. 0. ... 0.00149536 0.0015564 0.0015564 ]
```

se trata ni más ni menos que de un **vector**. Estamos en uno de los apogeos del libro. A la hora de programar, **el audio es vectorial**. Esto es fundamental.

# AUDIO = VECTOR

¡No confundir *lista* con *vector*! Son dos estructuras de datos diferentes:

1. una *lista* es el tipo de datos `list`, propio de Python, discutido en la sección 1.8.2 de este mismo libro;
2. un *vector* es más parecido al concepto matemático. Se hace necesario un complemento para manejar esta estructura en `python`.

La diferencia se pone de manifiesto cuando ejecutamos

```
In [4]: type([0, -14, 15, 0.0015])
Out[4]: list
In [5]: type(x)
Out[5]: numpy.ndarray
```

más allá de la diferencia numérica, el tipo de datos es lo importante. El primero es una `list`, el segundo, un *vector de numpy*.

Naturalmente, surge la pregunta...

## 2.2. ¿Qué es numpy?

RESPUESTA: es el **módulo** que usamos en `python` para trabajar con **vectores**, bajo las **reglas algebraicas usuales**.

Cargamos `numpy` con `import`.

```
In [6]: import numpy as np
```

con este comando decimos a `python` que cargue `numpy` con el nombre `np`. Esta es una convención establecida. Así, cuando vemos `np` en un código, sabemos que se trata de `numpy`, y no de otra cosa.

Usamos el comando `array` para crear vectores.

```
In [7]: x = np.array([7, 8, 32, -27])
```

Atentos a los **corchetes y paréntesis**. La función `array` definida en el módulo `numpy` toma una `list` como argumento, y por eso hay que escribirla así para que funcione. Estaría *mal* escribir `np.array(7, 8, 32, -27)`.

Ejecutando la sentencia correcta, *almacenamos un vector de números enteros en la memoria*. ¿Y si quiero decimales?

```
In [8]: y = np.array([0.2, 5, -0.8, -0.4])
```

Aún habiendo declarado al vector `y` con una componente entera, basta que exista una decimal (`float`) para que todas las demás lo sean. Ergo, la segunda componente no es igual al número entero 5 sino al 5.0.

```
In [9]: y[1]
Out[9]: 5.0
```

Ya tenemos tres cosas importantes sobre la función `array` de `numpy`:

1. toma una lista;
2. convierte todos sus elementos a un mismo tipo;
3. devuelve un vector, en el sentido algebraico usual.

### 2.2.1. Rutinas de creación de vectores

Hay otras formas de crear vectores además del comando `array`.

```
In [10]: x = np.arange(2.22, 7.5, 1.5)
```

```
In [11]: y = np.linspace(2.22, 6.72, 4)
```

Ambos crean *intervalos* de valores. La forma de hacerlo es un tanto distinta.

El primero (`arange`) los crea en secuencia. El vector `x` se calcula así

$$2.22, \quad 2.22 + 1.5, \quad (2.22 + 1.5) + 1.5, \quad ((2.22 + 1.5) + 1.5) + 1.5$$

Las componentes deben ser estrictamente menores que 7.5 para figurar en `x`.

El segundo (`linspace`) parte un intervalo en porciones de igual tamaño. El vector `y` se calcula entonces de la siguiente manera:

$$2.22 + \left( \frac{6.72 - 2.22}{4-1} \right) 0, \quad 2.22 + \left( \frac{6.72 - 2.22}{4-1} \right) 1, \quad 2.22 + \left( \frac{6.72 - 2.22}{4-1} \right) 2, \quad 2.22 + \left( \frac{6.72 - 2.22}{4-1} \right) 3$$

Muchas veces necesitaremos crear vectores nulos y vectores conformados en su totalidad por unos:

```
In [12]: np.zeros(4)
Out[12]: array([0., 0., 0., 0.])
In [13]: np.ones(7)
Out[13]: array([1., 1., 1., 1., 1., 1., 1.])
```

Hay algunas rutinas más. Por ahora nos quedaremos con estas.

## 2.2.2. Operaciones vectoriales

En `numpy` trabajamos con la suma, resta y producto por escalar de igual manera que hacíamos con vectores de  $\mathbf{R}^n$ .

```
In [14]: x = np.array([4, 0, -1, 0.2])
In [15]: y = np.array([0.5, 3, -2, 1])
In [16]: x + y
Out[16]: array([ 4.5, 3. , -3. , 1.2])
In [17]: x - y
Out[17]: array([ 3.5, -3. , 1. , -0.8])
In [18]: -4 * x
Out[18]: array([-16. , -0. , 4. , -0.8])
```

PRIMER NOVEDAD: producto, división y potenciación *también* se hacen **componente a componente**. Es más: `numpy` opera **componente a componente** con *casi* cualquier función por defecto. Ejemplos:

```
In [19]: x * y
Out[19]: array([2. , 0. , 2. , 0.2])
In [20]: x / y
Out[20]: array([8. , 0. , 0.5, 0.2])
In [21]: x**y
Out[21]: array([2. , 0. , 1. , 0.2])
In [22]: np.sin(2.5*np.pi*x)
Out[22]: array([-1.2246468e-15, 0.0000000e+00,
 -1.0000000e+00, 1.0000000e+00])
```

La cuarta operación incluye al número  $\pi$  definido por `numpy` como `pi`, y la función seno, definida por `sin`. La biblioteca trae cargadas ya las constantes  $\pi, e, \gamma$  definidas respectivamente `pi`, `e`, `euler_gamma`. Es importante distinguir la constante `e` de la `e` en el último resultado. La primera es la constante  $2.718281828459045\dots$ , y la segunda `e` tiene, a su derecha, el exponente del número en la notación científica. Así

$$-1.2246468e-15 = -1.2246468 \cdot 10^{-15} \approx 0$$

$$0.0000000e+00 = 0.0000000 \cdot 10^0 = 0$$

pero, por otro lado, `e` =  $2.718281828459045\dots$

Además de las trigonométricas, `numpy` trae muchas funciones matemáticas por defecto (raíces, exponenciales, hiperbólicas, inversas, etc...) pero atención, ¡no toda función se aplica componente a componente (por eso más arriba está escrito y remarcado ese *casi*)! La sumatoria y algunos descriptores estadísticos contraen el vector a un escalar:

```
In [23]: np.sum(x)
Out[23]: 3.2
In [24]: np.mean(x)
Out[24]: 0.8
In [25]: np.min(x)
Out[25]: -1.0
```

Hay listas exhaustivas en la documentación sobre funciones matemáticas y descriptores propios de la estadística en

<https://docs.scipy.org/doc/numpy/reference/routines.math.html>  
<https://docs.scipy.org/doc/numpy/reference/routines.statistics.html>

Son muy útiles, ténganlas a mano cuando programen.

SEGUNDA NOVEDAD: ¡podemos operar con vectores y escalares! No hace falta que ambos operandos sean siempre vectores para tener un resultado.

Ejemplos:

```
In [26]: 5 + x
Out[26]: array([9. , 5. , 4. , 5.2])
In [27]: 0.2 - x
Out[27]: array([-3.8, 0.2, 1.2, 0. ])
In [28]: x**2
Out[28]: array([16. , 0. , 1. , 0.04])
In [29]: 3/y
Out[29]: array([ 6. , 1. , -1.5, 3. ])
In [30]: 4**y
Out[30]: array([2.00e+00, 6.40e+01, 6.25e-02, 4.00e+00])
```

**Concatenamos vectores** con `hstack` (del inglés, *horizontal stack*, “amon-tonamiento horizontal”). **Pasamos una lista** de vectores, así:

```
In [31]: z = np.array([2, 3])
In [32]: np.hstack([x, y, z])
Out[32]: array([ 4.,  0., -1.,  0.2,  0.5,  3., -2.,
   1.,  2.,  3.])
```

Pero volvamos al audio.

### 2.2.3. Audio en estéreo

Casi nunca procesamos audio para dejarlo mono. El audio suele ser **estéreo**, tiene **más de un canal**. ¿Cómo carga numpy el audio estéreo? Si `audio2.wav` es el nombre de un archivo de audio estéreo, ubicado en el mismo directorio en el que estamos trabajando con la consola de python, entonces

```
In [33]: x, fs = sf.read('audio2.wav')
In [34]: x
Out[34]:
array([[ 0.0000000e+00,  0.0000000e+00],
       [-3.05175781e-05, -3.05175781e-05],
       [-3.05175781e-05, -3.05175781e-05],
       ...,
       [ 0.0000000e+00,  0.0000000e+00],
       [ 0.0000000e+00,  0.0000000e+00],
       [ 0.0000000e+00,  0.0000000e+00]])
```

**El audio estéreo se carga como matriz de dos columnas.**

La propiedad `shape` indica el **tamaño de la matriz**:

```
In [35]: x.shape
Out[35]: (661500, 2)
```

así, **las filas son muestras y las columnas son canales**. La duración del audio es  $(x.shape[0]) / fs \left(= \frac{661500 \text{ muestras}}{44100 \text{ muestras/segundo}} = 15 \text{ s}\right)$ .

El canal izquierdo es `x[:,0]` y el derecho `x[:,1]`. La primer muestra del canal derecho es `x[0,1]`. La última muestra del canal izquierdo es `x[-1,0]`.

Pasamos de tiempo a muestras con la frecuencia de muestreo  $f_s$ .

$$\boxed{T \cdot f_s = N \\ \text{segundos} \cdot (\text{muestras/segundo}) = \text{muestras}}$$

Por lo tanto, el primer segundo de audio estéreo es  $x[:fs, :]$ . El último segundo es  $x[-fs:, :]$ . El tramo de señal que abarca desde el segundo cuatro al siete es  $x[4*fs:7*fs]$  (son tres segundos de audio).

Por esto deben estudiarse las **matrices** en Ingeniería de **Sonido**.

#### 2.2.4. Rutinas de creación de matrices

Las rutinas para matrices son las mismas que para vectores, salvo sutiles diferencias y añadidos. La más elemental es el comando **array**, que toma una lista y devuelve un vector.

Para construir matrices con **array**, usamos listas de listas:

```
In [36]: A = np.array([[1, -1], [8, 0]])
In [37]: A
Out[37]:
array([[ 1, -1],
       [ 8,  0]])
```

Así, cada **sublista** es una **fila**.

También podemos usar racionales obviamente:

```
In [38]: B = np.array([
    ...: [1, -0.5],
    ...: [1, 0.5],
    ...: ])
```

Creamos matrices con **zeros** y **ones** empleando **pares ordenados** como argumentos, en vez de simples números:

```
In [39]: np.zeros((3, 5))
Out[39]:
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

Más sobre *creación de vectores y matrices* en la documentación oficial:

<https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html>

### 2.2.5. Concatenación de matrices

Tenemos dos archivos de audio estéreo con efectos de sonido:

1. alguien entrando a un auto (`auto1.wav`);
2. autos arrancando (`auto2.wav`).

Queremos unir estos dos audios para crear nuestro propio efecto. Los importaremos con `soundfile`. Las matrices resultantes tienen muchas filas pero solamente dos columnas. Se concatenan con `vstack`.

```
In [40]: x1, fs = sf.read('auto1.wav')
In [41]: x2, __ = sf.read('auto2.wav')
In [42]: x = np.vstack([x1, x2])
In [43]: x1.shape
Out[43]: (220500, 2)
In [44]: x2.shape
Out[44]: (104835, 2)
In [45]: x.shape
Out[45]: (325335, 2)
In [46]: x1.shape[0] + x2.shape[0] == x.shape[0]
Out[46]: True
```

**La suma de la cantidad de filas de las matrices individuales coinciden con las de la matriz concatenada.** Guardemos nuestro efecto:

```
In [47]: sf.write('auto.wav', x, fs)
```

DATO DE COLOR: `hstack` concatena matrices con igual cantidad de filas.

```
In [48]: C = np.hstack([A, B])
In [49]: C
Out[49]:
array([[ 1. , -1. ,  1. , -0.5],
       [ 8. ,  0. ,  1. ,  0.5]])
```

### 2.2.6. Operaciones matriciales

Podemos hacer combinaciones lineales de matrices, aplicar funciones componente a componente y extraer descriptores estadísticos.

```
In [50]: 2*A - 3*B
Out[50]:
array([[-1. , -0.5],
       [13. , -1.5]])
In [51]: np.cos(np.pi * (A/(B**A)))
Out[51]:
array([-1.000000e+00,  6.123234e-17],
      [ 1.000000e+00,  1.000000e+00])
In [52]: A.sum()
Out[52]: 8
```

`A.sum()` es lo mismo que `np.sum(A)`. En principio, `A.sum()` también contrae la matriz `A` a un escalar. Su manejo es *casi* el mismo que para vectores. ¡Este comportamiento puede cambiarse! Por ejemplo, **sumamos sólo las filas con `axis=0` y sólo las columnas con `axis=1`**

```
In [53]: A.sum(axis=0)
Out[53]:
array([ 9, -1])
In [54]: A.sum(axis=1)
Out[54]:
array([0, 8])
```

En ambos casos, la **matriz** también fue **contraída**, ¡pero a un **vector!** en vez de un escalar. Por eso el *casi...* las matrices dan una posibilidad más que los vectores, podemos operar individualmente por filas o por columnas.

### 2.2.7. Expansión de vectores y matrices

*Generaremos una diente de sierra mediante síntesis aditiva.* El método de síntesis aditiva consiste en **generar una onda sumando sinusoidales**.

Conocemos la ecuación de la diente de sierra gracias a Fourier:

$$A[n] = \sum_{k=1}^K c_k \sin\left[2nk\pi \left(\frac{f_0}{f_s}\right)\right], \quad \text{con } c_k = \frac{(-1)^k}{k\pi} \quad (2.1)$$

Entran en juego algunos parámetros importantes:

- $K$  es el número de sinusoidales empleadas para formar la onda;
- $f_0$  es la frecuencia fundamental de la onda (su altura tonal);
- $f_s$  es la frecuencia de muestreo. Tomemos el valor estándar  $f_s = 44100$ .

La diente de sierra ideal se consigue cuando  $K \rightarrow \infty$ , nos conformaremos con un  $K$  finito:  $K = 10$  será suficiente. Tomaremos también  $f_0 = 1000$  Hz.

Fijados  $K$ ,  $f_0$  y  $f_s$  la fórmula todavía requiere manejar dos variables: el “tiempo discreto”  $n$  y el “número de armónico”  $k$ .

El tiempo discreto lo armamos con `arange`. Recuerden:

$$\boxed{T \cdot f_s = N \\ \text{segundos} \cdot (\text{muestras/segundo}) = \text{muestras}}$$

Por lo tanto, para tener una duración de, por ejemplo,  $T = 3$  segundos, la longitud del vector tiene que ser  $N = 3f_s$ .

Para generar las sinusoidales (llamadas “parciales” o “armónicos”), podemos usar un `for` con un índice  $k$  entre 1 y  $K$ , el número de armónicos. En cada iteración,  $k$  toma un valor fijo. Por ende, puede calcularse cada  $c_k$ . Después bastará multiplicar  $c_k$  por la función seno, evaluada en un múltiplo del vector `n`, para generar los sumandos. Es como dice la fórmula (2.1).

Aquí una implementación posible:

```
K = 10 # número de armónicos
f0 = 1000 # frecuencia fundamental
fs = 44100 # frecuencia de muestreo
N = 3 * fs # longitud de la señal (= 3 segundos)
n = np.arange(N) # tiempo discreto

A = np.zeros(N).astype(float)
for k in range(1, K + 1):
    ck = (-1)**k / (k*np.pi)
    A += ck * np.sin(2*n*k*np.pi*f0/fs)
```

**El `for` del código es innecesario cuando usamos expansión (o “broadcasting”, en inglés).**

Antes usamos expansión sin saberlo.

```
In [55]: 4 + np.array([-3, 0, 1, -2])
Out[55]: array([1, 4, 5, 2])
```

En el ejemplo, numpy *expande* el escalar a un vector para poder operar componente a componente.

$$\begin{aligned} & 4 + [-3 \ 0 \ 1 \ -2] \\ & [4 \ 4 \ 4 \ 4] + [-3 \ 0 \ 1 \ -2] \\ & [4 - 3 \ 4 + 0 \ 4 + 1 \ 4 - 2] \\ & [1 \ 4 \ 5 \ 2] \end{aligned}$$

Así ocurre con toda operación entre vectores y escalares. Lo interesante (y útil para nosotros) es que ocurra también con vectores y matrices.

```
In [56]: np.array([0, 1, 2]) + np.array([[3], [4]])
Out[56]:
array([[3, 4, 5],
       [4, 5, 6]])
```

En este caso, *la expansión es matricial*:

$$\begin{aligned} & \begin{bmatrix} 0 & 1 & 2 \end{bmatrix} + \begin{bmatrix} 3 \\ 4 \end{bmatrix} \\ & \begin{bmatrix} 0 & 1 & 2 \\ \textcolor{red}{0} & \textcolor{red}{1} & \textcolor{red}{2} \end{bmatrix} + \begin{bmatrix} 3 & \textcolor{red}{3} & \textcolor{red}{3} \\ 4 & \textcolor{red}{4} & \textcolor{red}{4} \end{bmatrix} \\ & \begin{bmatrix} 3 & 4 & 5 \\ 4 & 5 & 6 \end{bmatrix} \end{aligned}$$

`numpy expande` una fila y una columna agregando tantas filas a la primera como componentes tiene la segunda, y tantas columnas a la segunda como componentes tiene la primera.

La expansión se hace **para operar con ambas cantidades como matrices**.

Aprovechamos la *expansión* para generar una matriz de armónicos. A cada fila corresponde un instante de tiempo, y a cada columna, un armónico. Si notamos **A** (letra gruesa) a esta “matriz de armónicos”, entonces

$$\mathbf{A}[n, k] = c_k \operatorname{sen}\left[2nk\pi\left(\frac{f_0}{f_s}\right)\right] \quad (2.2)$$

con  $c_k$  definido por la ecuación (2.1).

Así, la combinación de (2.1) y (2.2) deviene

$$A[n] = \sum_{k=1}^K \mathbf{A}[n, k] \quad (2.3)$$

es decir,  $A[n]$  es la suma de las columnas de **A**.

El código queda entonces

```
k = np.arange(1, K + 1)
n = np.arange(N)[:, np.newaxis]
ck = (-1)**k / (k*np.pi)
A = (ck*np.sin(2*n*k*np.pi*f0/fs)).sum(axis=1)
```

`np.newaxis` acomoda las componentes del vector **n** en una matriz columna.

## 2.3. Reproducción de audio

Escuchamos el audio de un vector con `sounddevice.play`.

El código completo del ejemplo de la sección 2.2.7 queda

```
import numpy as np
import sounddevice as sd

K = 10 # número de armónicos
f0 = 1000 # frecuencia fundamental
fs = 44100 # frecuencia de muestreo
N = 3 * fs # longitud de la señal (= 3 segundos)

k = np.arange(1, K+1)
n = np.arange(N)[:, np.newaxis]
ck = (-1)**k / (k * np.pi)
A = (ck * np.sin(2*n*k*np.pi*f0/fs)).sum(axis=1)

sd.play(A, fs)
```

Si no suena nada, chequeá el volumen del parlante. Es muuuuy común olvidar subirlo en el momento más crucial.

## 2.4. Grabación de audio

Grabamos audio en una matriz de `numpy` con `sounddevice.rec`.

A modo de ejemplo, grabemos 5 segundos de audio. Recordamos para ello, una vez más, el importantísimo cuadro

$$\boxed{\begin{array}{rcl} T \cdot f_s & = & N \\ \text{segundos} \cdot (\text{muestras/segundo}) & = & \text{muestras} \end{array}}$$

o sea que la longitud del vector será  $N = 5f_s$ . Escribamos:

```
N = 5 * fs
canales = 1

x = sd.rec(N, fs, canales).ravel() # reduce a 1D
sd.wait() # esperar a que termine la grabación
sd.play(x, fs)
```

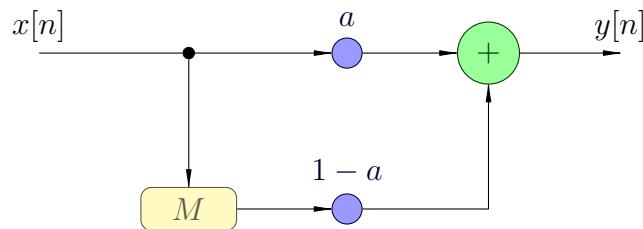
Si el código corre pero nada suena chequeá el volumen del parlante y el micrófono. Alguno de los dos debe estar bajo.

## 2.5. Efectos de audio en tiempo

Como el audio es representado en forma de matriz, **aplicar efectos** no es otra cosa que **manipular los elementos de una matriz**.

### 2.5.1. Eco

Agreguemos eco a nuestra voz. Para eso necesitamos aplicar un **retardo** a la señal y sumar a la original esta versión retardada. Esquemáticamente:

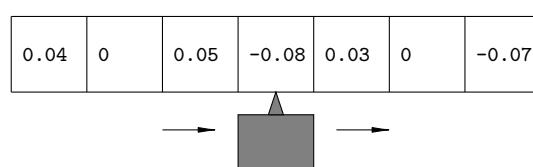


Estos diagramas de bloque son ampliamente utilizados en el estudio de señales y sistemas [1]. Veamos rápidamente qué significa cada cosa:

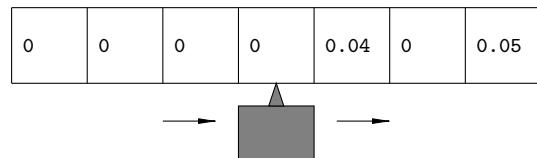
- $x[n]$  es la *señal de entrada* (nuestra grabación “ limpia ”) e  $y[n]$  es la *señal de salida* (la grabación con efecto). Denotamos entre corchetes a las variables independientes discretas, en este caso  $n$ .
- Los círculos  $\circlearrowright$  ( $a, 1 - a$ ) son *multiplicadores*. Como su nombre lo indica, multiplican señales por escalares. Algunos prefieren dibujar triangulitos en vez de círculos. Da igual realmente.
- El círculo  $\oplus$  es un *sumador* de señales.
- El bloque  $M$  aplica un *retardo* de  $M$  muestras.

Es fácil implementar sumas vectoriales y multiplicaciones por escalar con `numpy`, ya lo vimos. El problema es el retardo.

La función `play` de `sounddevice` lee secuencialmente los valores del vector `x` para enviarlos a la placa de audio. Es como una especie de “aparato con cabezal móvil” que lee algo escrito en una cinta.



Habrá retardo cuando el cabezal tarde más en alcanzar el inicio de la señal. Una forma de hacerlo es agregando ceros.



Así `sounddevice` leerá 0 por un tiempo y lo interpretará como silencio. Comenzará la reproducción efectiva en cuanto comience a leer los valores no nulos de señal que vienen inmediatamente después. En definitiva:

## Una señal se retarda agregándole ceros

**Agregamos ceros** con las rutinas `zeros` y `hstack` de `numpy`.

```
xdelay = np.hstack([np.zeros(3*fs), x]) # retardo de 3 s
sd.play(xdelay, fs)
```

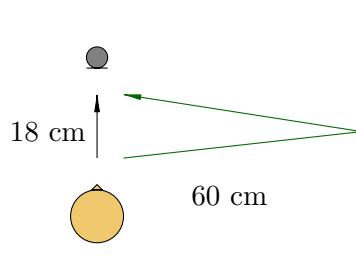
Esa es la implementación del bloque  $M$  para  $M = 3f_s$ . Para implementar el resto de los elementos en el diagrama, debemos tomar una precaución: ¡no podemos sumar `x` y `xdelay` porque tienen distinto tamaño! Así que debemos armar una señal auxiliar, previa al sumador de señales.

```
xlimpia = np.hstack([x, np.zeros(3*fs)])
a = 0.7
y = a*xlimpia + (1-a)*xdelay
sd.play(y, fs)
```

### 2.5.2. Filtro peine

Hay una persona hablando frente a un micrófono junto a una pared muy reflejante. Al micrófono llegan dos ondas: el discurso del tipo, **jy la onda reflejada contra la pared!**

Supongamos que la distancia persona-micrófono es de 18 cm y la distancia persona-pared es de 60 cm. El sonido directo recorre obviamente 18 cm.



El recorrido de la onda reflejada es la suma de los lados del triángulo isósceles del dibujo. Pitágoras nos da la fórmula de la longitud:

$$2 \cdot \sqrt{(60 \text{ cm})^2 + \left(\frac{18 \text{ cm}}{2}\right)^2} \approx 2\sqrt{(60 \text{ cm})^2} = 120 \text{ cm}$$

*La diferencia de distancias implica una diferencia en el tiempo de arribo de las ondas.* Como en condiciones normales de temperatura y presión la rapidez del sonido es aproximadamente 340 m/s, la diferencia de tiempo es

$$\Delta t = \frac{\Delta x}{c} = \frac{120 \text{ cm} - 18 \text{ cm}}{34000 \text{ cm/s}} = 0.003 \text{ s} = 3 \text{ ms.}$$

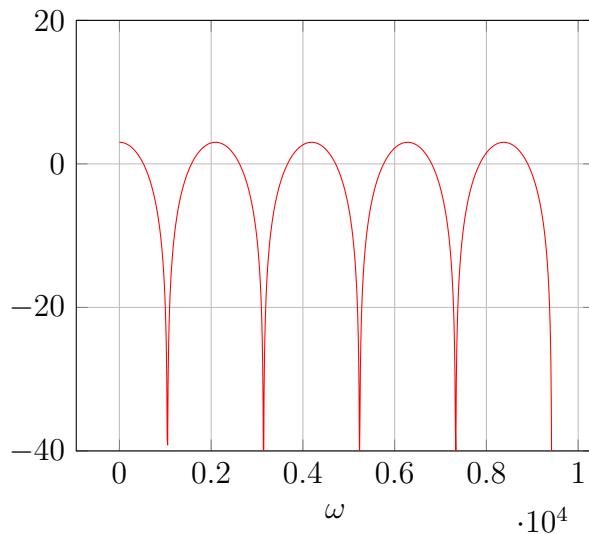
Dado que las ondas arriban al micrófono con una diferencia temporal, se produce **interferencia**. Este fenómeno se pone de manifiesto en el dominio de la frecuencia. Vemos el **comportamiento en frecuencia cuando la señal original es un tono puro con frecuencia variable  $f$**  y pulsación variable  $\omega = 2\pi f$ . La señal reflejada tiene la misma forma, pero arriba al micrófono con un retardo temporal  $\Delta t$ . El sonido directo es entonces  $\cos(2\pi f t) = \cos(\omega t)$ , y el reflejado,  $\cos(\omega(t - \Delta t))$ . Al micrófono llega

$$y(t) = \cos(\omega t) + \cos(\omega(t - \Delta t))$$

con un poco de álgebra, se deduce

$$|y(t)| = \sqrt{2} \{1 + \cos(\omega \Delta t)\}^{\frac{1}{2}} \left| \cos\left(\omega\left(t - \frac{1}{2}\Delta t\right)\right) \right|$$

por lo tanto, la amplitud de  $y(t)$  es una función de la pulsación  $\omega$ , para un determinado  $\Delta t$ . La gráfica de  $A(\omega) = 20 \log \{1 + \cos(\omega \Delta t)\}^{\frac{1}{2}} = 10 \log \{1 + \cos(\omega \Delta t)\}$  es



que pareciera tener como forma de peine (o eso vio alguien, en algún momento) y por eso, al sistema formado por la persona, el micrófono y la pared, se lo denomina justamente **filtro peine**.

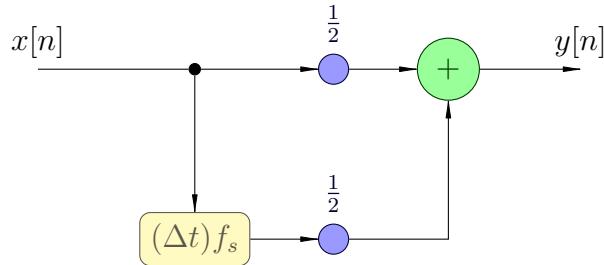
**Implementamos el filtro peine con un retardo.** El número  $M$  de muestras necesarias para tener el retardo de tiempo  $\Delta t$  se calcula con la fórmula de siempre, que incluyo *una vez más* para resaltar su importancia:

$$\boxed{\begin{array}{l} T \cdot f_s \\ \text{segundos} \cdot (\text{muestras/segundo}) = \text{muestras} \end{array}} = N$$

Por ende,  $M = (\Delta t) \cdot f_s$ .

Además, la señal limpia y su versión con retardo deben tener la misma amplitud. Esto lo conseguimos cuando  $a = 1 - a \iff a = \frac{1}{2}$ .

El diagrama de bloques deviene

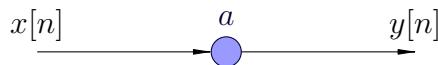


y el código es igual que el del retardo, con  $M = dt * fs$  y  $a = 0.5$ . En el ejemplo de la persona hablando junto a la pared,  $dt = 0.003$ .

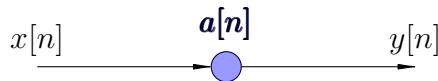
### 2.5.3. Tremolo

El **tremolo** es un **cambio periódico en la amplitud** de la señal.

Modificábamos la amplitud de la señal con un *multiplicador*:



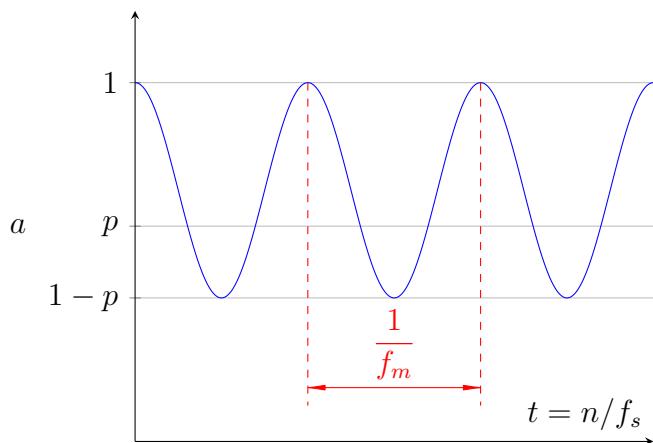
en tal caso,  $y[n] = a \cdot x[n]$ , con  $a$  constante. Para tener *amplitud variable con el tiempo*, tomamos  $a$  dependiente de  $n$ . La salida es una multiplicación de señales  $y[n] = a[n] \cdot x[n]$ , llamada **modulación**. Decimos en tal caso que **la señal a modula a la señal x**.



Como la amplitud varía *periódicamente* con el tiempo, entonces  $a[n]$  es una función periódica de  $n$ . Una posible es la siguiente:

$$a[n] = \left(1 - \frac{1}{2}p\right) + \frac{1}{2}p \cos\left(2\pi f_m \cdot \frac{n}{f_s}\right), \quad 0 \leq p \leq 1$$

con  $p$  la “profundidad” del efecto,  $f_m$  la frecuencia de modulación y  $f_s$  la frecuencia de muestreo de la señal de entrada. Si  $p = 0$ ,  $a[n]$  es constantemente igual a uno. Por ende,  $y[n] = a[n] \cdot x[n] = x[n]$ . No hay efecto. Contrariamente, si  $p = 1$ ,  $a[n] = \frac{1}{2} + \frac{1}{2} \cos\left(2\pi f_m \frac{n}{f_s}\right)$ . El factor varía como mucho entre 0 y 1.



La frecuencia de modulación  $f_m$  indica cuántas veces por segundo aplico el cambio de amplitud completo. Parto de  $a[n] = 1$ . Paso una vez por el mínimo de  $a[n]$ , y vuelvo al máximo. Eso es un ciclo. La cantidad de veces que entra ese ciclo en un segundo es igual a  $f_m$ . Los valores más razonables o “musicales” son  $0.5 \leq f_m \leq 20$ .

Entonces

aplicamos *tremolo* multiplicando por una función periódica de  $n$ .

Lo implementamos así:

```
def tremolo(x, fs, p, fm):
    n = np.arange(x.shape[0]) # vector de tiempo discreto
    a = (1-p/2) + 1/2*p*np.cos(2*np.pi*fm*n/fs)
    y = a * x
    return y
```

### 2.5.4. Distorsión armónica

Hagamos una “disto”. ¿A qué llaman “disto” los músicos? A la **distorsión armónica**, un fenómeno de **alinealidad en amplitud**.

- Es **armónica** porque *agrega componentes en frecuencia*.
- Hay **alinealidad en amplitud** porque *la salida no es función lineal de la entrada*.

Un sistema como  $y[n] = x^2[n]$  modifica la entrada  $x[n] = \cos(2\pi f_0 n/f_s)$  de manera tal que  $y[n] = \cos^2(2\pi f_0 n/f_s) = \frac{1}{2} + \frac{1}{2} \cos(2\pi(2f_0)n/f_s)$ . O sea, de  $f_0$  en la entrada, pasamos a  $2f_0$  en la salida. *La frecuencia se fue al doble*.

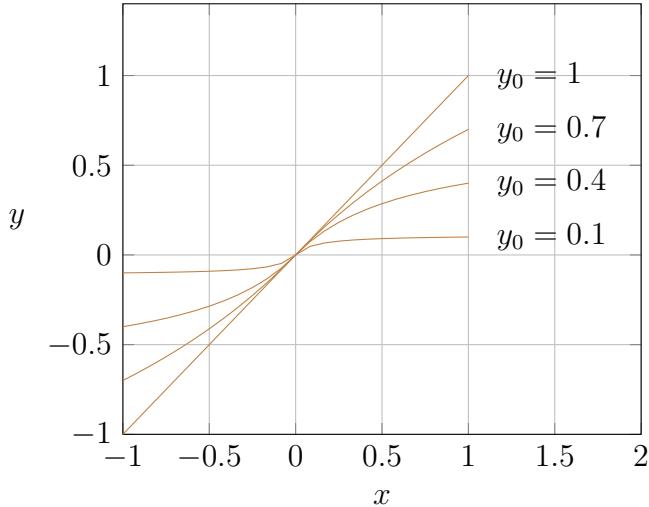
En cambio, un sistema lineal como  $y[n] = \frac{1}{2}x[n]$  produce, para el mismo  $x[n]$  de antes, la salida  $y[n] = \frac{1}{2} \cos(2\pi f_0 n/f_s)$ . ¡No cambia la frecuencia!

**Producimos distorsión armónica si usamos funciones no lineales.**

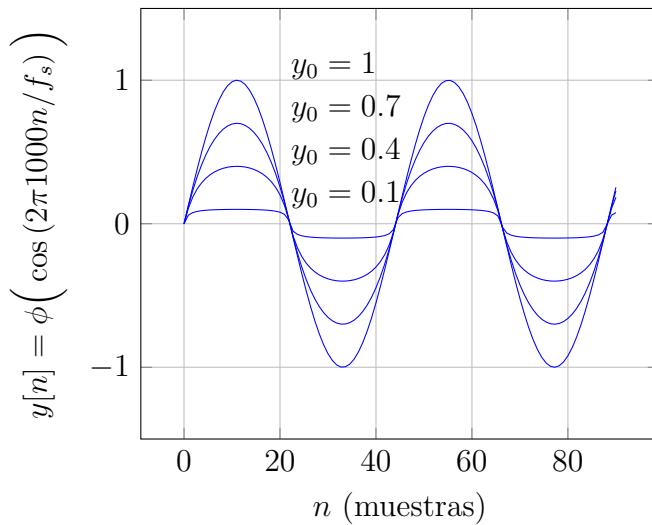
Para un audio  $\mathbf{x}$ ,  $\mathbf{x}^{**2}$  tiene distorsión armónica. Pero suena fea. Una que suena razonablemente bien es:

$$\phi(x) = \frac{y_0 x}{(1 - y_0)|x| + y_0}, \text{ con } 0 < y_0 \leq 1$$

La gráfica  $y = \phi(x)$  para algunos  $y_0$  es:



La función  $\phi$  es impar: la distorsión es simétrica. Cuando la entrada es un tono puro (por ejemplo de 1000 Hz) pasa lo siguiente:



A menor  $y_0$ , mayor distorsión, siempre que  $0 < y_0 \leq 1$ . Fuera de ese rango, las cosas suenan bastante mal.

La implementación de la “disto” en python es muy sencilla:

```
def bosio_disto(x, y0):
    return (y0*x) / ((1-y0)*np.abs(x)+y0)
```

### 2.5.5. Reverberación

Reventamos un petardo en el escenario de un teatro y lo grabamos desde alguna butaca. El resultado es la *respuesta al impulso* de la sala, correspondiente a los puntos de emisión y recepción. Fourier nos dice que *un impulso contiene todas las frecuencias*. Por ende, **la respuesta al impulso contiene las características acústicas de la sala en todas las frecuencias**.

Esto es muy útil. Podemos saber cómo sonaría cualquier señal en la sala, sin necesidad de viajar hasta ella para registrar especialmente la señal. **La operación que “sumerge” la señal en la sala se llama convolución.** Su definición matemática es [1]:

$$y[n] = x[n] * h[n] = \sum_k x[k] h[n-k]$$

En este caso,  $x[n]$  es la señal de entrada (la que quiero “sumergir”),  $h[n]$  es la respuesta al impulso (la grabación del petardo) e  $y[n]$  la señal de salida. Los límites de la sumatoria dependen de cuántas muestras tienen  $x$  y  $h$ .

EJEMPLO: quiero soplar el clarinete en el teatro Roma, pero no tengo guita ni permiso para hacerlo. ¡Buena noticia! Tenemos una respuesta al impulso

del teatro Roma en estéreo. Entonces puedo grabarme con `sounddevice` y convolucionar las señales, para sonar como si lo hubiese grabado allá:

```
import soundfile as sf
from scipy.signal import convolve

h, fs = sf.read('IR.wav') # respuesta al impulso
N = 5 * fs
canales = 1
x = sd.rec(N, fs, canales).ravel()
sd.wait()
y_izq = convolve(x, h[:, 0])
y_der = convolve(x, h[:, 1])
y = np.vstack([y_izq, y_der]).T
y *= np.max(np.abs(x)) / np.max(np.abs(y))
sd.play(y, fs)
```

Como `x` es mono y `h` es estéreo, se hace un pequeño truco:

1. convolucionar `x` con los canales izquierdo y derecho de `h` por separado;
2. concatenar verticalmente las señales mono generadas por convolución;
3. pasar las filas a columnas, y las columnas a filas. Esto se llama *transposición* de la matriz, y se realiza con el operador `.T`.

Después, se multiplica `y` por `np.max(np.abs(x)) / np.max(np.abs(y))` para que, de esta forma, `x` e `y` tengan la misma excursión en amplitud.

`numpy` también tiene una función `convolve`, pero es mucho más lenta. `scipy` es una biblioteca de los mismos creadores de `numpy` que trae cargada la convolución, entre muchísimas cosas más, con la diferencia de que la función de `scipy` está optimizada. La convolución optimizada usa un conjunto de algoritmos que conforman la renombrada y famosa *transformada rápida de Fourier*, de amplio uso en procesamiento digital de señales. En este libro, la transformada de Fourier se presenta en el Capítulo 4.

## Referencias

- [1] A. V. Oppenheim y A. S. Willsky, *Signals and Systems*, 2.<sup>a</sup> ed. United States of America: Prentice Hall, 1997 (vid. págs. 46, 52).

# Capítulo 3

## Gráficos y operaciones numéricas

Tanto en ingeniería, como en física, economía, sociología o en cualquier disciplina que aspire a ser científica, tarde o temprano llegaremos a una situación en la que debamos manipular conjuntos de números. Estos pueden resultar de mediciones, estimaciones, predicciones, grabaciones y tantas otras posibilidades, y pueden tener infinidad de utilidades.

Supongamos que se realiza una medición de un parámetro en función de otro. Un registro de audio muestra **la amplitud de una onda sonora en función del tiempo**, pero podríamos hablar del valor del dólar en pesos en función del día del mes, o de la edad de Mirtha Legrand en función de cuánto tarda en maquillarse por día, entre tantas posibilidades. En general, se tendrá un conjunto de datos tomados como variable independiente (el tiempo, el día del mes, los millones) y otro conjunto de datos tomado como variable dependiente (la amplitud de la onda, el dólar, las cirugías). A lo largo de este capítulo usaremos los siguientes conjuntos para los ejemplos:

```
datos_x = np.array([1, 1.3, 1.7, 1.9, 2.1, 2.35, 2.55, 2.75,  
                    2.8, 3])  
datos_y = np.array([1.1, 1.3, 1.75, 2, 2, 2.3, 2.45, 2.67, 2.9,  
                    3.1])
```

Aquí `datos_x` es la variable independiente, en tanto `datos_y` es la variable dependiente. Ambos son *arrays* de *numpy* y, naturalmente, tienen el mismo largo. Esto es razonable: por cada valor de la variable independiente debe haber uno de la variable dependiente.

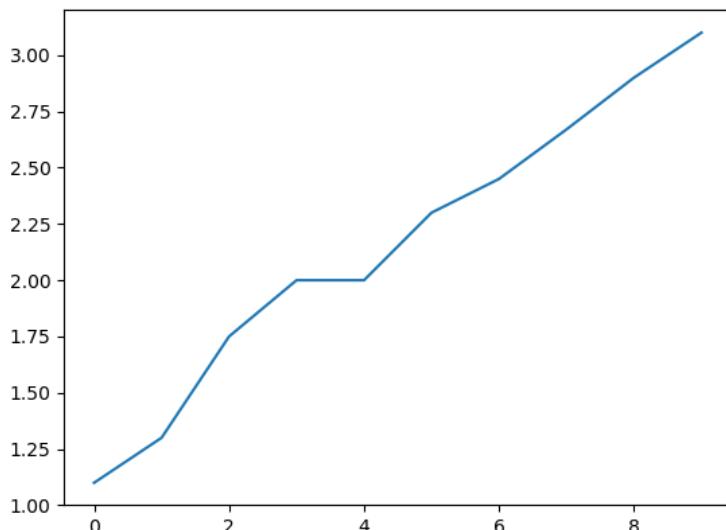
Para graficar el conjunto de datos, debemos importar el módulo de Python dedicado a esta función. El más elemental y conocido es el módulo `matplotlib.pyplot`, que podemos importar de dos maneras distintas:

```
from matplotlib import pyplot as plt
import matplotlib.pyplot as plt
```

La función dentro de este módulo que grafica un conjunto de datos en función de otro es, en su uso más elemental, la función `plt.plot`, que puede recibir *muchísimos* argumentos. Si le pasamos un solo conjunto de datos, lo grafica en función del número de muestra en el vector:

```
plt.plot(datos_y)
```

Esta sentencia devuelve este gráfico:

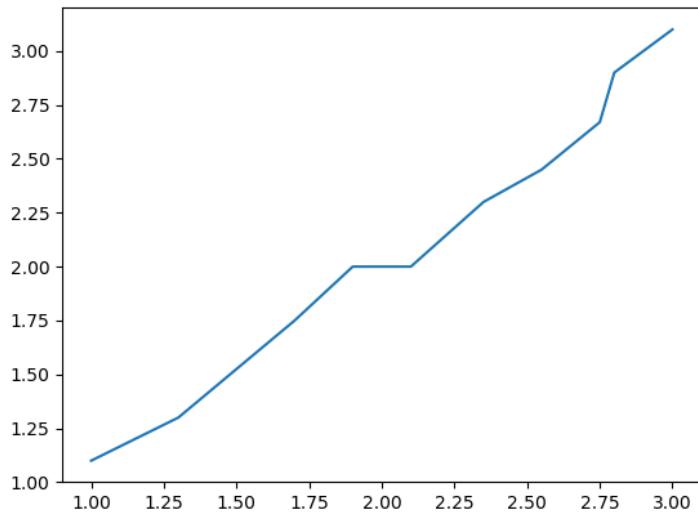


Este gráfico contiene los valores del *eje y* en función de la posición que ocupan. El valor de `datos_x` en la posición 0, o sea, `datos_x[0]`, es 1.1 y el valor en la posición 8, o sea, `datos_x[8]`, es 2.9. Por cierto, esto no hace falta mientras trabajemos en la consola de Spyder, pero fuera de ella, hace falta incluir la línea `plt.show()` al final de cualquier código para un gráfico. La idea es que `plt.plot` confecciona el gráfico con los datos introducidos, pero no lo hace visible.

Prosiguiendo, si queremos que un gráfico dé los valores en el *eje x* contra los del *eje y*, simplemente ponemos ambos ejes (el independiente primero) como argumentos:

```
plt.plot(datos_x, datos_y)
```

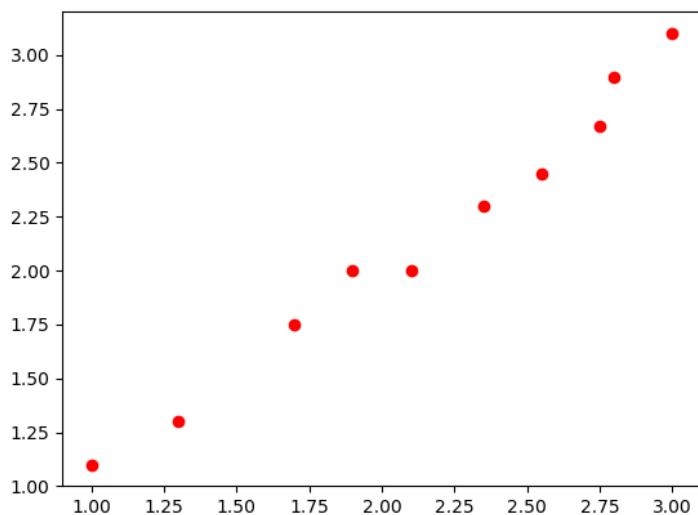
Esta sentencia nos da el siguiente gráfico:



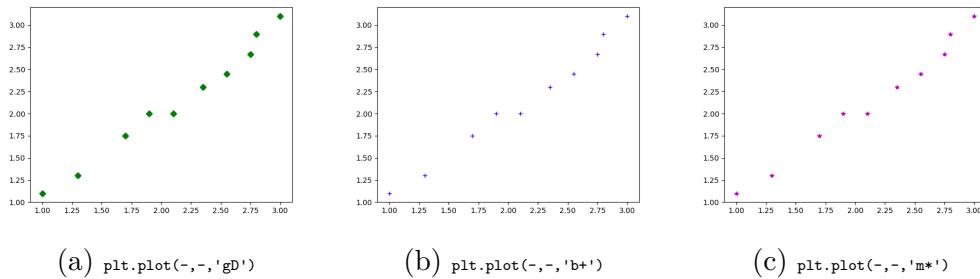
Ahora, el *eje x* contiene los valores del array `datos_x` y el *eje y* los del array `datos_y`.

Nótese que Python, por defecto, interpola con rectas los puntos que le damos, y usa siempre el mismo color. Podemos decirle que marque los puntos con otro marcador y sin interpolar:

```
plt.plot(datos_x, datos_y, 'ro')
```

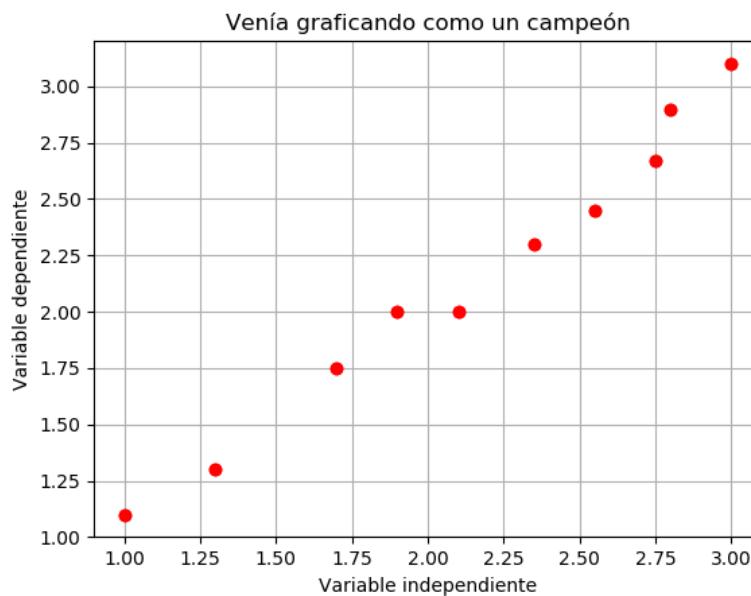


Aquí le hemos dicho a la función que grafique `datos_x` contra `datos_y`, pero que lo haga con el color rojo (de ahí el '`r`') y en círculos (de ahí el '`o`'). Tenemos muchas opciones de colores y marcadores para poner. A continuación se ven algunos comunes, pero pueden consultarlos todos en [https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.plot.html](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html)



Por supuesto, todos estos gráficos todavía son una porquería total. A continuación numeramos la figura, le agregamos nombre a los ejes y le ponemos título al gráfico. Además, agregamos una grilla cuadriculada:

```
plt.figure(1)
plt.plot(datos_x, datos_y, 'ro')
plt.xlabel('Variable independiente')
plt.ylabel('Variable dependiente')
plt.title('Venía graficando como un campeón')
plt.grid()
```



Fachero. La sentencia `plt.figure(1)` indica que los `plots` que sigan se ejecutarán en la figura 1. Si fuera a hacer muchos gráficos, estos podrían hacerse en los mismos ejes (lo cual es interesante si quiero comparar funciones), o, si los coloco en figuras diferentes (con `plt.figure(2)`) se armarán en ejes distintos. En lo que sigue de este capítulo vamos a hacer algunas cosas más con gráficos. Lo que deba ser explicado se explicará, pero todo debería ser razonablemente obvio.

## 3.1. Interpolaciones y ajustes

Cuando tenemos un conjunto de puntos sueltos, lo primero que querríamos tener es la función que los genera, esto es, una variable dependiente en función de la independiente. Esto porque una función puede ser derivada, maximizada, integrada y conocida hasta el hartazgo. Sin embargo, la función que generó los puntos no es conocida, y lo que haremos es **estimar otra función que la aproxime** a partir del conjunto de puntos dato. Considerando que vamos a hacer una estimación y buscaremos la función más sencilla posible, parece razonable empezar por buscar polinomios, que son las funciones más sencillas que manejamos.

### 3.1.1. Polinomios de Lagrange

*(Disclaimer: esto difícilmente resulte útil para alguien, hay mucho mejores recursos, pero es una estrategia fundamental y tal vez vale la pena conocerla y programarla es un desafío interesante. Es, por cierto, lo más difícil del capítulo).*

La primera técnica que utilizaremos es la **interpolación**, esto es, armar un polinomio que pase por todos los puntos dato. Si uno tiene un conjunto de dos puntos, entonces estos son unidos por una única recta, esto es, un polinomio de grado 1. Si tenemos tres puntos, puedo obtener una única parábola  $ax^2 + bx + c$  (un polinomio de grado 2) que pasa por ellos. En general, **un único polinomio de grado  $n$  queda determinado por  $n + 1$  puntos**.

Hace muchos años, Lagrange se inventó un método muy ingenioso para hallar un polinomio que pasa por  $n$  puntos. Si tengo, por ejemplo, tres puntos  $(x_0, y_0)$ ,  $(x_1, y_1)$  y  $(x_2, y_2)$  este polinomio de grado 2 se arma así:

$$L(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}y_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}y_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}y_2$$

No es tan monstruoso como parece: tiene un término por cada punto. La idea es que si  $x = x_0$  en el primer término se simplifica todo y queda  $y_0$ , en los

otros dos términos queda 0. Así,  $L(x_0) = y_0$ , el polinomio pasa por donde tiene que pasar. Lo mismo pasa con los otros puntos, y el polinomio está buenísimo. para un polinomio de grado  $n - 1$  (o sea,  $n$  puntos) queda:

$$\sum_{j=0}^{n-1} u_j(x) y_j$$

donde:

$$u_j(x) = \prod_{\substack{i=0 \\ i \neq j}}^{n-1} \frac{x - x_i}{x_j - x_i}$$

Esto de la sumatoria de una productoria no es tan fácil de implementar, pero tampoco es imposible. El código a continuación da el polinomio de Lagrange construido en base a los arrays `datos_x` y `datos_y`, evaluado en un vector cualquiera `x`. Esto nos permite usar lo que el polinomio valía en los puntos medidos en otros puntos que no fueron medidos:

```
def lagrange(x, datos_x, datos_y):
    """Implementa el P. de Lagrange que interpola datos_x y
    datos_y.

    datos_x: array de numpy
    datos_y: np.array
    x: variable independiente (un array de tiempo, p. ej.)
    asume len(datos_x) == len(datos_y)
    """

    puntos = len(datos_x)
    polinomio = np.zeros(len(x))
    for j in range(0, puntos):
        u_j = np.ones(len(x))
        for i in range(0, puntos):
            if j != i:
                u_j *= (x - datos_x[i]) / (datos_x[j] - datos_x[i])
        polinomio += u_j * datos_y[j]
    return polinomio
```

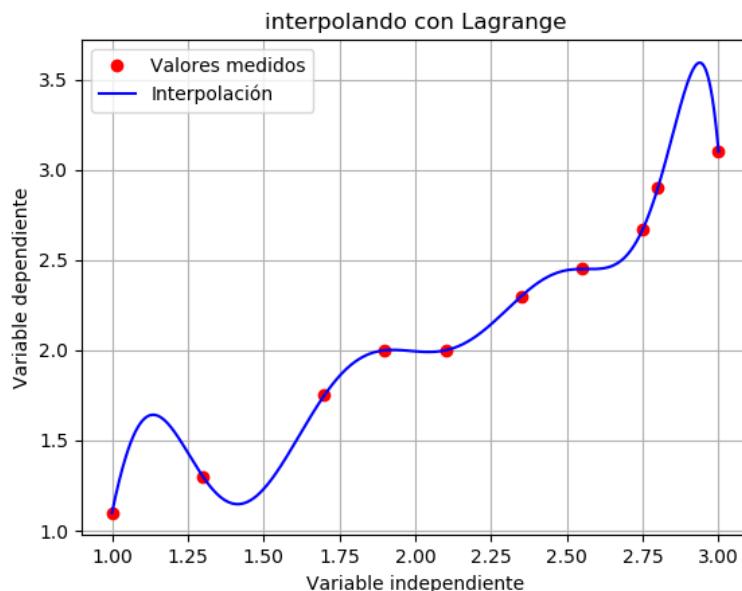
Veamos cómo queda el gráfico de los datos junto al polinomio de Lagrange que los interpola:

```
datos_x = np.array([1, 1.3, 1.7, 1.9, 2.1, 2.35, 2.55, 2.75,
                    2.8, 3])
```

```

datos_y = np.array([1.1, 1.3, 1.75, 2, 2, 2.3, 2.45, 2.67, 2.9,
                   3.1])
x = np.linspace(1, 3, 1000) # mil puntos entre 1 y 3
plt.figure(1)
plt.plot(datos_x, datos_y, 'ro', label='Valores medidos')
plt.plot(x, lagrange(x, datos_x, datos_y), 'b',
          label='Interpolación')
plt.xlabel('Variable independiente')
plt.ylabel('Variable dependiente')
plt.title('interpolando con Lagrange')
plt.grid()
plt.legend(loc='upper left')

```



Vemos que se genera un polinomio bastante raro que pasa por todos los puntos. Las desventajas de obtener la función que estima los datos de este modo son varias:

1. Es carísimo computacionalmente, pues si tengo 15 puntos, Lagrange me da un polinomio de grado 14.
2. No es razonable a nivel intuitivo, pues si yo mido 15 puntos no es porque creo que vayan a ser caracterizados por un polinomio de grado 14, uno tiende a buscar polinomios de grados bajos.

3. Los polinomios de grado alto generan *lobulaciones*, como se ve en la figura. Esos lóbulos al principio y al final no se ajustan al conjunto de datos, y aparecen porque el polinomio de grado alto tiene ese tipo de fluctuaciones.

Veamos, entonces, una mejor idea para estimar conjuntos de puntos.

### 3.1.2. Ajuste por cuadrados mínimos

El enfoque anterior probó no ser lo mejor del mundo. Ahora nos proponemos hacer algo más cercano a lo mejor del mundo. Cuando yo tengo un conjunto de datos, no estoy ciego antes de hacer algún tipo de ajuste o demás: puedo ver si parece una recta, una parábola o algo así, o puedo intentar forzarlo a que se parezca. En este nuevo enfoque, nos proponemos, dado un conjunto de puntos, **hallar el polinomio de grado  $n$  que mejor approxima ese conjunto de puntos**. Una cosa a tener en cuenta es que este polinomio, aunque sea el que mejor approxima al conjunto en promedio, no tiene por qué pasar por cada uno de los puntos. Esa es la diferencia entre interpolar (pasar por los puntos) y ajustar (hallar la función que más se adecúa al conjunto). Entonces, supongamos que tengo  $k$  puntos  $(x_0, y_0), (x_1, y_1), \dots, (x_{k-1}, y_{k-1})$ . Armo un polinomio de grado  $n$ ,  $p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$ . Mi objetivo es hallar los coeficientes del polinomio,  $(a_0, a_1, \dots, a_n)$ . Para eso, evalúo cada punto en el polinomio:

$$\begin{aligned} y_0 &= a_0 + a_1x_0 + a_2x_0^2 + \dots + a_nx_0^n \\ y_1 &= a_0 + a_1x_1 + a_2x_1^2 + \dots + a_nx_1^n \\ &\vdots \\ y_k &= a_0 + a_1x_k + a_2x_k^2 + \dots + a_nx_k^n \end{aligned}$$

Esto se puede escribir como producto de matrices:

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_k \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \ddots & & \vdots \\ 1 & x_k & x_k^2 & \cdots & x_k^n \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} \quad (3.1)$$

O más compacto:

$$\mathbf{y} = \mathbf{P} \mathbf{a} \quad (3.2)$$

Quiero despejar la columna de coeficientes  $\mathbf{a}$ , y uno diría alegramente que invierta  $\mathbf{P}$  y multiplique. Sin embargo, si lo piensan un poco,  $\mathbf{P}$  es de  $k$  filas y  $n$  columnas, no es cuadrada. Para poder invertirla, uso un truco:

premultiplico por  $\mathbf{P}^T$ , y como  $\mathbf{P}^T\mathbf{P}$  siempre es cuadrada, invierto esa para hallar  $\mathbf{a}$

$$\mathbf{P}^T\mathbf{y} = \mathbf{P}^T\mathbf{P}\mathbf{a}$$

luego

$$(\mathbf{P}^T\mathbf{P})^{-1}\mathbf{P}^T\mathbf{y} = \mathbf{a}$$

No hemos demostrado (ni lo haremos) por qué de hecho este es el mejor polinomio para aproximar los datos. Es tema de álgebra II, y no nos interesa. A continuación implementamos la función `cuad_min` en Python, que recibe como argumentos dos conjuntos de datos  $y$  un grado  $n$  para el polinomio, y da los coeficientes del mejor polinomio de grado  $n$  para ajustar esos datos:

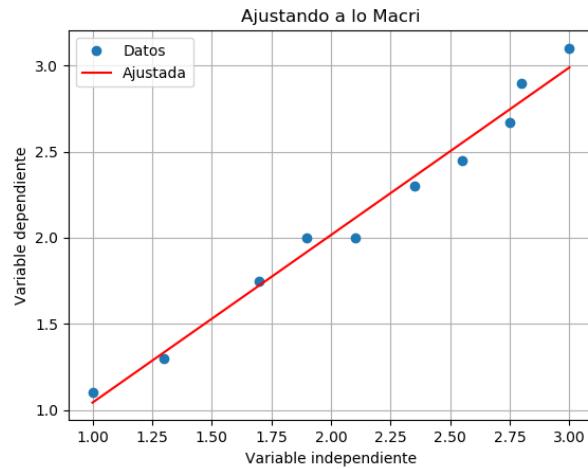
```
from numpy.linalg import inv

def cuad_min(datos_x, datos_y, n):
    """datos_x y datos_y son arrays.
    n es el grado del polinomio de ajuste
    """
    cols = n + 1
    fils = len(datos_x)
    A = np.zeros([fils, cols])
    for i in range(0, fils):
        for j in range(0, cols):
            A[i, j] = (datos_x[i])**j
    coefs = inv((A.T) @ A) @ (A.T) @ datos_y
    coefs = coefs[::-1]
    return coefs
```

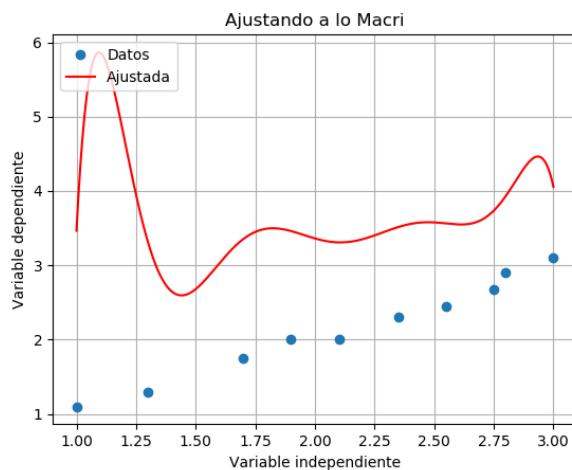
Analicemos el comportamiento del ajuste. Para no andar armando el polinomio con los coeficientes, usamos la función `polyval` de `numpy`, que evalúa un vector en un polinomio conociendo sus coeficientes

```
# Regresión lineal, grado 1
coefs = cuad_min(datos_x, datos_y, 1)
interpolada = np.polyval(coefs, x)
plt.figure(2)
plt.plot(datos_x, datos_y, 'o', label='Datos')
plt.plot(x, interpolada, 'r', linewidth=1.5, label='Ajustada')
plt.xlabel('Variable independiente')
```

```
plt.ylabel('Variable dependiente')
plt.title('Ajustando a lo Macri')
plt.grid()
plt.legend(loc='upper left')
```



**IMPORTANTE:** Como verán, el ajuste es mucho más razonable que la interpolación en un caso como este. El ajuste prueba, por si no se veía muy claro, que los datos se adecúan a una función lineal, y eso es información muy valiosa acerca de lo que sea que estuviésemos midiendo. De hecho, teniendo los coeficientes, tenemos (con cierto grado de error por la medición) *a qué* función lineal se adecúan. Algún genio capaz pensará: ‘entonces yo soy todopoderoso y Python hace las cuentas a toda velocidad. Le voy a dar un polinomio de grado 8 y va a ser más preciso’...



Esta porquería que quedó se debe a un fenómeno llamado *overfitting*, le exigí al polinomio un grado tan alto que las lobulaciones lo volvieron loco y la estimación se vuelve muy mala. Por eso que no hay que ponerse ambiciosos. En la práctica es raro ver ajustes de curvas con grados mayores a  $n = 3$  o  $n = 4$ .

**Nota:** *numpy* contiene la función `polyfit`. Hace lo mismo que `cuad_min`.

### 3.1.3. Ajuste de funciones potenciales

Esto es muy, muy fácil pero también es lindo e ingenioso, y sirve cuando vemos que nuestros datos parecen ajustarse a una función potencial. Una función potencial tiene la forma:

$$f(x) = \beta x^\alpha$$

Si tomamos logaritmo de los dos lados queda una función que depende linealmente de una nueva variable  $x'$ :

$$\log(f(x)) = \log(\beta) + \alpha \log(x) = c_0 + c_1 x'$$

Esto nos permite hacer algo muy copado: en vez de pensar cómo changos ajusto la función, tomo el logaritmo de los datos y les ajusto una recta por cuadrados mínimos. El primer coeficiente es el logaritmo de  $\beta$ , y el segundo es  $\alpha$ . Entonces, podemos despejar  $\beta = e^{c_0}$  y ya hallamos  $\alpha$  y  $\beta$  para reconstruir la función. Esto lo implementamos en una función sencillísima:

```
def ajuste_potencial(x, y):
    alfa, c0 = cuad_min(np.log(x), np.log(y), 1)
    beta = np.exp(c0)
    return alfa, beta
```

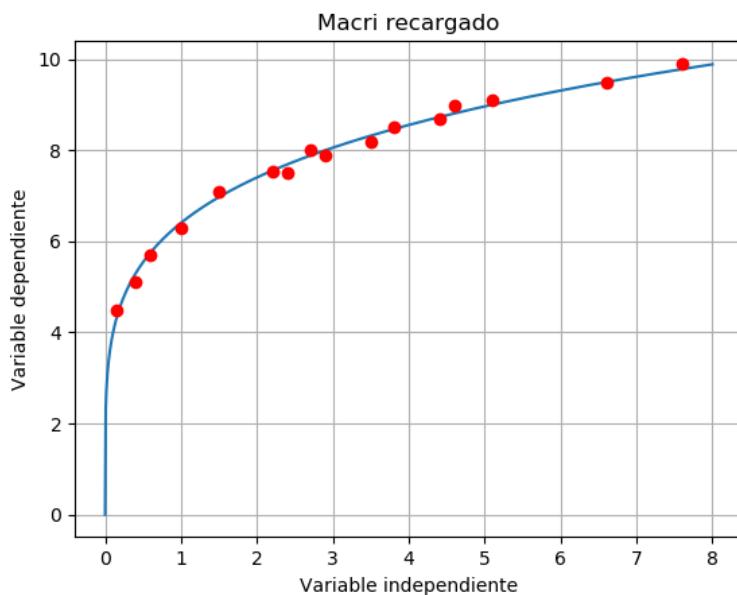
Y esto lo graficamos usando dos conjuntos de datos que inventé para ello:

```
datosX_pot = np.array([0.15, 0.4, 0.6, 1.01, 1.5, 2.2, 2.4,
                      2.7, 2.9, 3.5, 3.8, 4.4, 4.6, 5.1,
                      6.6, 7.6])
datosY_pot = np.array([4.5, 5.1, 5.7, 6.3, 7.1, 7.55, 7.5, 8,
                      7.9, 8.2, 8.5, 8.7, 9, 9.1, 9.5, 9.9])
coef_pot = ajuste_potencial(datosX_pot, datosY_pot)
x = np.linspace(0, 8, 1000)
ajustados = coef_pot[1] * (x**coef_pot[0])
plt.figure(3)
plt.plot(x, ajustados)
```

```

plt.plot(datosX_pot, datosY_pot, 'ro')
plt.xlabel('Variable independiente')
plt.ylabel('Variable dependiente')
plt.title('Macri recargado')
plt.grid()

```



## 3.2. Suavizado de datos

A menudo recibimos largas listas de datos con rápidas fluctuaciones y que, sin embargo, muestran un comportamiento homogéneo. Esto pasa, por ejemplo, con las respuestas al impulso en audio, o con las fluctuaciones del bitcoin u otras criptomonedas respecto a otras monedas. En esos casos, es difícil imaginar un polinomio que interpole las curvas, pero queremos no obstante suavizar esas fluctuaciones rápidas. Hay muchas técnicas para hacer esto, y aquí veremos la más elemental y conocida, y a partir de ella una más novedosa que ya viene implementada en Python.

### 3.2.1. Filtro de media móvil

Consideremos, a fin de trabajar con un ejemplo, que se tiene una señal senoidal afectada por ruido. Podemos ver el ruido como valores aleatorios que se suman a la señal, y en Python podría armarse así:

```

tiempo = np.linspace(-10, 10, 500)
signal = np.sin(2*np.pi*400*tiempo) + np.random.uniform(-0.2,
                                                       0.2, len(tiempo))

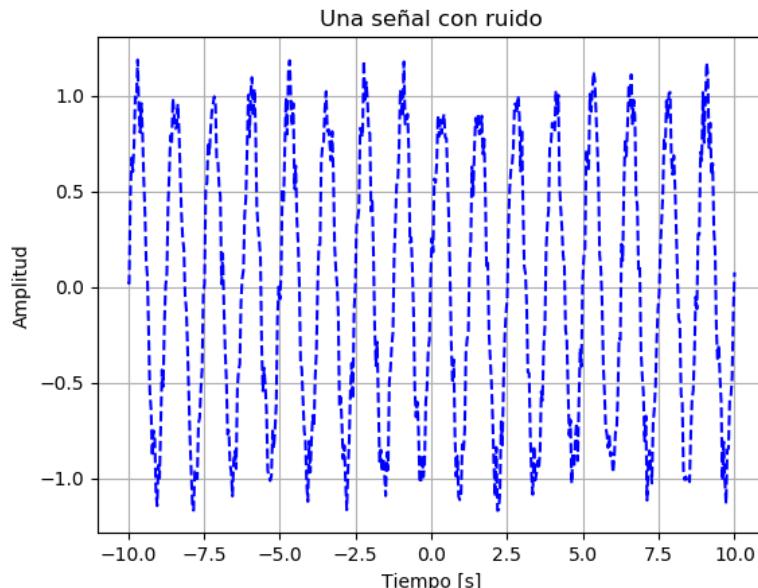
```

Aquí hemos tomado valores aleatorios entre  $-0.2$  y  $0.2$  de una distribución uniforme usando el módulo `np.random`. Esto se ve así:

```

plt.figure(4)
plt.plot(tiempo, signal, 'b--')
plt.xlabel('Tiempo [s]')
plt.ylabel('Amplitud')
plt.title('Una señal con ruido')
plt.grid()

```

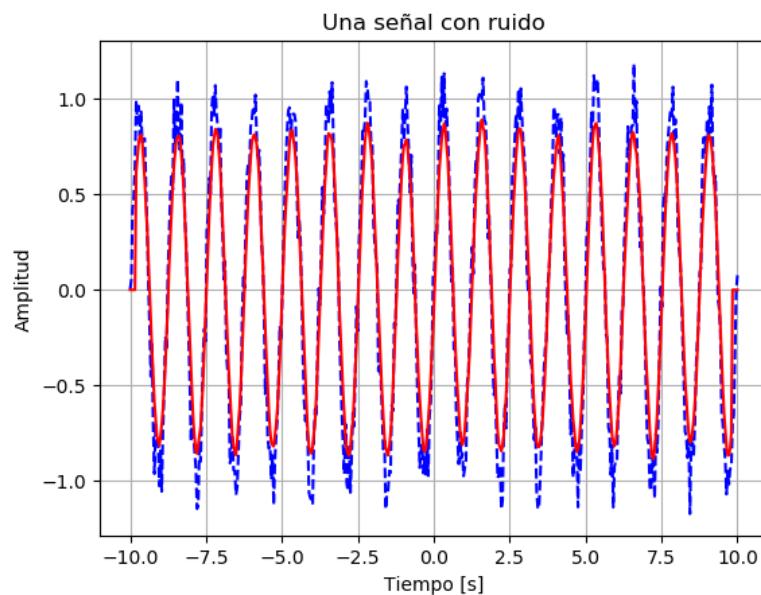


Una forma clásica de suavizar una señal es la siguiente: tomamos un ancho de ventana cualquiera, y ponemos esa ventana al principio de la señal. Ahora, **calculamos el valor promedio de la señal en esa ventana**. Tenemos un valor que representa aproximadamente el valor de la señal en esa ventana. Lo que se hace ahora es barrer la ventana muestra por muestra hasta el final, y por cada posición de la ventana calculamos el valor medio, obteniendo así una nueva señal hecha de los promedios en la ventana.

**Muy importante:** notar que, si la señal tiene 500 muestras y la ventana es de 30 muestras, entonces la señal suavizada tiene 470 muestras, pues la primera ventana va en las primeras 30 muestras y sólo puede barrerse 470

veces hasta llegar al final. Además, por esa misma razón la ventana no está desde “el principio” de la señal, y tiene una suerte de *delay* respecto a la señal. El delay es siempre igual a la mitad del ancho de la ventana, así que podemos intentar corregir ese delay agregando 15 muestras al principio y 15 al final. Implementamos entonces un filtro de media móvil con una ventana de 10 muestras y graficamos la señal con ruido y la suavizada:

```
ventana = 10
suave = np.zeros(len(signal)-ventana)
for i in range(0, len(signal)-ventana):
    suave[i] = np.mean(signal[i:i+ventana])
# Agregamos ceros para compensar el delay
suave = np.hstack([np.zeros(ventana//2), suave,
                   np.zeros(ventana//2)])
plt.figure(4)
plt.plot(tiempo, signal, 'b--')
plt.xlabel('Tiempo [s]')
plt.ylabel('Amplitud')
plt.title('Una señal con ruido')
plt.plot(tiempo, suave, 'r')
plt.grid()
```



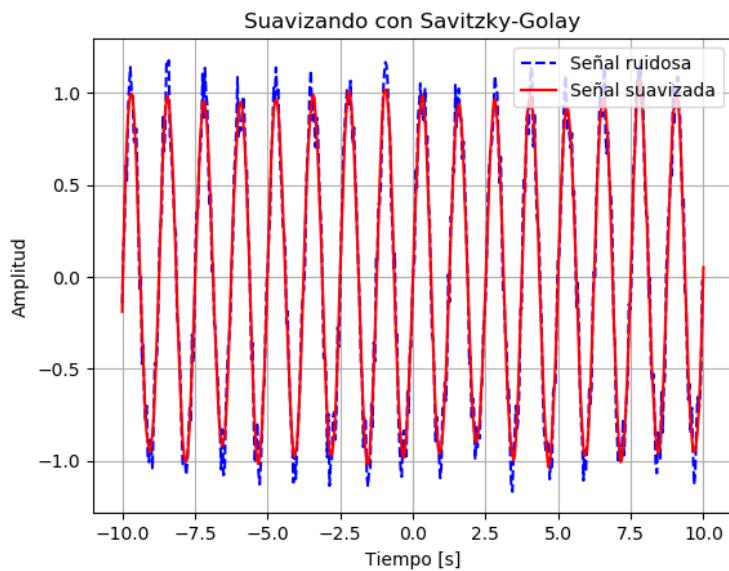
Se ve que ha reconstruido un seno bastante bien. Para el que le interese, el filtro de media móvil elimina las fluctuaciones rápidas de la señal, que son

causadas por las frecuencias altas. Si el filtro de media móvil las elimina, entonces es un filtro pasa-bajos.

### 3.2.2. Filtro de Savitzky-Golay

El filtro de Savitzky-Golay es un filtro más moderno y es bastante ingenioso. Es una mezcla de dos cosas que vimos en este capítulo, la media móvil y el ajuste por cuadrados mínimos. Lo que se hace es tomar ventanas **con un número impar** de muestras, y se obtiene el polinomio de grado  $n$  que mejor aproxima la señal en esa ventana. Después, nos quedamos con el valor a la mitad de ese polinomio sobre la ventana. De esta manera, nos quedamos con un valor por ventana sólo que está hallado de una forma menos grosera, si se quiere, que el valor medio. No hace falta implementarlo: el filtro de Savitzky-Golay está implementado en el módulo `scipy.signal`, sólo hay que pasarle la señal, el tamaño impar de la ventana y el grado del polinomio:

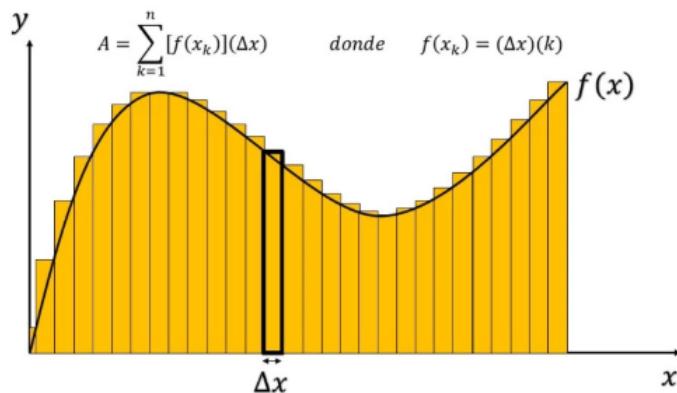
```
from scipy.signal import savgol_filter
filtrada = savgol_filter(signal, 17, 3)
plt.figure(5)
plt.plot(tiempo, signal, 'b--', label='Señal ruidosa')
plt.grid()
plt.plot(tiempo, filtrada, 'r', label='Señal suavizada')
plt.xlabel('Tiempo [s]')
plt.ylabel('Amplitud')
plt.title('Suavizando con Savitzky-Golay')
```



## 3.3. Área y área acumulada

### 3.3.1. Cálculo de áreas

A menudo necesitamos calcular el área de una señal, como si fuera continua, ya que nos representa la energía (o algo parecido) en una señal. El problema es que no podemos calcular una integral, así que tenemos que hacer otra artimaña. La más precaria es la que vemos cuando introducimos la integral definida: tomar rectángulos entre muestra y muestra.



Entonces, implementamos una función en Python para calcular el área:

```
def area_rectangulos(datos_x, datos_y):
    area = 0
    delta_x = (datos_x[-1]-datos_x[0]) / (len(datos_x)-1)
    for i in range(0, len(datos_x)):
        area += delta_x * datos_y[i]
    return area
```

Aquí hemos calculado el paso `delta_x` dividiendo el ancho del intervalo por la cantidad de muestras que contiene. Por supuesto, a menos paso y más rectángulos, menos error vamos a tener. Otra forma de mejorar esto es el **método de los trapecios**, donde en vez de usar rectángulos, unimos cada muestra en el eje *y* por una recta, quedando por calcular el área de un trapecio. Esto es casi igual para implementar, recordando que el área de un trapecio es  $(b_{\text{menor}} + b_{\text{mayor}})h/2$ :

```
def area_trapézios(datos_x, datos_y):
    area = 0
    h = (datos_x[-1]-datos_x[0]) / (len(datos_x)-1)
    for i in range(0, len(datos_y)-1):
```

```

area += (datos_y[i]+datos_y[i+1]) * h / 2
return area

```

Este método está implementado en la función `numpy.trapz`

### 3.3.2. Área acumulada

Ahora bien, podemos calcular el área debajo de una lista de datos, pero nos está faltando un método para armar una función área: cuando trabajamos integrales, no sólo nos interesa calcular el número del área bajo la función, sino obtener la función primitiva. Para hacer eso en variables numéricas, lo que en general se hace es trabajar con el área acumulada. Así, para hallar el área entre 0 y  $x$  de la función  $f(t)$ , hacemos:

$$F(x) = \int_0^x f(t) dt$$

Ahora, podemos implementar esta función en Python haciendo un vector como este:

```

vector_area = [area(datos[0:1]), area(datos[0:2]), ...,
               area(datos[0:N])]

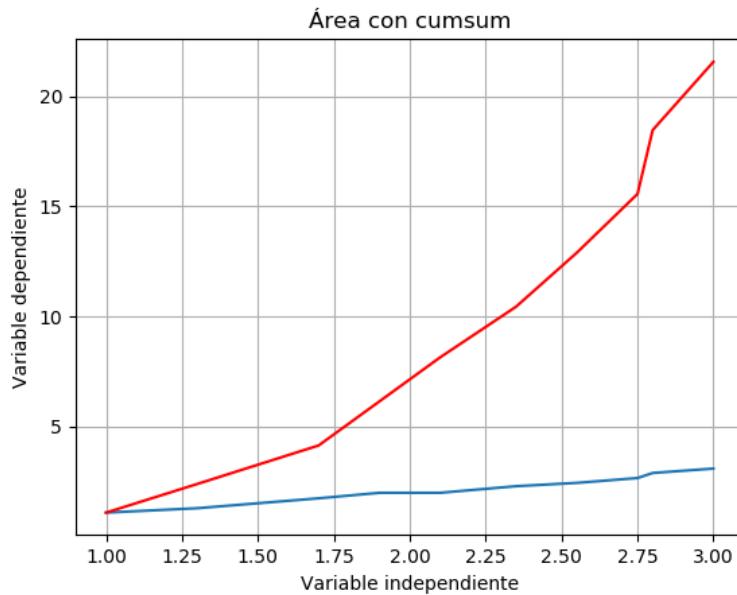
```

Aquí `area` puede ser cualquiera de las funciones que armamos, y `datos` es cualquier vector de datos. La última componente del `vector_area` es el área bajo todo el conjunto. `numpy` trae implementada una función que hace esto, la función `np.cumsum`, que recibe un vector de datos y hace el área acumulada a cada muestra con el método de los rectángulos. Así, podemos probarlo con nuestros conjuntos `datos_x` y `datos_y`, que claramente formaban algo parecido a la recta  $y = x$ , por lo que esperaríamos que salga una especie de parábola:

```

vector_area = np.cumsum(datos_y)
plt.figure(6)
plt.plot(datos_x, datos_y, label='Función original')
plt.plot(datos_x, vector_area, 'r', label='Función área')
plt.xlabel('Variable independiente')
plt.ylabel('Variable dependiente')
plt.title('Área con cumsum')
plt.grid()

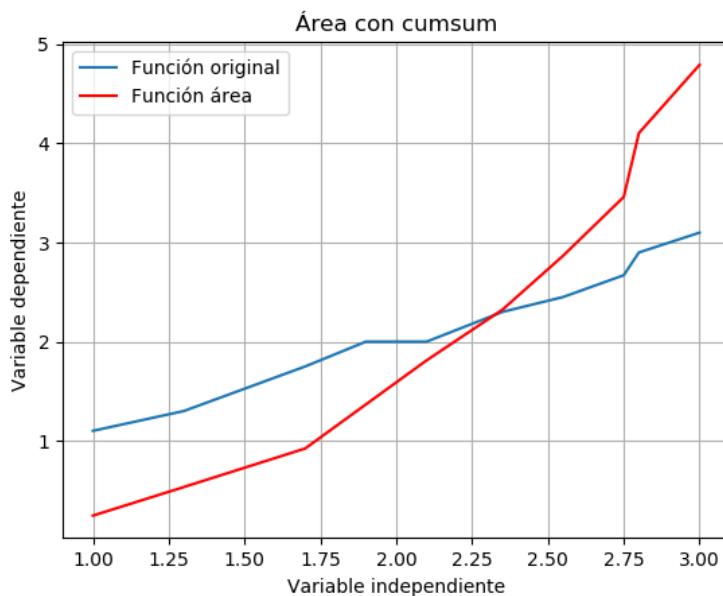
```



Bueno, eso está bastante feo. La escala es engañosa pero la recta ciertamente va de  $x = y = 1$  a  $x = y = 3$ . El problema es que la parábola, si es  $y = x^2/2$ , ¡no debería llegar hasta veintipico!

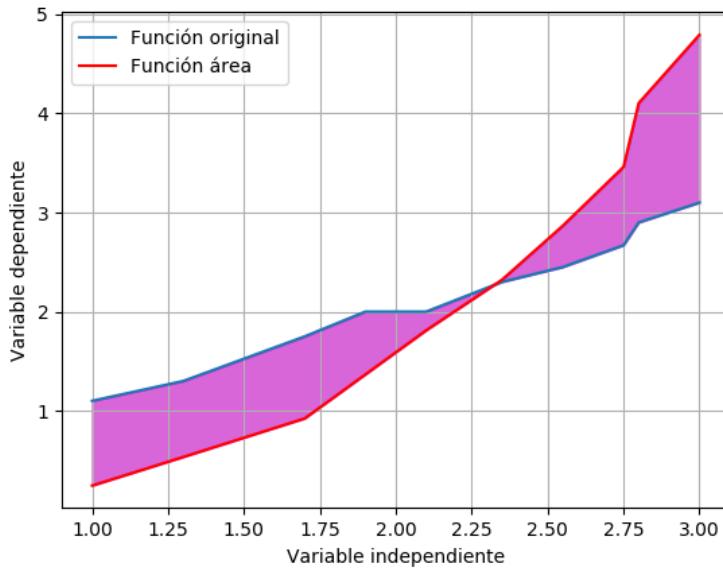
Aquí el error es que `np.cumsum` no tiene por qué saber de dónde a donde va el eje  $x$ , y asume que tiene pasos de 1 en 1, de modo que calculó el área bajo una recta que no es la que queremos. Para que esto funcione, debemos multiplicar `vector_area` por el paso entre muestra y muestra en el vector `datos_x`, para así tener en cuenta el ancho de ese eje, como habíamos hecho en nuestras funciones de área:

```
delta_x = (datos_x[-1]-datos_x[0]) / (len(datos_x)-1)
vector_area = np.cumsum(datos_y) * delta_x
plt.figure(6)
plt.plot(datos_x, datos_y, label='Función original')
plt.plot(datos_x, vector_area, 'r', label='Función área')
plt.xlabel('Variable independiente')
plt.ylabel('Variable dependiente')
plt.title('Área con cumsum')
plt.grid()
```



Genial, ahora se parece más a lo que tiene que ser. Para el que le interese, un buen ejercicio sería programar la función `cumtrapz(datos_x, datos_y)`, que no viene con `numpy` y sería más precisa que `np.cumsum`, que usa el método de los rectángulos. La función `cumtrapz` sí está implementada en Matlab, y muchos se burlan de ello porque parece el nombre de un canal porno. En fin, ya que estamos en plan de dispersarnos, `matplotlib` incluye una función muy pavota para graficar área entre dos curvas, la función `plt.fill_between(datos_x, datos_y1, datos_y2)`, que pinta el área entre `datos_y1` y `datos_y2`. Si no le ponemos nada en `datos_y2`, simplemente hace el área entre la curva `datos_y1` y el eje *x*:

```
plt.figure(7)
plt.plot(datos_x, datos_y, label='Función original')
plt.plot(datos_x, vector_area, 'r', label='Función área')
plt.fill_between(datos_x, datos_y, vector_area, color='m',
                 alpha=0.6)
plt.xlabel('Variable independiente')
plt.ylabel('Variable dependiente')
plt.grid()
plt.legend(loc='upper left')
```



Súper sexy. Es demasiado más de lo mismo como para codearlo, pero *scipy* incluye otro método popular para hacer integrales llamado **regla de Simpson**. Es igual que el de los trapecios, pero en vez de unir los puntos con rectas, lo hace con arcos de parábola. Lo pueden invocar con `scipy.integrate.simps` y no tiene mucha mayor ventaja que un menor error en la estimación del área.

## 3.4. Resolución de ecuaciones diferenciales

### 3.4.1. La idea teórica: método de Euler

*Esto es matemática, el que no quiera puede no leerlo*

Todos sabemos lo que es una ecuación diferencial, una ecuación que relaciona una variable dependiente con sus derivadas hasta el orden que sea y con una variable independiente. Supongamos que tenemos una ecuación diferencial de esta forma, junto a un valor inicial de la misma:

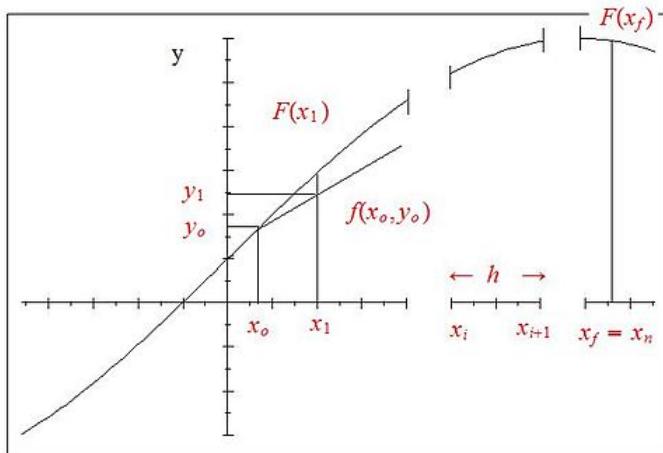
$$y'(t) = f(t, y)$$

$$y(t_0) = y_0$$

Si tenemos un vector de tiempo  $t$  con  $N$  valores, lo que nos interesa es hallar el vector  $y(t)$ , esto es,  $y(t_1) = y_1, y(t_2) = y_2, \dots, y(t_N) = y_N$ . Mostramos a continuación una gran idea de Euler para aproximar la solución  $y = F(t)$ . La idea es la siguiente: quiero hallar  $y = F(t)$  y sólo tengo una ecuación

para  $y'(t) = f(t, y)$  y el valor inicial  $y_0 = F(t_0)$ . Entonces, evaluamos en el punto inicial y es claro que  $F'(t_0) = y'(t_0) = f(t_0, y_0)$ . Este valor inicial es la pendiente de la recta tangente a  $F(t)$  en  $t_0$ , y por ende podemos estimar, según la gráfica:

$$\frac{y_1 - y_0}{t_1 - t_0} = f(t_0, y_0) \simeq F'(t_1)$$



En fin, la idea es que nos queda:

$$y_1 = (t_1 - t_0)f(t_0, y_0) + y_0$$

O sea, podemos calcular  $y_1$  a partir del valor inicial y el paso  $h = t_1 - t_0$ . Así podemos hallar  $y_2$  a partir del nuevo  $y_1$  hallado y en general:

$$y_{i+1} = h f(t_i, y_i) + y_i$$

Este método es, como se ve, recursivo y fácil de implementar, pero obviamente ya viene implementado. Además, como se ve en la figura,  $f(t, y)$  puede no parecerse mucho a  $F(t)$ . El método que viene implementado en Python no es por lo tanto este, sino algo muy parecido teóricamente aunque más pesado computacionalmente llamado 'Métodos de Runge-Kutta'. Como implementarlos en Python tiene su ciencia, pasemos directo a eso.

### 3.4.2. Ecuaciones diferenciales en Python 3

Vamos a trabajar con una función del módulo `scipy`, a saber:

```
from scipy.integrate import solve_ivp
```

Esta función nos pide varios argumentos obligatorios y tiene muchísimos opcionales. Los obligatorios son los que aparecen en la función

```
solve_ivp(f, [t_0, t_N], y_0)
```

- $f$  es la función que nos da la ecuación diferencial en sí misma;
- $t_0$  y  $t_N$  son una tupla que tiene los extremos del vector tiempo e
- $y_0$  es desde luego el valor inicial de la solución.

Entonces, supongamos que tenemos un vector tiempo de mil muestras, que va de 0 a 15 segundos y esta ecuación diferencial:

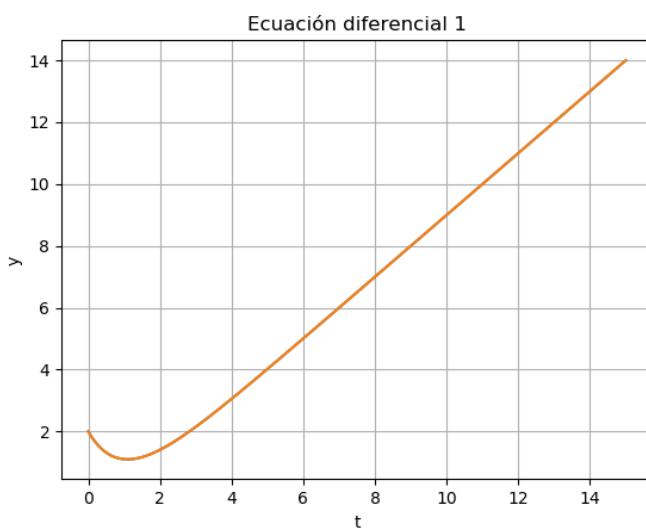
$$\begin{aligned}y'(t) &= t - y \\y_0 &= 2\end{aligned}$$

Y esto es tan estúpido como poner:

```
tspan = np.linspace(0, 15, 1000)
def edo1(t, y):
    return t - y
sol1 = solve_ivp(edo1, [tspan[0], tspan[-1]], [2],
                  t_eval=tspan)
```

Aquí estamos evaluando la solución en el vector `tspan` con el argumento opcional `t_eval`. La variable `sol1` es un objeto con muchos atributos interesantes, pero nos interesa rescatar y graficar el atributo `sol.y[0]`, quedándonos:

```
plt.plot(tspan, sol1.y[0])
plt.xlabel('t')
plt.ylabel('y')
plt.title('Ecuación diferencial 1')
plt.grid()
```



Algo súper interesante y que nunca había aparecido hasta ahora es que la función `solve_ivp` tuvo como un argumento la función `edo1`. Esto es algo que siempre se puede hacer en Python, pasar una función como argumento de otra función. Las funciones que reciben funciones como argumentos se llaman **funciones de alto orden** y fueron mencionadas en la sección 1.10.4.

A continuación mostramos algunos de los súperpoderes de la función `solve_ivp` con ejemplos. Supongamos inicialmente que queremos resolver la ecuación diferencial más fácil de mundo:

$$y'(t) = C = f(t, y)$$

Esto es muy fácil de hacer para cada valor de  $C$  que se nos ocurra, ¿pero qué tal si queremos variar el valor de  $C$  con facilidad? El método no nos permite poner  $C$  como variable de la función que va dentro de `solve_ivp`, pues tiene que ser  $f(t, y)$ . Una opción es hacer una función auxiliar:

```
def f(t, y, c):
    dydt = c[0]
    return dydt

def edo2(t,y):
    c = [1.2]
    return f(t, y, c)

sol2 = solve_ivp(edo2, [tspan[0], tspan[-1]], [-100],
                  t_eval=tspan)
# y_0 = -100
```

Otro superpoder del método es resolver varias EDO relacionadas entre sí, como por ejemplo:

$$\begin{aligned}\frac{dy_0}{dt} &= \alpha \cos(\beta t), \\ \frac{dy_1}{dt} &= \gamma y_0 + \delta t\end{aligned}$$

Armaremos un vector `c = [4, 3, -2, 0.5]` representando las constantes de la ecuación,  $(\alpha, \beta, \gamma, \delta)$ . Como hay dos ecuaciones diferenciales, necesitamos una tupla de valores iniciales, y nos queda algo así:

```
def f(t, y, c):
    dydt = [c[0] * np.cos(c[1]*t), c[2]*y[0] + c[3]*t]
    return dydt
```

```

def edo3(t, y):
    c = [4, 3, -2, 0.5]
    return f(t, y, c)

sol3 = solve_ivp(edo3, [tspan[0], tspan[-1]], [0,3],
                  t_eval=tspan)
# plt.plot(sol3.y[0])
# plt.plot(sol3.y[1])

```

Este último caso puede no parecer muy importante pero es bastante clave, porque nos permite resolver el problema final, que es de muchísima aplicación. ¿Qué hacemos con una ecuación de orden mayor a 1, o sea, algo como este oscilador forzado que tanto conocemos?:

$$\frac{d^2y}{dt^2} + \frac{b}{m} \frac{dy}{dt} + \frac{k}{m}y = \frac{F}{m} \cos(7t)$$

Para poder hacerlo hay que llevar el problema a algo de la forma  $y'(t) = f(t, y)$ . Felizmente, una ecuación de orden  $N$  siempre puede convertirse en  $N$  ecuaciones de orden 1. Lo hacemos con un cambio de variable.

$$\begin{aligned} \frac{d^2y_0}{dt^2} + \frac{b}{m} \frac{dy_0}{dt} + \frac{k}{m}y_0 &= \frac{F}{m} \cos(7t) \\ y_1 &= \frac{dy_0}{dt} \end{aligned}$$

Así la ecuación se reescribe como estas dos ecuaciones:

$$\begin{aligned} \frac{dy_1}{dt} &= \frac{F}{m} \cos(7t) - \frac{b}{m}y_1 - \frac{k}{m}y_0 \\ \frac{dy_0}{dt} &= y_1 \end{aligned}$$

Ahora son dos ecuaciones igual que en el ejemplo anterior. Sale con fritas:

```

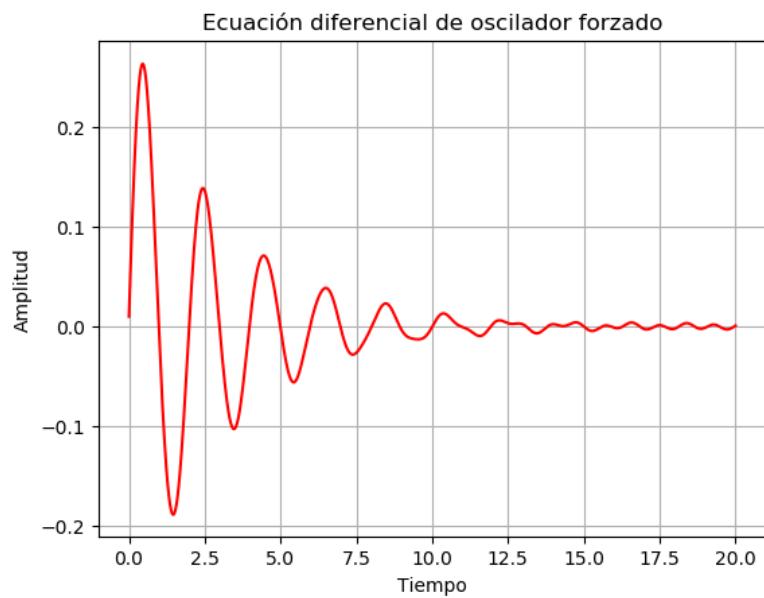
def f(t, y, c):
    dydt = [y[1], (0.001/c[0])*np.cos(7*t)-(c[1]/c[0])*y[1]-
            (c[2]/c[0])*y[0]]
    return dydt

def edo4(t, y):
    c = [0.01, (10**0.5) / 500, 0.1]
    return f(t, y, c)

sol4 = solve_ivp(edo4, [tspan[0], tspan[-1]], [0.01, 0.95],
                  t_eval=tspan)

```

Obviamente pusimos dos valores iniciales. Terminamos con el gráfico de este sistema tan conocido.



Y colorín colorado, este capítulo se ha terminado.

# Capítulo 4

## La transformada de Fourier

Hemos hablado varias veces de cosas que sabemos hacer por Fourier, que debemos a Fourier, que salen por Fourier y siempre Fourier. Gran tipo ese Fourier, un fenómeno. Casi toda la matemática de una carrera de ingeniería puede verse, si uno piensa en las aplicaciones, como una larga escalera a poder entender bien la transformada de Fourier. Por eso, entenderla en toda su dimensión exige análisis matemático complejo, una buena cuota de álgebra, un espíritu aventurero y un corazón puro (?). Bueno, no, pero análisis complejo y álgebra sí. Por eso, este capítulo no va a tener mucha programación dura ni algoritmos muy delicados. Tampoco va a tener mucha matemática dura, que no es lo que buscamos en este libro, pero sí va a tener un vistazo general de las cosas que se pueden hacer con la transformada de Fourier, por qué funcionan y cómo se implementan en Python. Para ello, vamos a utilizar varias funciones ya implementadas en las librerías `numpy` y `scipy`, cuya aplicación es bastante técnica y nos va a exigir explorar el oscuro —aunque apasionante— mundo del procesamiento digital de señales.

### 4.1. El análisis de espectro

Hasta ahora estuvimos viendo señales que representaban audio como variaciones de presión a través del tiempo, o sea, una amplitud que va variando en función de una variable en segundos, minutos o lo que sea. Eso está buenísimo, es muy útil en muchísimas cosas y ya mostramos cómo hacer algunas cosas divertidas a partir de eso, pero la realidad es que no se parece demasiado al modelo como nosotros pensamos el sonido —o muchas otras señales—. Decimos esto porque, cuando escuchamos un sonido, no sólo nos preocupamos porque sea fuerte o suave, sino que también nos fijamos en que sea “grave” o “agudo”, y más aún, *sonidos igual de “graves” o “agudos” suenan sin embargo*

go muy diferentes. Cuando tocamos una nota “La” en un piano o un “La” en un saxofón, sabemos claramente cuál es el piano y cuál es el saxofón. Vamos a tratar de entender cómo se da esa diferencia y cómo usarla para analizar y procesar señales en Python o cualquier otro lenguaje.

Las señales acústicas más pavotas de todas son las señales senoidales, también conocidas como *tonos puros*. Tienen una amplitud, un ángulo de fase, pero lo que más nos va a importar hoy es que tienen **una única frecuencia**. Esto lo conocemos bien: una señal de mayor frecuencia hace más ciclos por segundo. La fórmula general de una señal senoidal es:

$$x(t) = A \operatorname{sen}(2\pi ft + \phi) \quad (4.1)$$

Vamos a definir y graficar tres senoides con este modelo, y vamos a usarlas como ejemplo a lo largo del capítulo. Todo lo que vimos sobre gráficos en el capítulo 3 nos va a resultar muy útil:

$$\begin{aligned} x_1(t) &= 10 \operatorname{sen}(2\pi(3)t) \\ x_2(t) &= 5 \operatorname{sen}(2\pi(6)t) \\ x_3(t) &= 7 \operatorname{sen}(2\pi(10)t) \end{aligned}$$

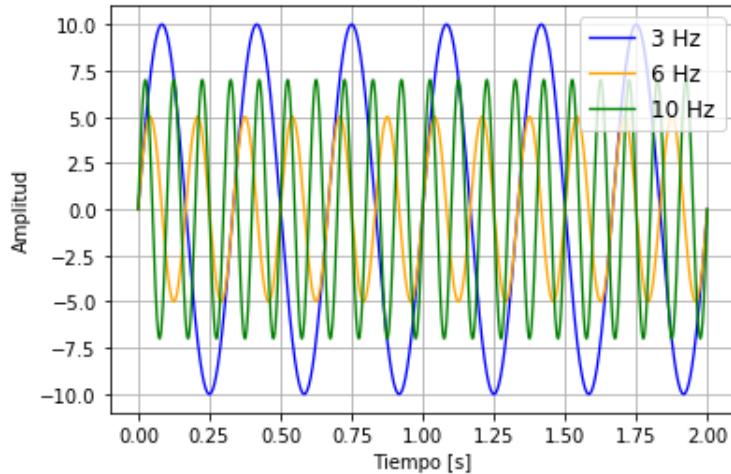
Armemos un gráfico y de paso repasamos un poco el capítulo anterior:

```
import numpy as np
from matplotlib import pyplot as plt
from scipy import signal

fs1 = 44100
tiempo_tono = np.linspace(0, 2, 2*fs1)
tono1 = 10 * np.sin(2*np.pi*3*tiempo_tono)
tono2 = 5 * np.sin(2*np.pi*6*tiempo_tono)
tono3 = 7 * np.sin(2*np.pi*10*tiempo_tono)

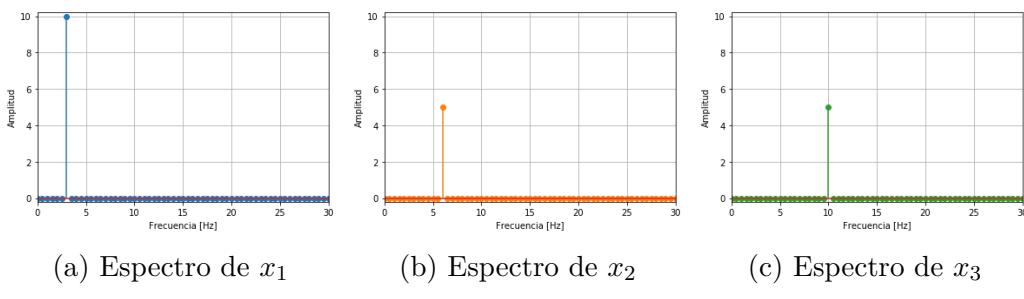
plt.plot(tiempo_tono, tono1, linewidth=1.3, label='3 Hz',
          color='blue')
plt.plot(tiempo_tono, tono2, linewidth=1.3, label='6 Hz',
          color='orange')
plt.plot(tiempo_tono, tono3, linewidth=1.3, label='10 Hz',
          color='green')
plt.xlabel('Tiempo [s]')
plt.ylabel('Amplitud')
plt.grid()
plt.legend(fontsize='large')
```

El gráfico resultante es:



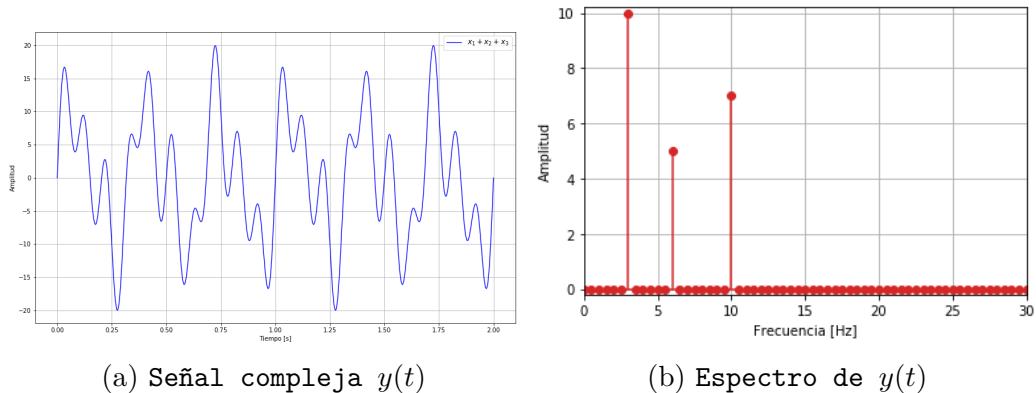
Nada raro. Las señales con frecuencia más alta fluctúan más veces por segundo, las de frecuencia más baja menos, todo normal. Ahora bien, la señal es periódica. Una vez que sabemos la amplitud y la frecuencia, el gráfico medio que se cuenta solo. Podemos caracterizar esto en un gráfico mucho más elemental y claro valiéndonos sólo de la frecuencia de cada señal y la amplitud que tiene. Podemos, concretamente, hacer un gráfico de amplitud vs. frecuencia, poniendo un valor de 10 en 3 Hz para  $x_1$ , uno de 5 en 6 Hz para  $x_2$ , y uno de 7 en 3 Hz para  $x_3$ .

Ahora bien, si cada señal es periódica, y a medida que pasa el tiempo sólo repite su comportamiento, podemos caracterizar a un seno con un gráfico distinto, mucho más cómodo, solamente sabiendo su frecuencia y su amplitud. De este modo, si la señal tiene una frecuencia de 3 Hz y una amplitud de 10, tendremos un pico de altura 10 en el valor 3 del eje de frecuencias. Un gráfico de  $x_1$ ,  $x_2$  o  $x_3$ , por ejemplo, nos quedaría así:



Esto nos permite analizar cómo está hecha una señal a partir de su composición en frecuencias. De este modo, este nuevo diagrama, que denominaremos espectro, nos muestra algo nuevo: ya no importa cómo varía la amplitud de

la señal en función del tiempo, sino cuánta amplitud hay en cada frecuencia que compone una señal. Si analizo un sonido más interesante, podría armar, por ejemplo, la señal más compleja  $y(t) = x_1(t) + x_2(t) + x_3(t)$ . Si graficamos la señal en dominio del tiempo y en dominio de la frecuencia tenemos:

(a) Señal compleja  $y(t)$ (b) Espectro de  $y(t)$ 

Este gráfico muestra todo el poder que tiene esta forma de visualizar señales. Fíjense que el diagrama temporal es un quilombo total, incomprensible, nadie sabe qué está pasando ahí con facilidad, pero en cambio el frecuencial es claro y cristalino, como un manantial que fluye. Sí, así de poético. La primera gran ventaja de esta representación es, por lo tanto, que permite analizar con mucha facilidad la composición de una señal complicada como combinación de señales sencillas, y así entender cómo se construye la señal compleja. Esta práctica se denomina *análisispectral* y es lo más útil del universo, pues nos permite entender señales de otro modo muy complicadas si no incomprensibles. Sus aplicaciones son, en serio, incontables.

Hay una segunda gran utilidad, mucho más divertida, para esta forma de pensar las señales, que viene desde la perspectiva del diseño: es muy difícil imaginar cómo va a quedar una onda temporal compleja, y en cambio es más fácil imaginar cómo va a quedar agregándole más “energía aguda” o “energía grave”, esto es, agregar cosas en el espectro que lo hagan más interesante. Esta pavada es el fundamento de algunas de las prácticas más poderosas del procesamiento de señales acústicas. En particular, esto es lo que nos permite la *ecualización* y la *síntesis aditiva*.

La operación que nos permite, dada una señal temporal, obtener su comportamiento en frecuencia, se llama *Transformada de Fourier*, y algunos de sus súperpoderes son el objeto de este capítulo.

## 4.2. La Transformada (discreta) de Fourier

Entender la transformada de Fourier es algo que se come como tres materias de facultad y tiene un montón de versiones y demás. Para este apunte vamos a ser bastante informales y además sólo vamos a mostrar la fórmula y cómo es que la implementamos usando `numpy`. La transformada de Fourier se calcula así (no nos importa que lo sepan hacer, pero bueno, como para que vean de qué va la cosa):

$$X(f) = \sum_{n=-\infty}^{\infty} x[n]e^{-j2\pi fn}$$

Esa señal nueva que tiene  $f$  como variable es la que contiene la amplitud en función de la frecuencia, y se calcula a partir de un vector  $x[n]$ , que es la versión discreta de una señal analógica  $x(t)$ . Además, algo que uno puede hacer es, si tiene la señal transformada  $X(f)$ , puede recuperar la señal discreta original  $x[n]$  con otra operación, llamada transformada inversa de Fourier:

$$x(t) = \int_a^{a+2\pi} X(f)e^{j2\pi fn} df$$

Acá  $a$  puede ser cualquier cosa. En rigor, estas operaciones no son exactamente la operación que se hace en la computadora<sup>1</sup>. Por otro lado, la operación discreta es muy pero muy parecida, y a fines prácticos es lo similar trabajar con ella. En `numpy`, la transformada de Fourier se hace con un algoritmo que la calcula bastante rápido, y que se llama `fft` (por *fast Fourier transform*). El siguiente código levanta una nota La (440 Hz) de piano grabada en un archivo .wav, con intensidad *mezzoforte*, calcula su transformada de Fourier y grafica el espectro:

```
import numpy as np
from matplotlib import pyplot as plt
import soundfile as sf

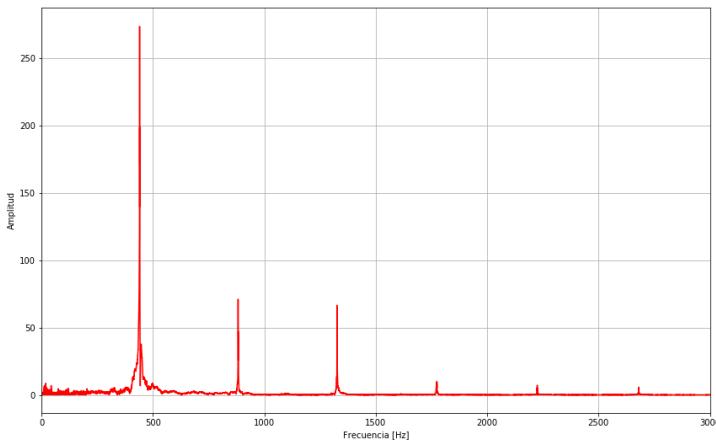
(piano1, fs1) = sf.read('piano_mf_A4.wav')
piano1 = piano1[:, 0]
freqs1 = np.fft.rfftfreq(len(piano1), 1/fs1)
fourier1 = np.fft.rfft(piano1)
espectro1 = np.abs(fourier1)
espectro1[0] = 0
```

Y le hacemos un grafito al resultado:

---

<sup>1</sup>La operación aquí mencionada se llama DTFT (*discrete time Fourier Transform*) y la que se hace en la computadora es DFT (*discrete Fourier Transform*)

```
plt.figure(1)
plt.plot(freqs1, espectro1, 'r')
plt.grid()
plt.xlim([0, 3000])
plt.xlabel('Frecuencia [Hz]')
plt.ylabel('Amplitud')
```



Hay un millón de cosas para comentar de este gráfico:

- Como primera medida, vemos que la nota *La* tiene un pico muy pronunciado en 440 Hz. Esto está bien, tiene sentido, pero lo más importante es que ese pico *no es el único* del espectro. Si zoomeamos bien vamos a ver que hay un pico en 880 Hz, otro en 1320 Hz, y otros más chicos en 1760, 2200 y 2640. Todos estos son múltiplos de 440 Hz, que es la *fundamental* del sonido analizado. Esos múltiplos se llaman *armónicos*.
- El código lee la señal de audio y la lleva a mono, y luego utiliza la función `np.rfftfreq` para obtener el vector que va al eje *x* y la función `np.rfft` para el eje *y*.
- Después hace algo muy importante que es tomar el módulo: la transformada de Fourier tiene una *j* dando vuelta, con lo cual puede (y en general, en la computadora siempre lo hará) dar un resultado complejo, así que hay que calcularle el módulo antes de graficar.
- Finalmente, eliminamos el primer valor de la señal (o lo dejamos fuera de consideración), también llamado componente de continua. Si nuestra señal estuviera desplazada hacia arriba o abajo en una constante (o sea, fuera de la forma  $x(t) + C$ ) esa constante es una señal de frecuencia cero (no oscila), por lo cual su valor se computa en la frecuencia cero

y suele dar mucho muy grande. Como las variaciones que interesan en general son relativas al  $C$ , y no absolutas, lo eliminamos y chau.

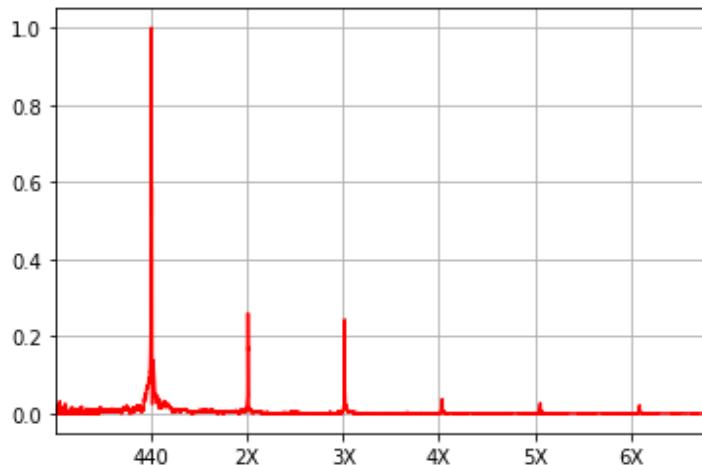
Una cosa del gráfico que merece muchísima consideración es la amplitud de cada pico. En general, no hay nada muy ilustrativo en ese valor numérico a menos que lo calibre contra algo, ya que, por cómo se computa la transformada, si yo tomara una señal con más muestras da más alto y eso no implica más energía en una frecuencia real. Por ello, lo que suele hacerse es normalizar o escalar el gráfico, lo cual puede hacerse de muchas maneras. Entonces, en este caso, agregamos la línea:

```
espectro1 /= max(espectro1)
```

También vamos a agregar unos ticks en las armónicas, para mejor visualización:

```
plt.xticks([440, 880, 1320, 1760, 2200, 2640],  
         ['440', '2X', '3X', '4X', '5X', '6X'])
```

Nos queda lo siguiente:



Fíjense que esa amplitud de 1 en el pico de la fundamental no significa nada. ¿1 dB, 1 Pa? Nada. Lo que sí es muy interesante es ver qué amplitud tienen los otros picos, porque si en 880(2X) tengo aproximadamente 0.26 y en 1320(3X) tengo más o menos 0.24, entonces puedo decir que una nota de piano se forma con un 26 % de la fundamental en la 2da armónica y un 24 % en la 3ra. Esto puedo estudiarlo hasta frecuencias más elevadas y ver qué resulta. Con esto, lo que puedo hacer es *extender esto para diseñar cualquier nota de piano*: basta tomar la nota base que quiero que suene y sumarle

tonos puros con las amplitudes adecuadas en las frecuencias adecuadas. Esto se conoce como *síntesis aditiva* y es evidentemente muy útil.<sup>2</sup>

### 4.3. Filtros, Transformada Z y ecualización

Es verdaderamente difícil explicar esto sin caer en el cálculo complejo y la teoría de señales y sistemas. Intentaremos minimizar lo más posible el “creer o reventar” en esta sección, pero es cierto que puede llegar a ser bastante abstracto. Bah, ni que sea imposible, con algo de paciencia se entiende, como todo. Pongamos especial atención al código, de todos modos, que es lo que nos importa desde lo que se hace en Python.

Ya habíamos visto de pasada, en la sección 2.5.5, que el efecto de un sistema sobre una señal se modela con una operación llamada **convolución**. Así, dada una entrada  $x[n]$ , y una señal que usamos para modificar la entrada,  $h[n]$  (en general, será una respuesta al impulso), la señal de salida se obtiene como:

$$y[n] = x[n] * h[n]$$

Esta operación, lo hemos charlado, es compleja, lenta y tediosa, y sobre todo difícil de interpretar para señales y respuestas al impulso de cierta complejidad. Por ello, se usa una operación conocida como Transformada Z, que, así como la transformada de Fourier nos lleva las cosas a un mundo donde la variable independiente es la frecuencia, nos lleva las señales a un mundo donde la variable independiente es  $z$ , una variable compleja abstracta:

$$X(z) = \sum_{n=-\infty}^{\infty} x[n]z^{-n}$$

La transformada Z nos servirá principalmente por dos cosas. En primer lugar, fíjense que  $z$  es cualquier número complejo, pero si analizamos la transformada  $z$  para casos en que  $|z| = 1$ , o sea, cuando  $z = e^{j2\pi f}$ , la transformada Z es idéntica a la transformada de Fourier.

La otra propiedad importante es que en el mundo  $z$  las convoluciones se convierten en productos, y la señal de salida anterior queda como:

$$Y(z) = X(z)H(z) \longrightarrow H(z) = Y(z)/X(z)$$

Este cociente es, en cualquier situación de interés, un cociente de polinomios, y  $H(z)$  es una función que nos representa qué efecto tiene el sistema

---

<sup>2</sup>Este método, por cierto, tiene importantes limitaciones. Algunas de ellas las exploraremos en los ejercicios sobre este capítulo

que analizo (o sea, su respuesta al impulso) sobre cualquier entrada  $X(z)$ . Esto es, la transfiere a la salida  $y$ , por eso, se llama *Función de transferencia*.

Ahora viene la parte interesante: como es un cociente de polinomios, la función  $H(z)$  tiene *ceros* (que mandan toda la entrada a cero) cuando hay raíces en el numerador y *polos* (que mandan la entrada a infinito) cuando hay raíces en el denominador. Esto nos sirve muchísimo, porque *para valores de  $z$  cercanos a un cero el sistema está atenuando y, para valores de  $z$  cercanos a un polo el sistema está amplificando*.

¿Y qué es esa  $z$  entonces? Ni idea, pero como dijimos, si  $z = e^{j2\pi f}$  la operación se convierte en la transformada de Fourier, y entonces podemos decir que distintos valores de  $f$  nos engendran distintos valores de  $z$ . *Si un valor de  $f$  nos acerca a un cero, tengo atenuación en esa frecuencia, y si un valor de  $f$  nos acerca a un polo, entonces tengo amplificación en esa frecuencia.* Esta práctica, atenuar o amplificar en función de la frecuencia multiplicando por una función que atenúa o amplifique ciertas frecuencias, se llama *filtrar o ecualizar*. Más importante, podemos diseñar un filtro con un método muy sencillo eligiendo los polos y ceros del cociente de polinomios que es su función de transferencia, antitransformarlo y, con ello, ya tenemos diseñado un filtro a partir de su respuesta al impulso  $h[n]$ .

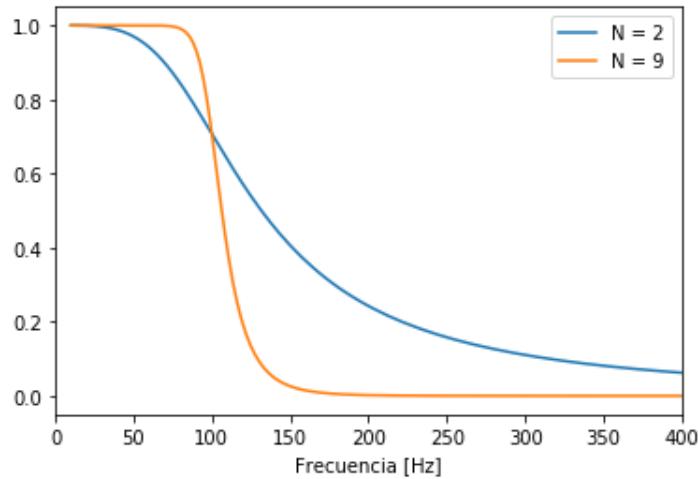
## 4.4. Filtros estándar

En Python, felizmente, hay hechas funciones que nos permiten realizar este proceso con poco trabajo. Vamos a mostrar cómo hacer filtros Butterworth y filtros Chebyshev tipo 1 y 2, que son filtros muy famosos que se usan para millones de cosas. Utilizaremos el módulo `signal` de la librería `scipy`, en particular la función `signal.butter`, que toma como argumentos *el orden del filtro, la frecuencia de corte, qué efecto tiene el filtro (pasabajos, pasaaltos, pasabanda, etc.), el tipo de filtro (analógico / digital) y cómo nos da la salida*. Vamos a hacer dos gráficos con distintos órdenes del filtro para entender qué es cada una de estas cosas:

```
#signal.butter(N, f_corte, btype='low', analog=True,
#                  output='ba')
from scipy import signal
###FILTROS BUTTERWORTH
plt.figure(2)
b1, a1 = signal.butter(2, 100, 'low', analog=True)
w1, h1 = signal.freqs(b1, a1)
plt.plot(w1, (abs(h1)), label='N = 2')
b2, a2 = signal.butter(9, 100, 'low', analog=True)
```

```
w2, h2 = signal.freqs(b2, a2)
plt.plot(w2, (abs(h2)), label='N = 9')
plt.xlim([0, 400])
plt.xlabel('Frecuencia [Hz]')
plt.legend(fontsize='large')
```

Sí, ya sabemos, es un montonazo, pero el grafito lo deja más claro:



En ambos casos le pasamos a `signal.butter` una frecuencia de corte de 100, que sea un filtro “lowpass” o pasabajos, de tipo analógico. En el primer caso le pasamos orden 2, y en el segundo, orden 9. La función `signal.butter` devuelve dos vectores `b` y `a` que son los coeficientes de numerador y denominador de la función de transferencia para un filtro que cumpla con lo pedido, y la función `signal.freqs` devuelve la respuesta en frecuencia del filtro (en la variable `h1`) y las frecuencias para el eje `x` a partir de los coeficientes obtenidos de `signal.butter`. Cositas para decir:

- La frecuencia de corte nos dice en dónde el filtro “deja de ser efectivo”. Esto se definió arbitrariamente como cuando le saca 3 dB o más a la señal en la frecuencia en cuestión. Esto equivale a la frecuencia para donde el filtro multiplica por  $\sqrt{2}/2 \approx 0.707$ . Si se fijan, en 100 Hz, ambos filtros tienen ese valor.
- El orden nos dice qué tan preciso es el filtro, esto es, qué tan escarpadamente llega a la frecuencia de corte. El filtro de orden 9 multiplica por 1 prácticamente todo lo que va entre 0 y 100 Hz (o sea, lo deja intacto), y por 0 lo que sea mayor a 100 Hz (o sea, lo hace nulo). Es un pasabajas muy preciso, como puede verse. El filtro de orden 2, por

otro lado, es más impreciso y no cae completamente en 100, sino que tarda un poco más en caer.

- Que el filtro sea pasabajos significa que deja pasar desde 0 hasta cierto valor, que es la frecuencia de corte. Si el filtro fuera pasaaltos, dejaría pasar desde una frecuencia de corte para adelante, y no para atrás. Un filtro pasabanda, finalmente, es un filtro que tiene una frecuencia de corte inferior y una superior.
- No hemos especificado el `output='ba'`, que está por defecto. Hay varias formas de dar un filtro digital. Las más comunes son dar los polos y ceros (que se hace poniendo `output='zpk'`) o dar los coeficientes del numerador y denominador en la función de transferencia, que son los vectores `b` y `a`, respectivamente. Eso es elegir `output='ba'`<sup>3</sup>.
- `analog=True` indica que podemos pasar la frecuencia de corte en Hz. El filtro digital puro debería especificarse según el valor de  $\omega = 2\pi f$  en el exponente de  $e^{j\omega}$ , que indica cuánto rotamos en el círculo donde nos vamos encontrando los polos y ceros. Obviamente ese valor debe ir entre 0 y  $2\pi$ , donde una rotación de  $2\pi$  se corresponde con la frecuencia de Nyquist, que es la máxima frecuencia presente en la señal que trabajamos y por encima de la cual filtrar no tendría sentido. Como la frecuencia máxima que podamos usar viene dada por la mitad frecuencia de muestreo, siempre haremos corresponder  $\omega$  con  $f_s/2$ .

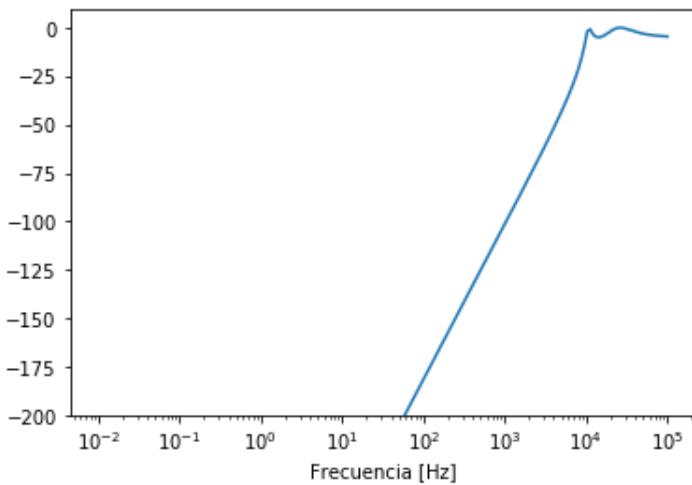
Hagamos ahora un filtro de otro tipo, a saber, un Chebyshev tipo 1, y que sea pasaaltos, que es muy parecido a la anterior con alguna que otra salvedad. En general el filtro Chebyshev 1 se diseña en decibeles (o sea, los valores de ganancia que vemos se suman, no se multiplican), y lo único que cambia es que además del orden se agrega un valor de *ripple*, que es cuánto fluctúa el filtro en la banda de paso. La frecuencia de corte es la primera frecuencia para la cual se alcanza el *ripple* deseado:

```
###FILTRO CHEBYSHEV 1
plt.figure(4)
b3, a3 = signal.cheby1(4, 5, 10000, 'high', analog=True,
                       output='ba')
w3, h3 = signal.freqs(b3, a3)
plt.semilogx(w3, 20*np.log10(abs(h3)))
```

---

<sup>3</sup>Hay un output más, '`sos`' (*second order sections*), que es una corrección numérica para filtros que pierden resolución en baja frecuencia. Por eso va a ser el que más usemos, pero no es muy diferente a nivel teórico.

```
plt.ylim([-200, 10])
plt.xlabel('Frecuencia [Hz]')
```



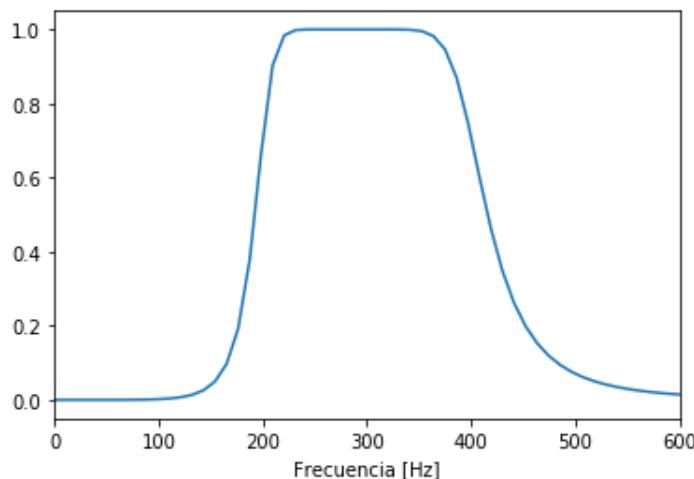
Es un Chebyshev 1 pasaaltos de orden 4, con un *ripple* de 5 dB en la frecuencia de corte de 10 kHz. Como ven, hay ganancia cercana a 0 (no suma nada) en la banda de paso, o sea, de 10 kHz para adelante, y resta muchísimo en la banda de parada. Para interesados, el Chebyshev tipo 2 es algo parecido pero el *ripple* no está en la banda de paso sino en la parada, en la parte donde el filtro atenúa. Para cerrar, armamos un filtro Butterworth pasabanda de orden 5, que es el que se usa para la mayoría de las aplicaciones en acústica. Vamos a diseñarlo como función, y vamos a hacerlo digital (o sea, no dando la frecuencia en Hz, sino referenciando qué valor de  $\omega$  dentro de la circunferencia  $|z| = 1$  define las frecuencias de corte):

```
###FILTROS BUTTERWORTH
def butter_bandpass(lowcut, highcut, fs, order=5):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    sos = signal.butter(order, [low, high], analog=False,
                        btype='band', output='sos')
    return sos
```

Por facilidad de uso, las frecuencias de corte inferior y superior se eligen en Hz, como los argumentos `lowcut` y `highcut`, pero se llevan a valores digitales dividiendo por la frecuencia de Nyquist. En vez de sacar 'ba'

usamos '`sos`' porque nos da más estabilidad en bajas frecuencias, y establecemos que el filtro será digital. Le hemos dado el valor de orden 5 por defecto, que es el que se suele usar para aplicaciones de medición. Lo usamos así para hacer un filtro de orden 5 que deje pasar lo que esté entre 200 y 400 Hz:

```
sos = butter_bandpass(200, 400, 44100, order=5)
w, h = signal.sosfreqz(sos, worN=2000)
plt.figure(5)
plt.plot((fs1/2)*(1/np.pi)*w, abs(h))
plt.xlim([0, 600])
```



La función `signal.sosfreqz` hace lo mismo que `signal.freqs` pero con `output='sos'`. El argumento `worN` nos dice para cuántos valores de frecuencia computar la respuesta del filtro. Se ve cómo el 0.707 está en 200 y 400 Hz, con lo cual el filtro cumple lo pautado.

Nos queda una ultimísima cosa de la cual no hemos hablado: estos filtros están preciosos, su respuesta está genial, ¡pero la idea de esto es usarlo para filtrar una señal! ¿Cómo filtramos la señal con el filtro diseñado?

Bueno, la teoría más o menos ya la tenemos: sacamos los coeficientes del filtro (polinomio numerador, polinomio denominador), armamos los polinomios, dividimos y eso es la función de transferencia, que se multiplica en frecuencia. El proceso lógico para partir de una señal y obtener su versión filtrada es así:



De todos modos, por ser este proceso siempre igual, no tiene mucho sentido hacerlo una y otra vez, así que aprovechamos las funciones que ya vienen hechas en el módulo `signal`. Aquí hay distintas funciones para filtrar con coeficientes, polos o ‘sos’, pero todas son más o menos lo mismo. Para un filtro hecho con ‘sos’, por ejemplo, es tan tonto como insertar un vector con la señal de audio y la salida del filtro hecho con las técnicas anteriores:

```
y = signal.sosfilt(sos, audio)
```

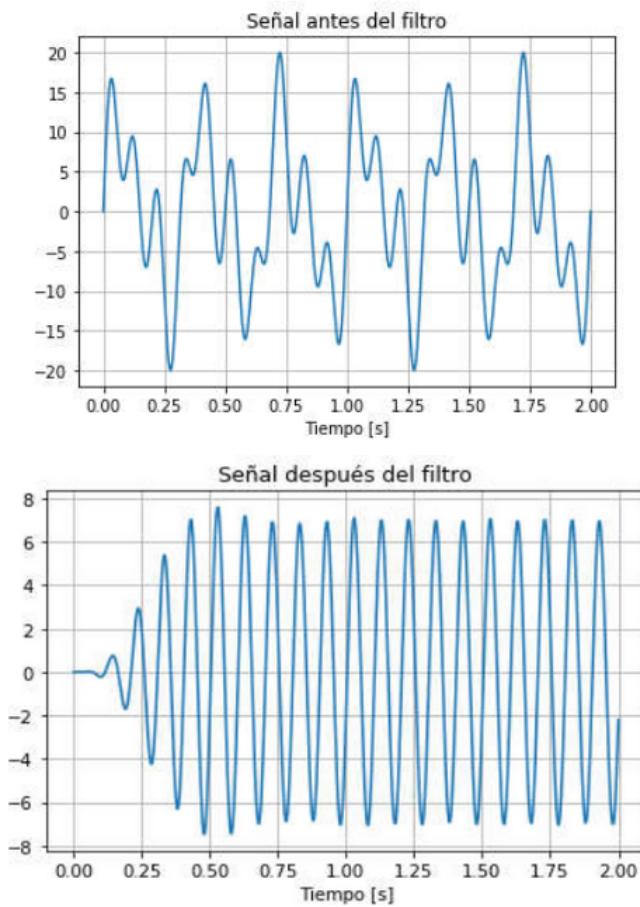
Y listo, y contiene la señal filtrada.

Una función súper útil para filtrar señales con un Butterworth pasabanda (que complementa la función `butter_bandpass` que hicimos) sería:

```
def butter_bandpass_filter(data, lowcut, highcut, fs,
                           order=5):
    sos = butter_bandpass(lowcut, highcut, fs, order=order)
    y = signal.sosfilt(sos, data)
    return y
```

Para ver este filtro en acción, construyamos la señal `tono1 + tono2 + tono3` con las señales que usamos al principio del capítulo. Esta señal tiene contenido en 3, 6 y 10 Hz. Si lo pasamos por un filtro pasabanda centrado en 10 Hz, deberíamos recuperar la señal `tono3`. Entonces:

```
x_t = tono1 + tono2 + tono3
y = butter_bandpass_filter(x_t, 8, 12, 44100)
plt.figure(10)
plt.plot(tiempo_tono, x_t)
plt.xlabel('Tiempo [s]')
plt.grid()
plt.figure(11)
plt.plot(tiempo_tono, y)
plt.grid()
plt.xlabel('Tiempo [s]')
```



Tenemos un pequeño transitorio al aplicar el filtro, pero la señal se estabiliza en el tono de 10 Hz que debería ser. Con ello nos damos por satisfechos, y, por lo tanto cerramos este capítulo tan abstracto de nuestras vidas. No dejen de pasar distintos filtros por señales de audio y escuchar el resultado, es verdaderamente ilustrativo, y no dejen de hacer los ejercicios del capítulo, que sirven para toda la vida.

# Capítulo 5

## Orientación a objetos

Señalamos al final de la sección 1.8.2 que **programar en python es manejar list, if y while**, porque son los elementos que lo constituyen como lo que se denomina un **lenguaje Turing-completo**. Todo elemento *adicional* existe para *minimizar redundancia y aumentar legibilidad*.

Los **objetos** no son la excepción.

En **python**, todas las variables que declaramos (**int, float, bool, str, list**) son en realidad objetos [1]. Su naturaleza se pone de manifiesto explícitamente al utilizar lo que se llaman **métodos** del objeto.

### 5.1. Métodos

Los **métodos** no son otra cosa que **funciones** [2]. Nosotros ya usamos métodos antes, como por ejemplo, el método **lower** de **str** en la sección 1.10, para convertir las mayúsculas en minúsculas. Ejemplo nuevo: el método **append** de **list**.

```
In [1]: numeros = [1055, 2339, -592, 2938, 12]
In [2]: numeros.append(71)
In [3]: numeros
Out[3]: [1055, 2339, -592, 2938, 12, 71]
```

Notar que **numeros.append(71)** es lo mismo que **numeros += [71]**.

No haremos una lista exhaustiva de los métodos que tienen **list** y **str**. Si les interesa eso, vean

- <https://docs.python.org/3/library/stdtypes.html#string-methods>
- <https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

Un objeto no tiene por qué limitarse a un tipo nativo de **python** como **list**. Podemos definir nuestros propios objetos a partir de **clases**.

## 5.2. Clases

Se denomina **clase** al **tipo de objeto** [3]. La *clase* define una *estructura* de datos, los *objetos contienen datos* que siguen esa estructura.

CLASE	OBJETO
define un	contiene los
<b>tipo</b>	<b>datos</b>

Declaramos una clase en python con la palabra **class**.

```
class Televisor:
    def __init__(self):
        self.canal = 1

    def cambiar(self, canal):
        if canal == 3: # TN
            print('LA RUTA DEL DINERO K')
        elif canal == 6: # C5N
            print('La rrrrealidad')

        self.canal = canal
```

Corramos el código para guardar la definición de la clase en la memoria, la cual usaremos en breve. Antes, es necesario definir y aclarar algunas cosas.

### 5.2.1. Constructor, atributos y métodos

La función `__init__` es el **constructor** de la clase. Es **lo primero que se ejecuta al crear un objeto**. Creamos un objeto de clase `Televisor` así:

```
mi_tele = Televisor()
```

Pongamos entonces en la Terminal

```
In [4]: mi_tele = Televisor()
In [5]: mi_tele.canal
Out[5]: 1
```

es decir, al crear el objeto `mi_tele`, su método `__init__` es ejecutado automáticamente, y este define internamente la variable `canal` como 1.

Las **variables** de un objeto se denominan **atributos** [1]-[3]. Así, `canal` es un atributo del objeto `mi_tele`.

La función `cambiar` es un **método**, que usamos así:

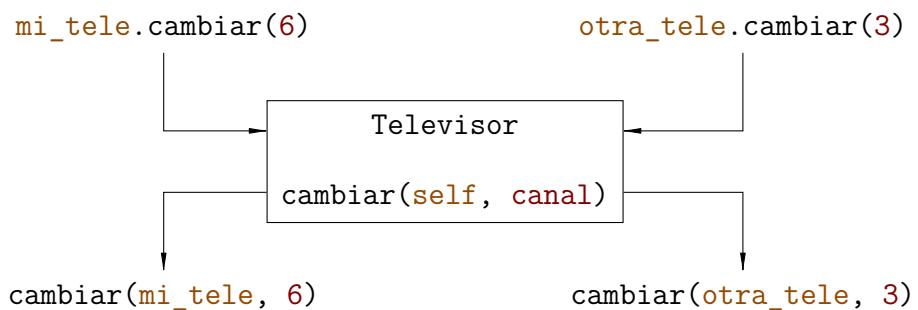
```
In [6]: mi_tele.cambiar(6)
La rrrealidad
In [7]: mi_tele.canal
Out[7]: 6
```

El método `cambiar` modifica efectivamente el atributo `canal`, como establecimos en la correspondiente definición del método dentro de la clase. Notemos que, para objetos de clase `Televisor`, el uso del método es exactamente el mismo que para objetos de `list` y `str`: se pone un punto `.` entre el nombre del objeto y el método que se quiere llamar.

Podemos tener más de un objeto de la misma clase.

```
In [8]: otra_tele = Televisor()
In [9]: otra_tele.cambiar(3)
LA RUTA DEL DINERO K
In [10]: otra_tele.canal
Out[10]: 3
```

Con esto, el significado del `self` queda un poco más claro: la variable `self` refiere al *objeto propiamente dicho* dentro de la correspondiente definición de la clase. En el caso de nuestro ejemplo:



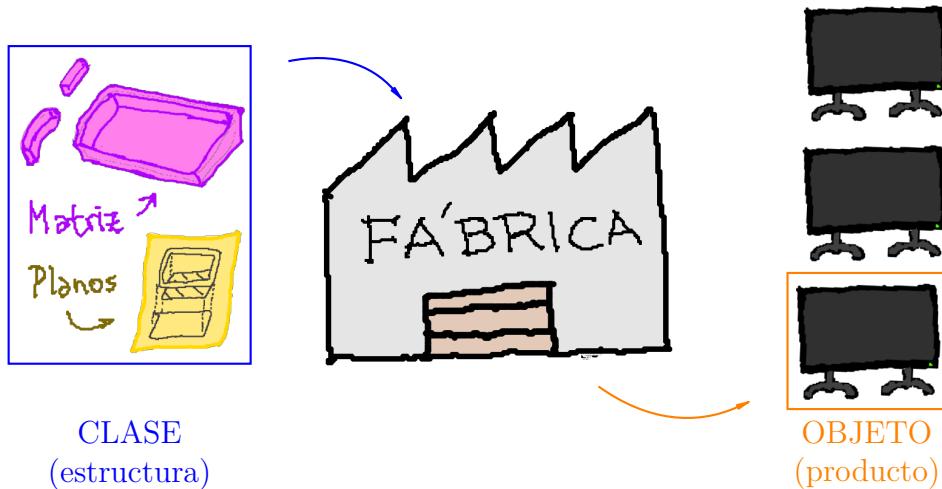
O sea, `mi_tele.cambiar(6)` llama al método `cambiar(self, canal)` de la clase `Televisor` estableciendo `self` como `mi_tele` y `canal` como 6. En cambio, `otra_tele.cambiar(3)` llama al método con `self` igual a `otra_tele` y `canal` igual a 3. Cada objeto tiene un manejo interno de la variable `self` que es independiente del otro.

### 5.3. Analogía industrial

En la sección 5.2 establecimos una importante distinción.

**La clase define un tipo, el objeto contiene datos de ese tipo.**

Los objetos son como productos de una fábrica [1].



En esta visión, la clase define la estructura que tendrá el producto, y cada uno de los objetos *es* el producto. Toda *funcionalidad* del producto es un *método*, y toda *característica* que podemos cambiar a través de los métodos es un *atributo*.

## 5.4. Herencia

Creemos una nueva clase SmartTV. Obviamente, el SmartTV es en sí mismo un Televisor, pero tiene métodos que la clase Televisor no, como por ejemplo ver\_youtube. Cuando decimos *el SmartTV es un Televisor* establecemos implícitamente una relación de **herencia** entre las clases [3].

Heredamos de Televisor la clase SmartTV así:

```
class SmartTV(Televisor):
    def ver_youtube(self):
        print('Justin Bieber surprises Billie Eilish')
```

es decir, heredamos cuando ponemos la palabra Televisor entre paréntesis después de la declaración convencional de la clase.

Ejecutemos el código y, después, pongamos en la terminal

```
In [11]: super_tele = SmartTV()
In [12]: super_tele.canal
Out[12]: 1
In [13]: super_tele.cambiar(6)
La rrrrealidad
In [14]: super_tele.ver_youtube()
Justin Bieber surprises Billie Eilish
```

El objeto de clase `SmartTV` ya viene con el atributo `canal` y el método `cambiar` porque los copia de la clase `Televisor`: se dice que `SmartTV` **hereda** los métodos y atributos de `Televisor`. Es por esta razón que no hace falta declarar explícitamente nada al definir la clase `SmartTV`.

#### 5.4.1. Sobrescritura

A menudo interesa modificar parte del comportamiento de una clase. Para eso, la clase heredada **vuelve a definir** uno de los métodos que heredó. Se dice que lo **sobrescribe** [1]-[3].

En nuestro ejemplo, sobrescribimos el método `cambiar` para que los objetos de tipo `SmartTV` muestren mensajes diferentes a los de tipo `Televisor` cuando cambiamos de canal.

```
class SmartTV(Televisor):
    def ver_youtube(self):
        print('Justin Bieber surprises Billie Eilish')

    def cambiar(self, canal):
        if canal == 3:  # TN
            print('BONADIO EN HD')
        elif canal == 6:  # C5N
            print('La rrrrealidad en HD')

        self.canal = canal

super_tele = SmartTV()
super_tele.cambiar(6)
```

La ejecución del código muestra

La rrrrealidad en HD

También podemos sobrescribir el constructor para agregar un atributo `conectado`, que indique si el `SmartTV` está o no conectado a internet. Sólo cuando `conectado` sea `True`, la función `ver_youtube` mostrará el mensaje.

```
class SmartTV(Televisor):
    def __init__(self, conectado=False):
        self.conectado = conectado

    def ver_youtube(self):
        if self.conectado:
            print('Justin Bieber surprises Billie Eilish')

    def cambiar(self, canal):
        if canal == 3: # TN
            print('BONADIO EN HD')
        elif canal == 6: # C5N
            print('La rrrrealidad en HD')

        self.canal = canal
```

El constructor, tratándose de una función como cualquier otra, puede tener argumentos obligatorios y opcionales, que se indican como siempre. Los argumentos del constructor se pasan a la hora de crear el objeto.

```
In [15]: super_tele = SmartTV(True)
In [16]: super_tele.ver_youtube()
Justin Bieber surprises Billie Eilish
In [17]: otra_super_tele = SmartTV()
In [18]: otra_super_tele.ver_youtube()
```

En el primer caso, el mensaje se ve, porque pasamos el argumento del constructor de la clase `SmartTV` como `True`. En el segundo no, porque pasamos implícitamente `False`, al omitir el argumento del constructor.

Si bien todo parece funcionar, hay un detalle problemático, que discutiremos inmediatamente a continuación.

### 5.4.2. Extensión

Ejecutemos el código escrito hasta ahora, y pongamos en la consola

```
In [19]: super_tele.canal
AttributeError: 'SmartTV' object has no attribute 'canal'
```

literalmente, el error nos dice que el objeto `super_tele`, de tipo `SmartTV`, *carece del atributo canal*. El problema es que, en la sección anterior, *sobreescrimos el constructor original* que definía el atributo.

Una forma de solucionar esto rápidamente sería copiar y pegar el contenido del constructor de `Televisor` al constructor de `SmartTV`. Acá se puede porque es una línea, y la clase hereda de una sola, pero no tendría sentido hacerlo si el código se vuelve más grande, complicado e inmanejable. Así que esta solución, más tosco parche que otra cosa, está totalmente descartada.

La solución que sí va es usar la función `super()`, que no tiene nada que ver con Superman ni con Dragon Ball, sino con la clase de la cual hereda la nuestra. En este caso, `SmartTV` hereda de `Televisor`, y consecuentemente, `Televisor` se denomina la **superclase** de `SmartTV` [1]-[3]. Un nombre solemne para una idea rigurosa: la relación de superclase es la *recíproca* de la herencia. Más formalmente, `S` es superclase de `C` si y sólo si `C` hereda de `S`.

Si dentro de un método de la clase llamamos a `super()`, podemos acceder al código del método respectivo definido en la superclase. Muy útil, porque ahora tenemos un camino directo al `__init__` del `Televisor`, desde adentro del `__init__` de la clase `SmartTV`:

```
class SmartTV(Televisor):
    def __init__(self, conectado=False):
        super().__init__()
        self.conectado = conectado

    def ver_youtube(self):
        if self.conectado:
            print('Justin Bieber surprises Billie Eilish')

    def cambiar(self, canal):
        if canal == 3: # TN
            print('BONADIO EN HD')
        elif canal == 6: # C5N
            print('La rrrrealidad en HD')

        self.canal = canal

super_tele = SmartTV()
print(super_tele.canal)
```

Ahora el código ejecutado muestra 1, como debe ser.

Acabamos de **extender** el constructor de nuestra clase `SmartTV` con el constructor de la superclase `Televisor`. Es decir que, además de otorgar a la

clase heredada de funcionalidad nueva (el atributo booleano `conectado`), el constructor de `SmartTV` también copia el código del constructor que habíamos definido para la clase `Televisor` (el atributo numérico `canal`).

Así como extendimos el constructor, podemos extender cualquiera de los métodos definidos en la clase heredada. Por ejemplo, el método `cambiar` de `SmartTV` podría ser este:

```
def cambiar(canal):
    if canal == 1079: # Animal Planet HD
        print('Los videos más graciosos de animalitos')

    super().cambiar(canal)
```

en cuyo caso, `cambiar(3)` y `cambiar(6)` seguirían mostrando los mismos mensajes que los objetos de tipo `Televisor`, con el añadido de `Los videos más graciosos de animalitos` para el nuevo canal definido en `SmartTV`.

## 5.5. Comentario final

Hay libros enteros dedicados a objetos [3]. Este capítulo debería ser considerado una síntesis, cuyo fin es dejar bases claras que puedan aplicarse. Además de los libros que aparecen en las **Referencias**, cuya lectura recomendamos, hay una descripción profunda del asunto acá:

<https://docs.python.org/3/tutorial/classes.html>

incluso siendo algo técnica, no deja de ser clara. De todas formas, la idea clave de este capítulo es el hecho de que la **orientación a objetos** no es más que una **forma de organizar datos**. Cuando hayan entendido eso, consideren el tema completamente comprendido, porque será cierto en lo que respecta a los usos de este libro.

## Referencias

- [1] M. Lutz, *Learning Python*, 5.<sup>a</sup> ed. United States of America: O'Reilly Media, 2013 (vid. págs. 94, 95, 97, 98, 100).
- [2] J. V. Guttag, *Introduction to computation and programming using Python*, 2.<sup>a</sup> ed. United States of America: Massachusetts Institute of Technology Press, 2016 (vid. págs. 94, 95, 98, 100).
- [3] D. Phillips, *Python 3 Object-oriented Programming*, 2.<sup>a</sup> ed. United Kingdom: Packt Publishing, 2015 (vid. págs. 95, 97, 98, 100, 101).

# Capítulo 6

## Interfaz gráfica de usuario

En los capítulos anteriores no tuvimos en cuenta al usuario. La máxima interacción que ofrecimos fueron las funciones `print` e `input`.

Los programas que usamos comúnmente a diario interactúan con el usuario de otra forma: por medio de una **interfaz gráfica**. En este capítulo desarrollaremos nuestras propias interfaces gráficas con PyQt5, un poderoso módulo que, afortunadamente, ya viene incluido en anaconda por defecto.

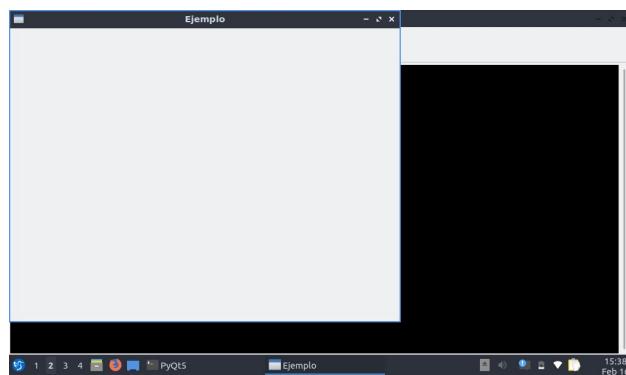
Usaremos cosas sobre objetos, así que conviene leer el Capítulo 5 primero.

### 6.1. La ventana

Arrancamos por la interfaz más sencilla posible: una ventana vacía.

```
from PyQt5.QtWidgets import QApplication, QWidget

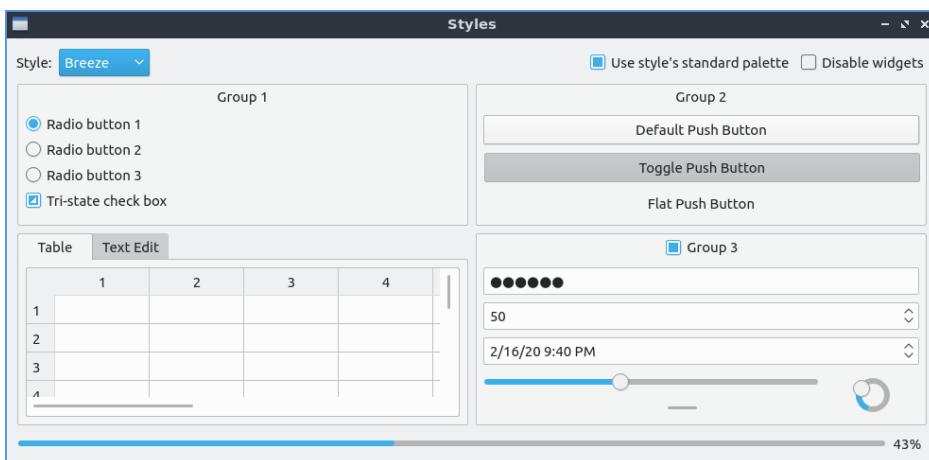
app = QApplication([])
ventana = QWidget(windowTitle='Ejemplo')
ventana.show()
app.exec_()
```



El objeto `QApplication` corresponde al programa per se. Tiene un argumento obligatorio, al que pasaremos una lista vacía en los ejemplos de este capítulo. Al declarar un objeto de tipo `QWidget`, PyQt5 guarda por defecto una ventana en la memoria, que cargamos y visualizamos con el método `show`. Por otro lado, el argumento `windowTitle` del constructor establece, como su nombre lo indica, el texto en la barra de título de la ventana.

## 6.2. Componentes

Si bien usamos `QWidget` para crear una ventana, “widget” se refiere, en realidad, a cualquiera de los elementos que se presentan al usuario para interactuar. Los “widgets”, o componentes, pueden ser botones, casillas de texto, imágenes, etc. El módulo PyQt5 trae estos, entre muchos otros componentes incorporados, algunos de los cuales se ven en la siguiente imagen.



Hagamos un recuadro que contenga texto y un botón. Al apretar un botón, mostraremos un texto en la consola.

Pero antes, algunas cosas que conviene saber primero:

1. Agregamos texto definiendo etiquetas con la clase `QLabel`.
2. Los botones son objetos de clase `QPushButton`.
3. Cuando hay más de un componente en una ventana, es necesario agruparlos en un contenedor, como por ejemplo, una caja. `QVBoxLayout` agrupa componentes en forma vertical, mientras que `QHBoxLayout` los agrupa en forma horizontal.

Usaremos `QVBoxLayout` para este ejemplo, es decir, el texto estará por encima del botón.

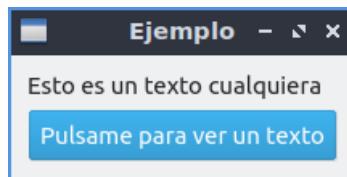
```
from PyQt5.QtWidgets import ( QApplication, QWidget,
    QVBoxLayout, QLabel, QPushButton)

app = QApplication([])

ventana = QWidget(windowTitle='Ejemplo')
caja = QVBoxLayout()
texto = QLabel('Esto es un texto cualquiera')
boton = QPushButton('Pulsame para ver un texto')

caja.addWidget(texto)
caja.addWidget(boton)

ventana.setLayout(caja)
ventana.show()
app.exec_()
```

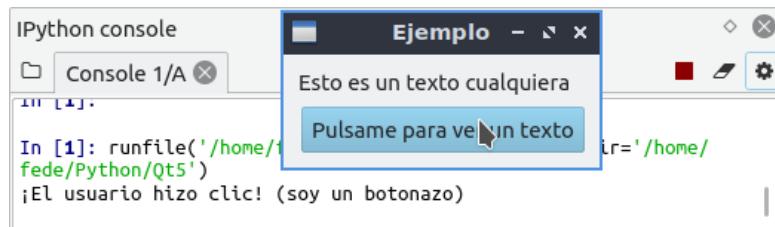


El botón no hace nada cuando lo pulsamos. Podemos agregarle un comando mediante una función

```
def clic():
    print('¡El usuario hizo clic! (soy un botonazo)')

y añadiendo bajo la declaración del botón

boton.clicked.connect(clic)
```

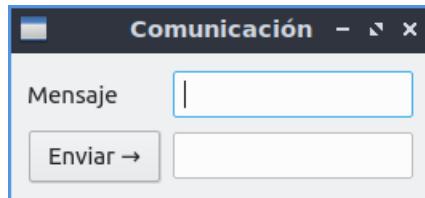


¡es tan fácil como se ve!

Acá utilizamos una función (`clic`) como parámetro de otra (`connect`). Este es un buen ejemplo de **función de alto orden**, es decir, una función que **recibe otra función como argumento**. Este concepto fue adelantado en la sección 1.10.4.

## 6.3. Información compartida

Enviemos un mensaje de un componente a otro.



Para un arreglo de componentes como el de la foto, podríamos usar cajas verticales dentro de cajas horizontales, o cajas horizontales dentro de cajas verticales, pero es innecesariamente complicado frente a una mejor alternativa: `QGridLayout`. Esta clase permite ordenar los componentes en una grilla, especificando la posición mediante filas y columnas en el método `addWidget`.

Para declarar las casillas de texto, usamos `QLineEdit`. Así:

```
from PyQt5.QtWidgets import QApplication, QWidget,
    QGridLayout, QLabel, QLineEdit, QPushButton)

app = QApplication([])
ventana = QWidget(windowTitle='Comunicación')
grilla = QGridLayout()

etiqueta = QLabel('Mensaje')
emisor = QLineEdit()
receptor = QLineEdit(readOnly=True)
boton = QPushButton('Enviar →')

grilla.addWidget(etiqueta, 0, 0)
grilla.addWidget(emisor, 0, 1)
grilla.addWidget(boton, 1, 0)
grilla.addWidget(receptor, 1, 1)
ventana.setLayout(grilla)
ventana.show()
app.exec_()
```

Notemos que sólo tiene sentido editar la casilla superior, porque es la que emite el mensaje. La casilla inferior sólo recibe el mensaje y lo muestra, así que no debería permitir edición por parte del usuario. Establecemos, para eso, el argumento booleano opcional `readOnly` en `True`.

La idea es que, al apretar el `boton`, el texto que contiene el `emisor` sea enviado al `receptor`. Obtenemos el texto del `emisor` con el método `text()`,

y establecemos el texto del receptor con `setText()`. Teniendo en cuenta esto, el código completo nos queda

```
from PyQt5.QtWidgets import ( QApplication, QWidget,
    QGridLayout, QLabel, QLineEdit, QPushButton)

def enviar_mensaje():
    receptor.setText(emisor.text())

app = QApplication([])
ventana = QWidget(windowTitle='Comunicación')
grilla = QGridLayout()

etiqueta = QLabel('Mensaje')
emisor = QLineEdit()
receptor = QLineEdit(readOnly=True)
boton = QPushButton('Enviar →')
boton.clicked.connect(enviar_mensaje)

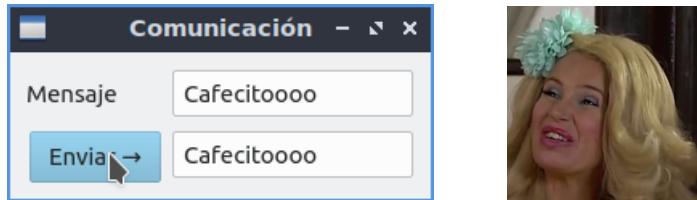
grilla.addWidget(etiqueta, 0, 0)
grilla.addWidget(emisor, 0, 1)
grilla.addWidget(boton, 1, 0)
grilla.addWidget(receptor, 1, 1)
ventana.setLayout(grilla)
ventana.show()
app.exec_()
```

NOTA: en `spyder`, es posible que el código no pueda ejecutarse más de una vez, porque a `PyQt` no le gusta tener que manejar dos instancias de `QApplication` en el mismo programa, y `spyder` mantiene disponible la primera `QApplication` creada. La solución a este problema es sustituir

```
app = QApplication([])
por
app = QApplication.instance()
if app is None:
    app = QApplication([])
```

cosa que implementaremos desde un principio en los códigos que siguen para ahorrarnos el problema.

Si bien el código funciona



es confuso que `enviar_mensaje` use los objetos `emisor` y `receptor` sin tenerlos siquiera como argumentos. En una interfaz sencilla como esta, esto no parece grave, pero lo será si nuestro programa cobra mayores dimensiones. En general, **no suele considerarse buena práctica trabajar con variables globales** por la innecesaria dificultad que añaden al código para leerlo.

**SOLUCIÓN: usar herencia.**

```
from PyQt5.QtWidgets import ( QApplication, QWidget,
    QGridLayout, QLabel, QLineEdit, QPushButton)

class VentanaMensaje(QWidget):
    def __init__(self):
        super().__init__(windowTitle='Comunicación')
        grilla = QGridLayout()
        etiqueta = QLabel('Mensaje')
        boton = QPushButton('Enviar →')
        boton.clicked.connect(self.enviar_mensaje)

        self.emisor = QLineEdit()
        self.receptor = QLineEdit(readOnly=True)

        grilla.addWidget(etiqueta, 0, 0)
        grilla.addWidget(boton, 1, 0)
        grilla.addWidget(self.emisor, 0, 1)
        grilla.addWidget(self.receptor, 1, 1)

        self.setLayout(grilla)
        self.show()

    def enviar_mensaje(self):
        self.receptor.setText(self.emisor.text())

app = QApplication.instance()
```

```
if app is None:
    app = QApplication([])

ventana = VentanaMensaje()
app.exec_()
```

Hace lo mismo que el código anterior, pero **está mejor ordenado: toda la información compartida queda dentro del objeto.**

Anteponemos `self` a las métodos heredados de la clase `QWidget` (como por ejemplo `setLayout`), así como también a los componentes extra que nos interesa tener disponibles desde cualquier método de la clase. En este caso, tales componentes son `emisor` y `receptor`, porque `enviar_mensaje` usa ambos.

## 6.4. Componente personalizado

Armemos un botón con un determinado color de fondo inicial, que pueda cambiarse al presionarlo. Para eso:

1. el nuevo botón tendrá una clase heredada de `QPushButton`;
2. el color de fondo inicial será un *argumento del constructor*;
3. habrá un método para abrir un cuadro de diálogo que permita elegir un color de fondo nuevo.

PyQt5 ya viene con una función para cargar el cuadro de diálogo con los colores: `QColorDialog` del submódulo `QtWidgets`.

```
from PyQt5.QtWidgets import QApplication, QColorDialog

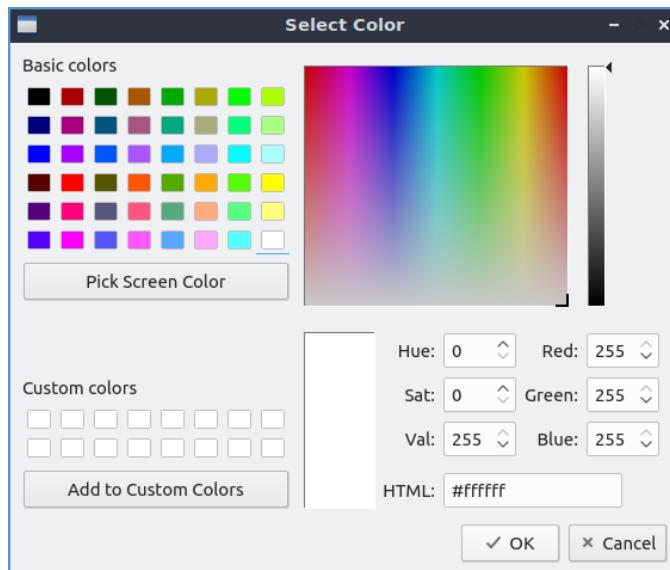
app = QApplication.instance()
if app is None:
    app = QApplication([])

color = QColorDialog.getColor()

print(color.isValid())

if color.isValid():
    print(color.name())
```

Al ejecutar el código, se muestra una ventana como la siguiente.



De acuerdo a las acciones que se tomen, hay dos posibilidades:

- a) pulsar **Cancelar** o
- b) elegir un color y pulsar **Aceptar**.

En el primer caso, `color.isValid()` devuelve `False`, y en la Terminal sólo se ve `False`. En el segundo, `color.isValid()` es `True`, y la consola nos muestra también, en consecuencia, el contenido de la variable `color.name()`, con el código HTML del color seleccionado.

Volvamos a las condiciones 1, 2 y 3, que determinan el estilo de código del botón personalizado que faremos. Llamaremos a la clase `BotonColor`. La condición 1 se implementa mediante la sintaxis de la sección 5.4:

```
class BotonColor(QPushButton):
```

La condición 2 establece que el color inicial es un argumento del constructor, lo cual es fácil de implementar:

```
def __init__(self, color):
```

Por último, la condición 3 requiere definir un método

```
def elegir_color(self):
    color = QColorDialog.getColor()
```

que además, contiene el código necesario para cambiar el color del objeto. El cambio de color de un `QWidget` en PyQt5 no es una cosa muy directa,

requiere cambiar lo que se llama la *hoja de estilos* del componente. No es algo excesivamente complicado, pero requiere algo de tiempo aprender la sintaxis. En consecuencia, nos limitaremos en este ejemplo a definir el siguiente método para cambiar el color del botón.

```
def cambiar_color(self, color):
    self.setStyleSheet('QPushButton {{ background: {color}; '
                      'width: 3em; height: 3em; '
                      'margin: 0.5ex; border-radius: 0.5ex; '
                      'border: 0.5ex solid {color}; }} '
                      'QPushButton:hover {{ margin: 0ex }} '
                      ''.format(color=color))
```

El cambio en la hoja de estilos no es sólo de color, sino también de tamaño: el botón debe agrandarse cuando pasemos el cursor por encima. Si quieren sumergirse en el mundo de las hojas de estilo de Qt5:

<https://doc.qt.io/qt-5stylesheet-customizing.html>

Entonces procedemos:

```
class BotonColor(QPushButton):
    def __init__(self, color):
        super().__init__()
        self.cambiar_color(color)
        self.clicked.connect(self.elegir_color)

    def cambiar_color(self, color):
        self.setStyleSheet('QPushButton {{ background: {color}; '
                          'width: 3em; height: 3em; '
                          'margin: 0.5ex; border-radius: 0.5ex; '
                          'border: 0.5ex solid {color}; }} '
                          'QPushButton:hover {{ margin: 0ex }} '
                          ''.format(color=color))

    def elegir_color(self):
        color = QColorDialog.getColor()
        if color.isValid():
            self.cambiar_color(color.name())
```

donde lo que hicimos fue:

- llamar una vez al método `cambiar_color` en el constructor, pasando el `color` como argumento;

- conectar el evento `clicked` del botón con el método `elegir_color`;
  - llamar al método `cambiar_color` si `color.isValid()` es True.

Obviamente, la definición de `BotonColor` requiere las clases `QPushButton` y `QColorDialog` de `PyQt5.QtWidgets`. Pondremos todo esto en el código completo, al final de la sección.

Ahora vamos a armar una ventana como la que sigue.



Para ello, debemos:

1. crear una `QApplication`;
  2. crear una ventana, mediante el objeto `QWidget`;
  3. crear tres objetos de clase `BotonColor` pasando distintos colores al constructor (en PyQt5 podemos pasar colores por nombre, para este ejemplo, serán `red`, `yellow` y `blue`);
  4. crear una caja contenedora horizontal (`QHBoxLayout`) para los tres botones, y vincularlo todo mediante `addWidget` y `setLayout`.

Se muestra a continuación el código completo.

```

'QPushButton:hover {{ margin: 0ex }} '
''.format(color=color))

def elegir_color(self):
    color = QColorDialog.getColor()
    if color.isValid():
        self.cambiar_color(color.name())

app = QApplication.instance()
if app is None:
    app = QApplication([])

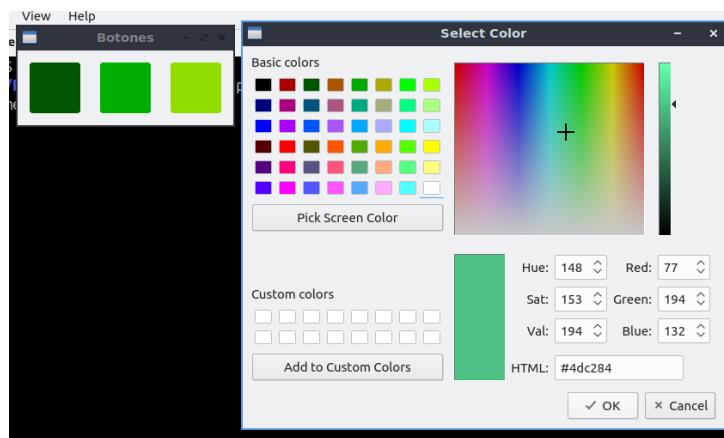
ventana = QWidget(windowTitle='Botones')
caja = QBoxLayout(spacing=15)
boton1 = BotonColor('red')
boton2 = BotonColor('yellow')
boton3 = BotonColor('blue')

caja.addWidget(boton1)
caja.addWidget(boton2)
caja.addWidget(boton3)

ventana.setLayout(caja)
ventana.show()
app.exec_()

```

Al presionar cada uno de los tres botones de colores, aparece una diálogo, con el cual es posible seleccionar un nuevo color y asignárselo al botón.



## 6.5. Incrustación de gráficos

Hay una forma de aprovechar todo lo que aprendimos en el capítulo 3 sobre `matplotlib` para hacer gráficos y mostrarlos directamente en nuestra interfaz: mediante la clase `FigureCanvas`.

`FigureCanvas` es un componente (de hecho, la clase `FigureCanvas` hereda de `QWidget`) cuyo constructor recibe *figuras* de `matplotlib` y las muestra en pantalla. Las figuras de `matplotlib` son objetos de tipo `Figure`, la cual es una clase definida en el submódulo `figure`. Esta no es la única clase que define `matplotlib` para usar, ni el único submódulo. Toda la información correspondiente a estas cosas está en <https://matplotlib.org/api/#modules>.

Crearemos a continuación una figura con un gráfico simple y lo incrustaremos en un “canvas”.

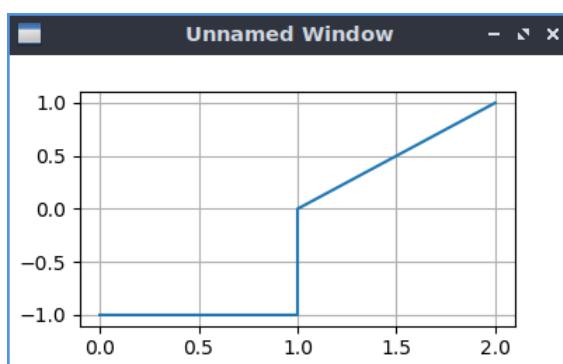
```
from PyQt5.QtWidgets import QApplication
from matplotlib.figure import Figure
from matplotlib.backends.backend_qt5agg import FigureCanvas

app = QApplication.instance()
if app is None:
    app = QApplication([])

figure = Figure()
axes = figure.subplots()
axes.plot([0, 1, 1, 2], [-1, -1, 0, 1])
axes.grid(True)

canvas = FigureCanvas(figure)
canvas.show()
app.exec_()
```

Una vez ejecutado el código, se abrirá una ventana como esta:



Hay dos diferencias importantes respecto a los métodos usados en el capítulo 3 para crear gráficos y modificar su aspecto.

1. El objeto `Figure` se crea directamente con su constructor, provisto por el submódulo `figure`, en vez de utilizar la función `figure` de `pyplot`.
2. Las funciones para graficar y cambiar aspecto (`plot`, `grid`) se llaman directamente como métodos del objeto `axes`, en vez de usar las funciones correspondientes de `pyplot`.

Hay algunas otras diferencias de nomenclatura entre métodos del objeto `axes` y funciones de `pyplot`, por ejemplo, `axes.set_xlabel` en vez de `plt.xlabel`, `axes.set_title` en vez de `plt.title`, etc. Salvo por estas pequeñas diferencias, el manejo de gráficos es prácticamente idéntico al que vimos en el capítulo 3.

### 6.5.1. Un gráfico interactivo

Agreguemos un botón para cambiar el color de línea del gráfico. Para ello, vamos a usar la clase `BotonColor` que definimos en la sección 6.4, con algunos añadidos útiles. La idea es aprovechar la clase `BotonColor` cambiándola lo menos posible, y la forma correcta de hacer eso es mediante *herencia* (así, de paso, ponemos en práctica la idea).

```
class BotonColorInteractivo(BotonColor):
    def __init__(self, line):
        super().__init__(line.get_color())
        self.line = line

    def elegir_color(self):
        color = QColorDialog.getColor()
        if color.isValid():
            self.cambiar_color(color.name())
            self.line.set_color(color.getRgbF())
            self.line.figure.canvas.draw()
```

Algunas observaciones importantes.

1. El constructor recibe *la línea del gráfico* a la cual se quiere cambiarle el color. La línea se crea con el método `plot` del objeto `axes`.

Es importante destacar que, como los objetos son *mutables*, **cualquier método que llamemos desde una referencia afectará al objeto original**. Lo mismo pasaba con las listas en la sección 1.8.2, página 22.

2. El método `elegir_color` modifica el color de la línea (además de abrir el cuadro de diálogo para elegir el color, obvio).
3. El método `getColor` de la clase `QColorDialog` devuelve el objeto `color`, de clase `QColor`. Pero el método `set_color` del objeto `line` requiere una lista que indique el nivel de rojo, verde y azul mediante números de tipo `float`. Esta lista es precisamente lo que devuelve el método `getRgbF` del objeto `color`, y por eso se usa.
4. El objeto `line` tiene un atributo `figure` que referencia a la figura en la que se encuentra. Este objeto, a su vez, tiene un atributo `canvas` que terminará referenciando al `FigureCanvas` del gráfico. El método `draw` redibuja todo el contenido del `canvas` y sin él, sería imposible ver los cambios producidos en el gráfico.

Ahora usaremos esta clase y `FigureCanvas` para crear dos componentes que compartirán un caja vertical (`QVBoxLayout`) en una misma ventana (`QWidget`). El código de la aplicación es muy sencillo.

```
ventana = QWidget(windowTitle='Gráfico interactivo')
caja = QVBoxLayout()

figure = Figure()
axes = figure.subplots()
line, = axes.plot([0, 1, 1, 2], [-1, -1, 0, 1])
axes.grid(True)

canvas = FigureCanvas(figure)
boton = BotonColorInteractivo(line)

caja.addWidget(canvas)
caja.addWidget(boton)

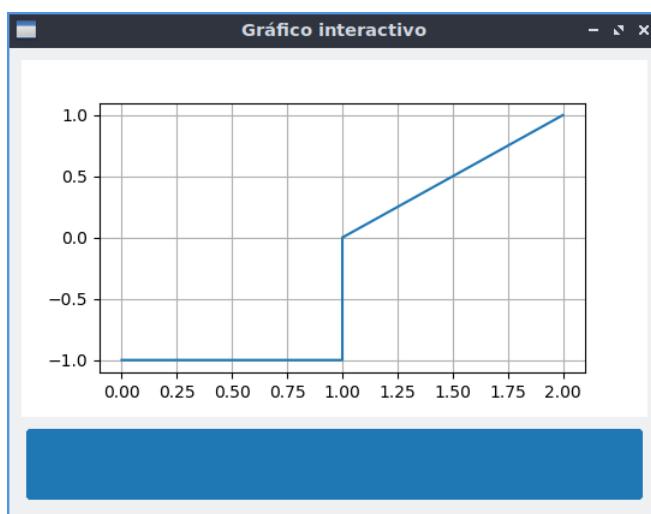
ventana.setLayout(caja)
ventana.show()

app.exec_()
```

*Nota:* la asignación `line, = ...` indica que `line` será el primer y único elemento de la lista de valores que resulte del método `plot`. Una forma parecida de hacer esto es

```
line = axes.plot([0, 1, 1, 2], [-1, -1, 0, 1])[0]
```

Si todo salió bien (es probable que tengan que importar algún que otro componente desde `PyQt5.QtWidgets`) debería aparecer esto:



Al presionar el botón, aparece el cuadro de diálogo, y al cambiar el color, tanto el botón como la línea tomarán el color elegido.

## 6.6. Diseño visual

Si bien el diseño de una interfaz gráfica por código es una práctica muy buena y ordenada, a veces queremos tener funcionando más rápidamente una interfaz simple. Anaconda trae dos herramientas dedicadas a tal fin:

1. `designer`, un programa gráfico para crear y distribuir componentes en una o más ventanas;
2. `pyuic5`, para generar automáticamente código a partir de la interfaz diseñada visualmente.

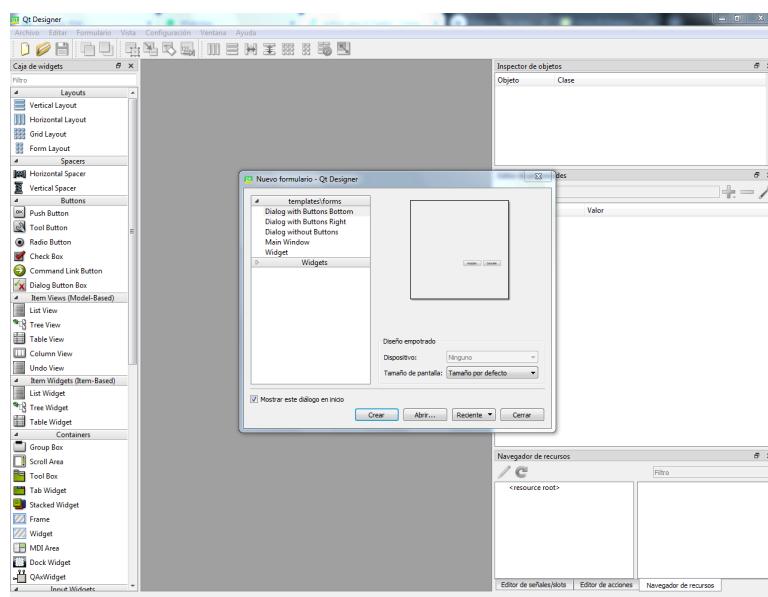
El flujo de trabajo suele reducirse a tres pasos bien determinados.

1. Primero, diseñar las ventanas de nuestro programa, que se guardan en archivos con extensión `ui`.
2. El archivo `ui` se da como argumento a `pyuic5` para que genere un código base de `python`, con las clases, atributos y métodos necesarios para el desarrollo del programa.
3. El código base se importa como módulo desde un script principal, encargado de crear una `QApplication` y crear los objetos correspondientes a las clases generadas por `pyuic5`.

Evidentemente, los pasos son intercambiables. Cabe señalar que **cualquier modificación del archivo ui desde designer no se verá reflejada en el código hasta que no ejecutemos pyuic5**.

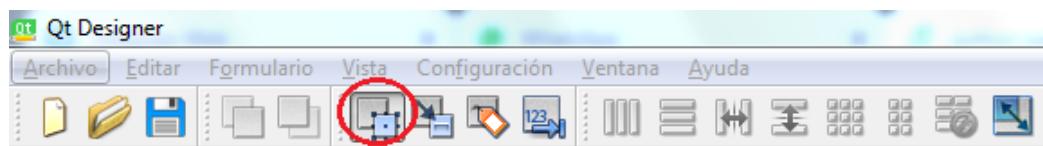
Dicho esto, abramos **designer**. Para ello:

1. Abrir Anaconda Prompt;
2. escribir **designer**;
3. pulsar **Enter** y se abre algo como esto:

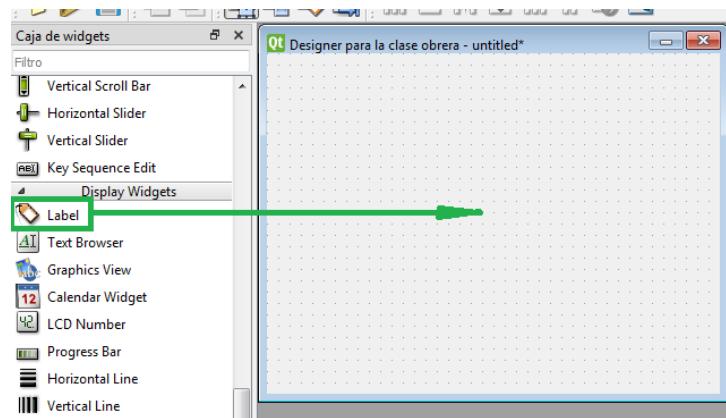


**Designer** abre automáticamente una ventana con plantillas y opciones para crear un archivo ui. Seleccionemos la opción **Widget** por ahora y pulsemos **Crear**, dejando el resto de las opciones como vienen por defecto. Esto crea una ventana, cuyas propiedades podemos ver y modificar al costado derecho en el **Editor de propiedades**. En el **Filtro**, escribamos **windowTitle** y cambiemos el valor por defecto (**Form**) a **Designer** para la clase obrera.

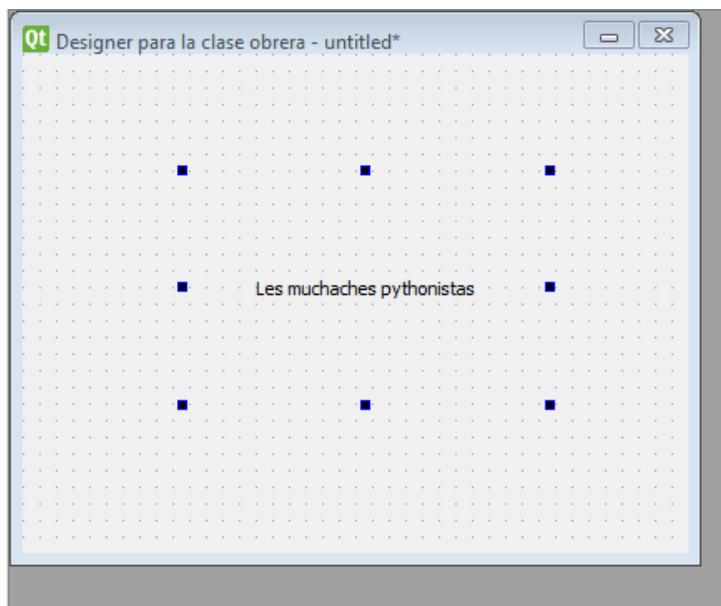
Agreguemos un texto. Para ello, debemos estar en el *modo de edición*, al cual podemos acceder desde **Editar > Editar widgets**, pulsando la tecla **F3**, o desde este botón:



Ahora buscamos Label en la Caja de widgets, ubicada por defecto a la izquierda del Designer, y la arrastramos hasta la ventana.



Vamos al Editor de propiedades y cambiamos la propiedad `text` por algo como `Les muchaches pythonistas` (todes unides triunfaremos) y establecemos la alineación horizontal en `AlinearCentroH` (la propiedad es `alignment`, campo `Horizontal`). Agrandemos el texto moviendo con el mouse los cuadrados azules que aparecen alrededor del componente.



El texto del Label también puede cambiarse haciendo doble clic o presionando la tecla F2 cuando el Label está seleccionado.

Previsualizamos la ventana yendo a `Formulario > Vista previa...` o con la combinación simultánea de teclas `Ctrl R`.

Guardemos la ventana como `ventanasoberana.ui` en algún directorio al que sea fácil acceder desde Anaconda Prompt (por ejemplo, nuestra carpeta de usuario o Documentos). Acto seguido, vamos al Prompt y ponemos `cd` y el directorio en donde hayamos guardado el ui (por ejemplo, `cd Documents`) y corremos `pyuic5 ventanasoberana.ui -o ui_ventanasoberana.py`, lo que genera el código `ui_ventanasoberana.py` con la clase `Ui_Form` lista para ser implementada en nuestra aplicación. La forma más sencilla de hacerlo es:

```
from PyQt5.QtWidgets import QApplication, QWidget
from ui_ventanasoberana import Ui_Form

app = QApplication.instance()
if app is None:
    app = QApplication([])

ventana = QWidget()
ui = Ui_Form()
ui.setupUi(ventana)

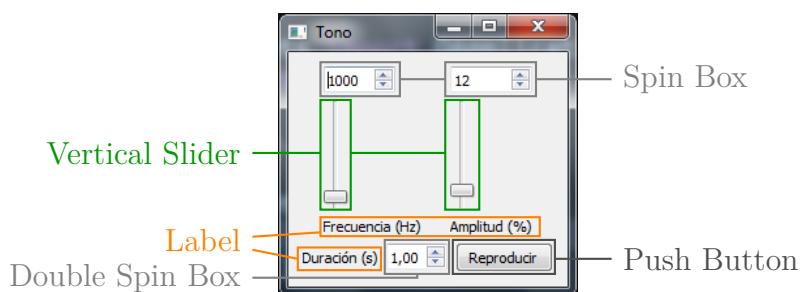
ventana.show()
app.exec_()
```

Este código tiene que estar en el mismo lugar que `ui_ventanasoberana.py`.

Si hicimos todo bien, tiene que aparecer la ventana tal cual la diseñamos.

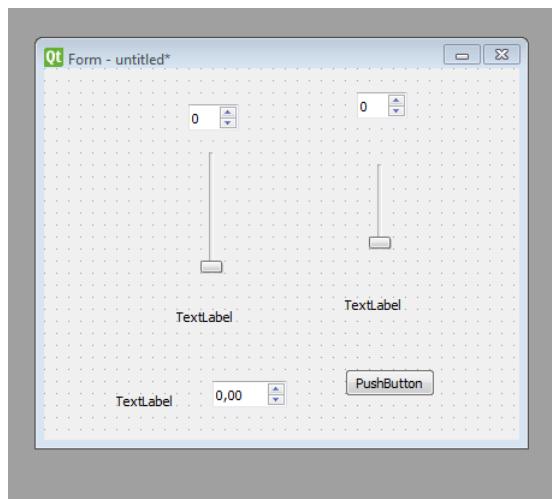
### 6.6.1. Generador de tonos puros

Ahora que incorporamos el flujo de trabajo con el diseñador visual, aprovechemos para poner en práctica las ideas de los capítulos 2 y 5, junto con las de este capítulo. Vamos a crear una ventana como la siguiente, donde se incluyen los nombres de los componentes tal cual figuran en `designer`.

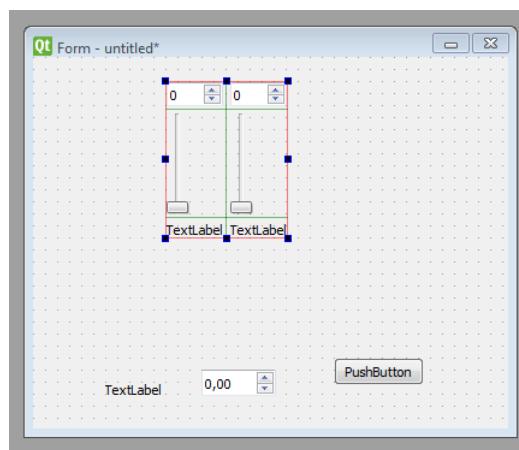


La idea es usar los contenedores que hagan falta, como para distribuir correctamente los componentes y permitir su redistribución al redimensionar la ventana. **Designer** cuenta con las cajas contenedoras y las grillas que tratamos en la sección 6.2, aparecen como **Layouts** en la Caja de widgets, así que eso no será problema.

Creamos un formulario y agregamos los nueve componentes de la figura. No hace falta preocuparse por la alineación y los contenedores aún, ya que de eso nos ocuparemos en breve. Debería quedar algo medio falopa como esto:



Ahora seleccionamos los primeros 6 componentes, contando de izquierda a derecha, de arriba para abajo, y vamos a **Formulario > Distribución en cuadrícula** (o sino, botón derecho, **Distribución > Distribución en cuadrícula**, o **Ctrl 5** en simultáneo). Esto los acomoda en una grilla:

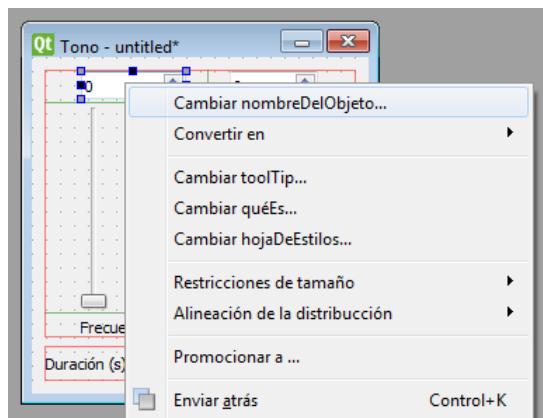


Si algo sale mal, podemos “romper” la distribución mediante la opción correspondiente en el menú, o pulsando la combinación de teclas **Ctrl+0**.

Seguimos un proceso análogo para los tres componentes restantes, pero esta vez, les damos distribución horizontal.

Aún falta un paso importante: *combinar los contenedores en uno solo*. Hay varias formas de hacer esto, una posible es seleccionar el espacio vacío de la ventana, e ir a **Formulario > Distribución vertical** (también se puede hacer botón derecho y elegir la distribución, como antes, o sino, se deselecciona todo y se hace **Ctrl+2**). Acto seguido, damos un título a la ventana (por ejemplo, **Tono** o **Generador de tonos puros**) y cambiamos el texto de los **Labels** y el **PushButton** para que coincidan con el diseño mostrado al principio de la sección (podemos darles doble clic para editar más rápido). Además, la ventana resultante puede ser redimensionada a una más chica, aunque esto ya es cuestión de gusto.

Con el objetivo de facilitar la programación, daremos nombres razonables a los **Spin Boxes**. De otro modo, la interacción con estos componentes requerirá usar los nombres por defecto: **spinBox**, **spinBox\_2**, ... no muy representativos de lo que hace cada cosa en realidad. La propiedad que contiene el nombre del objeto se llama, literalmente, **objectName**. En vez de cambiar esta propiedad, se puede hacer clic derecho sobre el objeto y elegir la opción **Cambiar nombre del objeto** (sobre los **Spin Box** también se puede hacer doble clic). En todo caso, llamémoslos **frequency**, **amplitude** y **duration**.



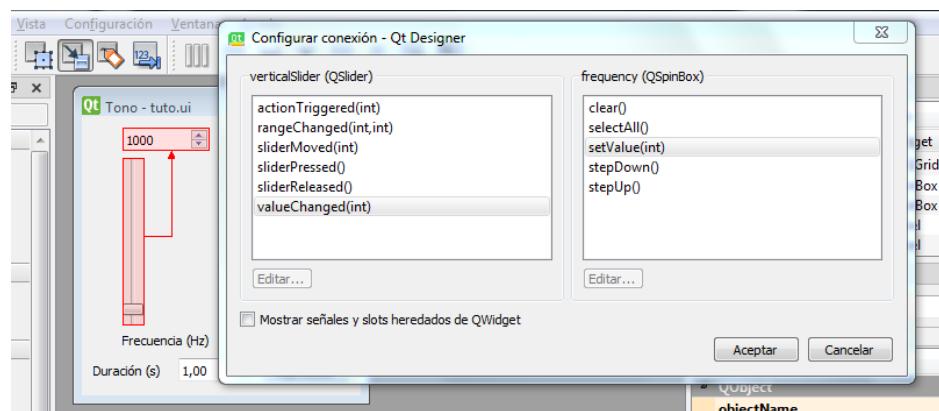
Ahora debemos limitar el rango de valores de los **Spin Boxes** y los deslizadores. Ponemos la propiedad **minimum** en 20 para **frequency**, y **maximum** en 16000. Al resto, **amplitude** y **duration**, le cambiamos el máximo por 100 y 3, respectivamente, y disminuimos el **singleStep** de **duration** a 0,01. Pongamos, por último, los **value** en lugares razonables, por ejemplo, 1000 para la frecuencia, 12 para la amplitud y 1 para la duración.

Con esto dejamos todo listo para el siguiente paso.

### 6.6.2. Conexión de señales entre componentes

En **designer** podemos conectar fácilmente los componentes **Spin Box** con los de tipo **Vertical Slider** para "sincronizarlos". Esto permite actualizar instantáneamente el valor de uno cuando se modifica el otro. Para ello, debemos entrar en el modo de *edición de señales/slots*, al que se accede yendo a **Editar > Editar señales/slots**, pulsando el ícono de la barra de herramientas, o la tecla F4.

Deslizamos el cursor desde cada uno de los **Vertical Slider** hacia los **Spin Box** correspondientes. Aparece la ventana para configurar conexiones, seleccionamos **valueChanged(int)** y **setValue(int)**.



Luego se repite el proceso en sentido opuesto, es decir, se conecta cada **Spin Box** con el **Vertical Slider** que le corresponda, seleccionando los mismo métodos que antes, a saber, **setValue(int)** y **valueChanged(int)**.

Guardamos la ventana (yo la llamé **generador.ui**) y pasamos a la parte jugosa del asunto.

### 6.6.3. Lógica del generador

Usamos **pyuic5** para generar la clase **Ui\_Form** (si es que la ventana sigue teniendo el nombre por defecto, esto es, si no cambiaron su nombre de objeto en **designer**). En mi caso, el comando que corro en el **Anaconda Prompt** es **pyuic5 generador.ui -o ui\_generador.py** desde la carpeta en donde se encuentra el archivo **generador.ui**, obvio (sino, no funciona).

Creemos un archivo en el mismo directorio (el nombre da igual) y armemos el código principal. Esta vez, dejaremos listo un objeto de clase **VentanaGenerador** que herede de **QWidget** y contenga un objeto de clase **Ui\_Form** para acceder a los componentes creados con **designer**.

```

from PyQt5.QtWidgets import QApplication, QWidget
from ui_generador import Ui_Form

class VentanaGenerador(QWidget):
    def __init__(self):
        super().__init__()
        ui = Ui_Form()
        ui.setupUi(self)
        self.ui = ui

    self.show()

app = QApplication.instance()
if app is None:
    app = QApplication([])

ventana = VentanaGenerador()
app.exec_()

```

Podemos crear un método `play` que se llame al pulsar el botón, cuyo nombre es `pushButton` (si es que, insisto, dejaron el `objectName` como estaba por defecto en `designer`). Este método debe leer los números en los `Spin Boxes`, generar la sinusoidal a partir de eso, y reproducir la sinusoidal generada. Para ello, vamos a importar `numpy` y `sounddevice` como hicimos en el capítulo 2, y definimos el método `play` así:

```

def play(self):
    ui = self.ui
    f = ui.frequency.value()
    A = ui.amplitude.value() / 100
    T = ui.duration.value()
    fs = 44100
    tono = A * np.sin(2*np.pi*f*np.arange(int(T*fs))/fs)
    sd.play(tono, fs)

```

y ahora podemos conectar manualmente el `pushButton` con el método, añadiendo en `__init__`

```
ui.pushButton.clicked.connect(self.play)
```

con esto ya todo funciona de maravilla.

El código completo quedó así:

```

import numpy as np
import sounddevice as sd
from PyQt5.QtWidgets import QApplication, QWidget
from ui_generador import Ui_Form


class VentanaGenerador(QWidget):
    def __init__(self):
        super().__init__()
        ui = Ui_Form()
        ui.setupUi(self)
        ui.pushButton.clicked.connect(self.play)
        self.ui = ui

    def show():
        self.show()

    def play(self):
        ui = self.ui
        f = ui.frequency.value()
        A = ui.amplitude.value() / 100
        T = ui.duration.value()
        fs = 44100
        tono = A * np.sin(2*np.pi*f*np.arange(int(T*fs))/fs)
        sd.play(tono, fs)

app = QApplication.instance()
if app is None:
    app = QApplication([])

ventana = VentanaGenerador()
app.exec_()

```

## 6.7. Comentario final

Acabamos de estudiar una partícula de un Universo: PyQt5 es lo que se llama una adaptación (“binding”) de Qt, que es una plataforma súper potente de desarrollo de aplicaciones en general, y no sólo de interfaces gráficas. Cualquier cosa que les interese sobre esto, y quieran profundizar, pueden ver la (extensa) documentación de Qt, en <https://doc.qt.io/>.

## Capítulo 7

# Digitalización y audio en tiempo real

Hasta ahora, hemos hecho muchas operaciones interesantes utilizando Python como lo que es, esto es, un lenguaje para comunicar instrucciones a la computadora. Ayudándonos con librerías como `numpy`, `soundfile` y `sounddevice` hemos leído o grabado archivos de audio como vectores y hemos hecho diversos procesos en tiempo o en frecuencia con esos vectores. Todo ello, aunque es valioso e importante, no siempre se parece a la experiencia de trabajo de un usuario que utiliza una computadora para procesar señales de audio. Para un programa que grabe señales de audio, por ejemplo, es deseable que el usuario pueda escuchar lo que se está grabando mientras se ejecuta la música, antes de grabarlo completamente. De otro modo, debe esperar a realizar un registro completo antes de saber si, por ejemplo, el sistema está recortando la señal, o introduciendo ruido. Asimismo, si se está diseñando un efecto, el proceso de grabar la señal, aplicar el efecto y reproducir el vector se vuelve tedioso.

En este capítulo, nos proponemos un proyecto más ambicioso, esto es, el de procesar la señal *en tiempo real*, lo cual significa que el sistema debe responder “instantáneamente” al usuario, de modo que, al menos aparentemente, el procesamiento sea continuo: la señal de salida, ya procesada, parece generarse al mismo tiempo que la señal de entrada ingresa al procesador. Evidentemente, esto exige una gran velocidad de procesamiento, y nuestro enfoque hasta ahora debe ser modificado para satisfacer esta necesidad. Imaginemos, por ejemplo, un guitarrista rockero, sucio y desprolijo, que va a grabar una canción de tres minutos de duración, en la cual desea aplicar un efecto a la guitarra. Si trabajáramos con nuestro método actual, nuestro rockero desprolijo debería grabar su guitarra limpia, luego debería aplicar el efecto procesando las  $180 \times 44100 = 7.938.000$  muestras, lo

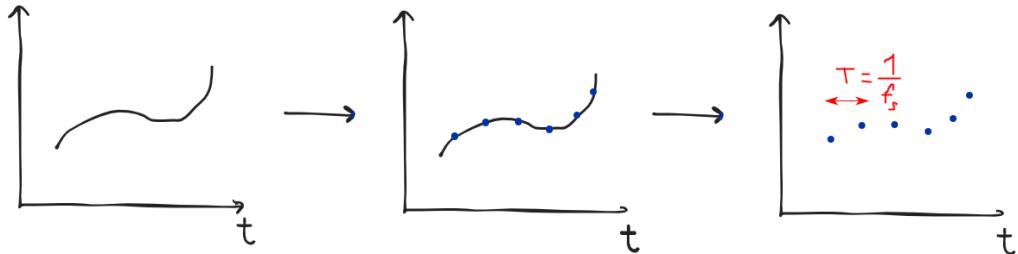
cual dependiendo del efecto puede demorar varios segundos, y finalmente reproducir la señal procesada. Esta forma de trabajo, desde ya, está lejos de ser aceptable. A fines de realizar esta tarea, debemos aprovechar al máximo las posibilidades de nuestra computadora, e implementar un método más idóneo para la tarea. Para ello, repasaremos a un nivel bastante bajo el proceso que atraviesa una señal sonora que es digitalizada para ingresar a un sistema, y utilizaremos la librería PyAudio para implementar, a modo de ejemplo, un analizador de espectro en tiempo real, esto es, un código que recibe la señal de entrada del micrófono del sistema, y muestra el contenido en frecuencia de la señal mientras esta se desarrolla.

## 7.1. El proceso de digitalización

Las señales del mundo no son, así sin más, procesables por una computadora: una señal sonora es analógica y, por lo tanto, *continua*, lo cual implica que entre dos valores cualesquiera de la señal hay infinitos valores, lo cual implica que cualquier fracción de la señal analógica, si fuese procesada por la computadora, insumiría un tiempo de procesamiento infinito. Para poder ser procesada por la computadora, la señal debe ser *digitalizada*, lo cual implica que pasa por tres etapas: **muestreo, cuantificación y codificación**.

### 7.1.1. Muestreo

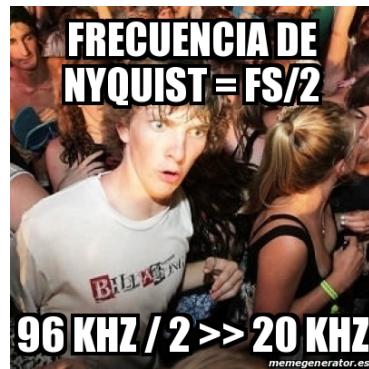
La primera etapa de la digitalización consiste en tomar muestras de la señal en tiempo, de modo que sólo se tengan muestras aisladas, discretas de la misma, en instantes de tiempo específicos. La separación de muestras será el tiempo  $T = \frac{1}{f_s}$ , donde  $f_s$  es la frecuencia de muestreo, esto es, cuántas muestras se toman por cada segundo de señal.



Evidentemente, cuanto más grande sea la frecuencia de muestreo, más información de la señal temporal se tendrá. Una mirada ingenua podría hacer

pensar que, de este modo, cuánto más alta sea la frecuencia de muestreo mayor será la calidad de la señal, mas esta opinión es en general errónea. Un hecho muy conocido en la teoría del procesamiento de señales es que si la máxima frecuencia de interés en nuestra señal es  $f_{\max}$ , entonces es suficiente con que  $f_s$  sea al menos el doble de  $f_{\max}$ , esto es,  $f_s > 2f_{\max}$ . Este resultado es parte del *teorema del muestreo* o *teorema de Nyquist-Shannon*.

Si bien hay buenos motivos (tecnológicos, más que teóricamente necesarios) para que  $f_s$  sea un poco más grande de lo estrictamente necesario, la extraña opinión de quienes sostienen que debe muestrearse con la mayor frecuencia posible, llegando a extremos tale como tomar 192.000 o 384.000 muestras por segundo, es del todo infundada, y su extensión a lo largo de la “comunidad el audio” resulta bastante sorprendente<sup>1</sup>.



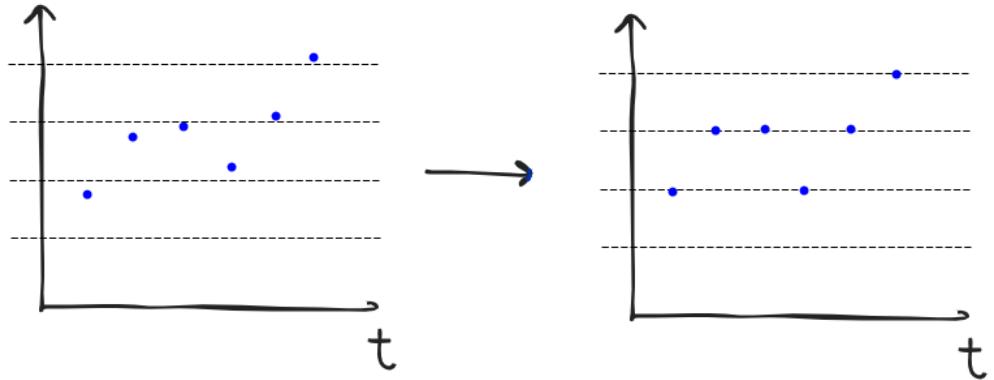
Hay mucha información concreta acerca del muestreo en la bibliografía, incluidas demostraciones completas del teorema de Nyquist-Shannon con sus sutilezas y particularidades, pero a fines de nuestro problema basta con comprender que, en este proceso, la señal continua es separada en muestras discretas en el tiempo.

### 7.1.2. Cuantificación

En la segunda parte del proceso, se realiza una segunda discretización de la señal, esta vez no en tiempo, sino en amplitud. Como la señal original era continua, cada una de las muestras discretas que nos han quedado podrían asumir *cualquier* valor, es decir, hay infinitos valores posibles para la amplitud de la señal. La **cuantificación digital** se encarga de llevar esos valores a un rango finito de posibilidades.

---

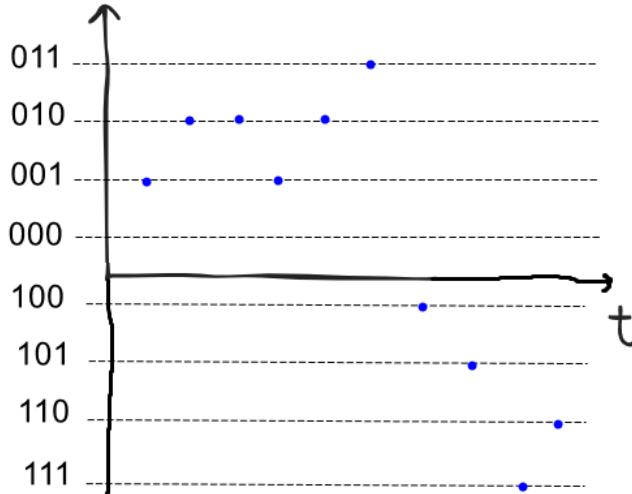
<sup>1</sup>En general se acepta que las señales de audio tienen una frecuencia máxima en el orden de los 20 kHz, aunque algunos estudios afirman que en algunos contextos podrían percibirse sonidos en el orden de los 24 o 25 kHz. Esto es discutido en el ejercicio 2 del Capítulo 4, pero no hay justificación para tomar frecuencias de muestreo tan exorbitantes.



De esta manera, cada muestra es *redondeada* al nivel más cercano que tenga, y ese nivel de cuantificación será su nuevo valor. Es importante notar que este redondeo *altera la señal de audio*, y por lo tanto la modifica. Por lo tanto, es claro que *cuántos más niveles se utilicen en la cuantificación, mayor resolución tendremos al digitalizar la señal*.

### 7.1.3. Codificación

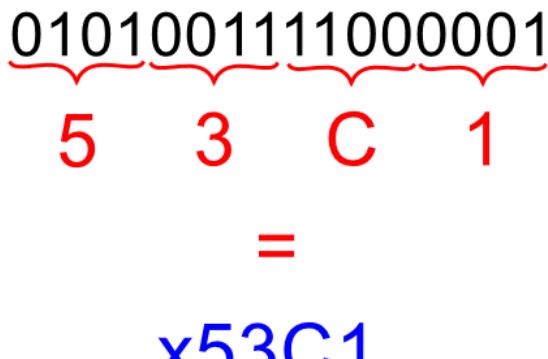
Lo más importante a entender, es que, para poder ser representado en la computadora, cada nivel de cuantificación se traduce en una cadena binaria de unos (1) y ceros (0), por ejemplo, de esta manera:



Nótese que, utilizando 3 dígitos entre unos y ceros (esto es, 3 *bits*), tenemos 8 niveles. En general, si tenemos  $N$  bits, tendremos  $2^N$  niveles de cuantificación, de modo que la mitad serán para representar valores negativos, y la mitad para representar valores positivos. Si se quiere representar el número

0, lo cual es habitual, se utilizará un nivel para eso. El audio digital, de esta manera, se encuentra en la computadora representado como cadenas de unos y ceros. Si, por ejemplo, se utilizan 16 bits en la cuantificación de una señal digitalizada, entonces se tienen  $2^{16} = 65536$  niveles posibles al digitalizarla, lo cual implica que se tienen, en forma aproximada, 32768 niveles para representar valores positivos y 32768 para valores negativos, y cada muestra de audio, en la computadora, se representará como una cadena de 16 unos y ceros. El primer dígito, en general, se utiliza para indicar **si un número es positivo o negativo, de suerte que los números que inician en 1 son negativos, y los que inician en 0 son positivos.** Como estas cadenas tan extensas deben ser segmentadas de alguna forma, el estándar es que esta información se maneje en paquetes de 8 bits, llamados *bytes*, y además, se utiliza la *representación hexadecimal*, que permite que cada grupo de 4 bits sea representado con un único símbolo, como vemos en la figura:

Binario	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F


  
x53C1

Un número precedido por “x” (o bien por “0x”) habitualmente designa un número en formato hexadecimal.

## 7.2. Un ejemplo en Python

Lo que acabamos de explicar es una pequeñísima punta de lo que pasa en el bajo nivel del proceso de digitalización, pero eso es lo que hacen nuestras placas de audio cuando conectamos un micrófono y registramos una señal. En

general, el trabajo en el bajo nivel del sistema suele ser tedioso y complicado, y exige bastante conocimiento de los protocolos de transmisión de la información, así como un alto conocimiento de algoritmos si se quiere procesar una señal sin llevarla a un equivalente vectorial. Veamos un ejemplo de esto en Python, intentando bajar estos conceptos a tierra. Utilizaremos el siguiente código, donde leemos los datos crudos del archivo 'Say Goodbye.wav', una canción buenísima del músico Hiroshi Satoh y que exportamos con un formato de 16 bits por muestra (por supuesto, deben tener un archivo con ese nombre para que el código funcione, o modificar ese nombre):

```
import soundfile as sf
import numpy as np
import wave

wf = wave.open('Say Goodbye.wav')
sato_binario = wf.readframes(4)

sato_deco_sf, fs = sf.read('Say Goodbye.wav', start=0, stop=4)

In [1]: sato_deco_sf
Out[1]: array([-3.05175781e-04, -1.52587891e-04],
[-2.74658203e-04,  1.83105469e-04],
[-3.05175781e-05, -3.05175781e-05],
[0.00000000e+00, -1.52587891e-04]])

In [2]: sato_binario
Out[2]: b'\xff\xfb\xff\xfb\xff\xf7\xff\x06\x00\xff\xff\xff\xff\x00\x00\xfb\xff'
```



Impactante, horrible... Bueno, `soundfile` no nos sorprende demasiado: una matriz de dos columnas, una por cada canal, de modo que cada señal de audio es un vector. Sin embargo, la variable `sato_binario` es mucho más interesante. ¡En esa variable hay números en formato hexadecimal! Python indica que esta es información codificada utilizando el prefijo `b`, de modo que `b'xff'` es el número hexadecimal `FF`. Vamos a tratar de entender qué hizo `soundfile` para leer estos números. Para eso, vamos a utilizar la función `np.frombuffer` para decodificar la información en la variable `sato_binario` con formato '`int16`':

```
sato_deco_np = np.frombuffer(sato_binario, dtype='int16')

In [3]: sato_deco_np
Out[3]: array([-10, -5, -9, 6, -1, -1, 0, -5],
dtype=int16)
```

Recordemos que estamos leyendo 4 muestras de audio, y sin embargo aquí aparecen 8 valores. Esta la sabemos: en la cadena de hexadecimales estaban *los dos canales del audio estéreo*. De este modo, uno de cada dos valores irá al canal izquierdo, y otro irá al canal derecho. Cada uno de esos valores representa una muestra en un número de 16 bits. Intentemos asociarlos a la cadena de valores en hexadecimal. Para ello, recordamos que aunque la información venga empaquetada en *bytes* (8 bits), cada número en hexadecimal representa 4 bits, y por lo tanto, *cada paquete de 4 números hexadecimales es una muestra*. Además, es importante saber que los bytes se leen desde el más chico al mas grande, de manera que los primeros valores, `xf6` y `xff` deberían leerse como un único número `xfffff6`. Este protocolo para almacenar y leer la información se denomina ***little endian***. Sabiendo eso, analicemos qué valor entero son los primeros valores en hexadecimal:

$$\begin{aligned} xFFF6 &= 1111111111110110 = 65526 = 65536 - 10 \\ xFFFB &= 111111111111011 = 65531 = 65536 - 5 \\ xFFF7 &= 1111111111110111 = 65527 = 65536 - 9 \\ x0006 &= 0000000000000110 = 6 \end{aligned}$$

¡Están apareciendo los números que habíamos decodificado! Evidentemente, vamos por buen camino. Observen que cuando los números empiezan en 1, son negativos, y los valores que se representan son, en realidad, *cuánto nos falta para llegar a  $65536 = 2^{16}$* . Esto se llama **representación en complemento a la base**. Finalmente, observemos cómo llegar a los valores de `soundfile`. Para ello, recordemos que `soundfile` devuelve valores de tipo `float` entre  $-1$  y  $1$ , y por ello, si nuestro número en formato '`int16`' puede, a lo sumo, representar el valor  $32768 = 2^{15}$ , ya sea positivo o negativo, la forma automática de llevar el número a un `float` entre  $-1$  y  $1$  será dividirlo por  $32768$  (siempre y cuando se haya codificado usando 16 bits, desde luego, pero el razonamiento se adapta fácilmente a otras codificaciones). Construyamos, entonces, el canal L y el canal R, tomando una de cada 2 muestras y dividiéndolos por  $32768$  e imprimamos los resultados junto a la matriz original `sato_deco_sf`:

```
satoL = sato_deco_np[:,2] / 32768
satoR = sato_deco_np[1::2] / 32768
```

```
In [4]: sato_deco_sf
Out[4]: array([-3.05175781e-04, -1.52587891e-04],
              [-2.74658203e-04,  1.83105469e-04],
              [-3.05175781e-05, -3.05175781e-05],
              [0.00000000e+00, -1.52587891e-04]])

In [5]: satoL
Out[5]: array([-3.05175781e-04, -2.74658203e-04,
               -3.05175781e-05,  0.00000000e+00])

In [6]: satoR
Out[6]: array([-1.52587891e-04,  1.83105469e-04,
               -3.05175781e-05, -1.52587891e-04])
```

Felizmente, Python nos ha dado la razón: decodificando los valores hexadecimales en formato '`int16`' (es decir, como enteros de 16 bits) y dividiéndolos por  $2^{15} = 32768$  hemos reproducido perfectamente una librería de alto nivel como `soundfile`. Esto es muy importante, porque pronto vamos a leer bits desde la entrada de la computadora, y si queremos procesarlos de alguna manera, es muy difícil hacerlo sin pasar a un formato vectorial. Finalmente, mostramos cómo recuperar una señal a su forma en bits. Para eso volvemos multiplicar el vector de `floats` por 32768, volvemos a convertirlo a formato '`int16`' con el método `.astype()` y volvemos a llevarlo a hexadecimal con el método `.tobytes()`:

```
estereo_de_nuevo = np.array([satoL, satoR]).T
ints_de_nuevo = (estereo_de_nuevo * 32768).astype('int16')
bits_de_nuevo = ints_de_nuevo.tobytes()
```

y por las dudas, verificamos:

```
In [7]: bits_de_nuevo == sato_binario
Out[7]: True
```

En resumen, el proceso que atravesamos fue el siguiente:

1. Hemos levantado una cadena de bits de un archivo de audio
2. Los convertimos en un vector con `np.frombuffer`
3. Lo normalizamos para que estén entre  $-1$  y  $1$  dividiendo por 32768
4. Lo llevamos de vuelta a `int` con `.astype()`
5. Lo llevamos de vuelta a valores en binario / hexadecimal con `.tobytes()`

Por supuesto, no hemos aplicado ningún proceso más allá de la conversión, pero podríamos haber aplicado algún proceso cuando tuvimos el vector normalizado, para luego volver a valores en binario. A ello nos dedicamos en la próxima sección.

### 7.3. PyAudio, *buffers* y *streams*

Bueno, ¡eso estuvo durísimo! Vamos a tratar de usarlo para hacer algo un poco más divertido, que es lo que esto tiene que ser. El procesamiento de audio en tiempo real se basa en el armado de *buffers* que viajan a través de un *stream*. La idea de un *buffer* es un cuerpo de datos de un tamaño fijo que se procesan todos juntos en la memoria de la computadora. De esta manera, suele definirse un *chunk* (pedazo) de datos que se procesan juntos. Como pasar millones de muestras de audio a un proceso tendría una gran demora, podemos, por ejemplo, dividir esos millones de muestras en *chunks* de 1024 muestras que se procesan juntas, esto es, un *buffer* de 1024 muestras. El *stream* es simplemente un flujo de datos que va desde la entrada a la salida del sistema. De esta manera, lo que tenemos que hacer es:

1. Abrir un stream para leer datos de la entrada del sistema (podría también abrirse un stream desde un archivo del sistema hasta la salida, pero en este caso nos proponemos usar el micrófono).
2. Definir parámetros sobre cómo viajará el audio en ese stream: canales, tamaño de buffer, frecuencia de muestreo, formato de codificación (es decir, cuántos bits se usarán para cuantificar), dispositivos de entrada y salida, entre otros.
3. Llevar los datos que pasan por el stream a un formato más manejable que el de los datos crudos.
4. Aplicar un proceso a esos datos, y luego actuar en consecuencia: podríamos querer grabarlos, graficarlos, o reformatearlos a binario y reenviarlos a la salida para escucharlos.

Este control del stream de audio es lo que permite hacer la librería PyAudio, que nos da conexiones con PortAudio, la librería de bajo nivel que controla las conexiones con el hardware en Windows y en varias distribuciones de Linux. ¡Ojo, que en algunos sistemas operativos puede haber dependencias que ignoramos y hacerlo andar puede exigir instalar más cosas! **Es muy recomendable instalar PyAudio utilizando `conda install pyaudio` y no tanto así `pip install pyaudio`.** Para instalaciones que así y todo

se nos complican en Windows, otra alternativa es instalar primero pipwin, haciendo `pip install pipwin` y, luego, `pipwin install pyaudio`. Una vez instalada, un modo de uso muy elemental es el siguiente<sup>2</sup>:

```
import pyaudio

RATE = 44100
CHUNK = 1024
CHANNELS = 1
FORMAT = pyaudio.paInt16

p = pyaudio.PyAudio()

stream = p.open(rate=RATE,
                 channels=CHANNELS,
                 format=FORMAT,
                 input=True,
                 output=True,
                 frames_per_buffer=CHUNK)

data = stream.read(4)
print(data)
```

En esta ejecución, Python nos ha devuelto la siguiente cadena:

```
b'\xe7\xff\xf2\xec\xff\xd9\xff'
```

Es muy probable que en otra computadora no resulte lo mismo. De hecho, ejecutarlo dos veces nos dará probablemente un resultado distinto. ¡Eso es esperable! El código abre un stream de audio utilizando tanto la entrada como la salida (que se activan con el booleano `True`), una frecuencia de muestreo (en este caso, `pyaudio` la llama `rate`) de 44100 muestras por segundo, un solo canal, un formato de codificación de 16 bits y 1024 muestras en cada buffer. Luego, leemos 4 muestras de ese stream. Este código simplemente imprime el valor binario de 4 muestras de audio que ingresan por el micrófono, y si el ruido ambiente dónde esté posicionado el micrófono que utilicemos está cambiando, es razonable pensar que cada ejecución del código imprimirá una cadena diferente.

Hagamos algo con esos datos. En principio, construyamos un grabador que pueda grabar por tiempo indefinido, en tanto podemos, por ejemplo, acumular los datos que ingresan al sistema en una variable que vaya creciendo

---

<sup>2</sup>La documentación oficial de PyAudio, junto a algunos ejemplos elementales, se encuentra disponible en <https://people.csail.mit.edu/hubert/pyaudio/docs/#id8>

y, a la par que se graban en esa variable, enviarlos a la salida para ser reproducidos. Para ello, inmediatamente después de definir la variable `stream`, colocamos:

```
data = b''
while True:
    monitoreo = stream.read(CHUNK)
    data += monitoreo
    stream.write(monitoreo)
    #print(monitoreo)

stream.close()
p.terminate()
```

Aquí hemos dispuesto la información en un loop, en principio, infinito. **Para detener este loop y, con ello, el `stream`, hay que generar un error en el sistema, con lo cual podemos presionar `ctrl + C` en Spyder para generar un `KeyboardInterrupt`** (pronto mejoraremos esto). La variable `data` contendrá todo el audio que hayamos grabado, acumulada en bloques de 1024, y la sentencia `stream.write()` envía la información que ingresó al sistema nuevamente a la salida, para monitorear lo que se está grabando. Descomentar la línea comentada enviará a la consola cada bloque de tamaño `CHUNK` que ingresa al sistema, lo cual resultará, evidentemente, en largas cadenas de números en formato hexadecimal imprimiéndose sucesivamente. Finalmente, cerramos el `stream` y terminamos la conexión con PortAudio.

## 7.4. Un analizador de espectro en tiempo real

Habiendo comprendido todo el procedimiento mostrado en la sección 7.2, así como la interacción sencilla con el `stream` provista por `PyAudio`, será sencillo realizar un analizador de espectro que funcione en tiempo real. Lo único que debemos agregar al código serán un gráfico de una figura con dos ejes (uno para el diagrama temporal y uno para el diagrama frecuencial), cuya información se actualice dentro de un loop, y dentro de ese loop, procesar cada ventana del buffer obteniendo su espectro y convirtiendo ese espectro en la información del gráfico. Comenzamos por construir un código para el diagrama temporal:

```
import numpy as np
import pyaudio
import matplotlib.pyplot as plt
```

```
p = pyaudio.PyAudio()

FORMAT = pyaudio.paInt16
RATE = 44100
CHANNELS = 1
CHUNK = 1024

# Inicializo dos ejes en una figura
fig, (ax1, ax2) = plt.subplots(2)
# Eje del gráfico temporal
ax1.set_xlim(0, CHUNK)
ax1.set_ylim(-1, 1)

stream = p.open(rate=RATE,
                 channels=CHANNELS,
                 format=FORMAT,
                 input=True,
                 output=True,
                 frames_per_buffer=CHUNK)

# Valores iniciales en los ejes X e Y
x = np.arange(CHUNK)
line, = ax1.plot(x, np.zeros(CHUNK))

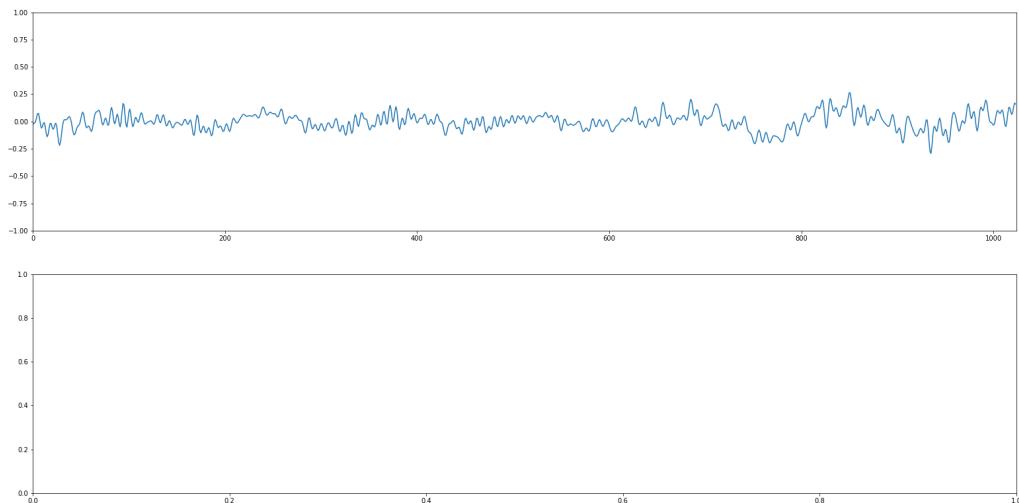
print(50 * '#')
print('Apretá Ctrl + C para parar el monitoreo')
print(50 * '#')
while True:
    monitoreo = stream.read(CHUNK)

    decodificada = (np.frombuffer(monitoreo, dtype='int16')
                     / 32768)
    line.set_ydata(decodificada)

    fig.canvas.draw()
    fig.canvas.flush_events()

stream.close()
p.terminate()
```

Aquí hemos creado una figura con dos cuerpos de ejes utilizando la sentencia `plt.subplots`, y hemos configurado los límites del primer grupo de ejes, destinado a graficar la señal temporal. El eje *x* del gráfico temporal simplemente será creciente, mostrando el número de muestra del buffer, en tanto las líneas a graficar son inicializadas con un vector de ceros. Finalmente, dentro del loop, los bytes dentro de `CHUNK` son normalizados y llevados a valores de tipo `float`. El vector resultante es asignado a la información en el eje dependiente con `line.set_ydata`, y la información es dibujada y actualizada con `fig.canvas.draw` y `fig.canvas.flush_events` respectivamente. Esto debería resultar en un gráfico como este, que se actualiza a medida que la entrada de audio del sistema recibe información:



El segundo eje todavía está en blanco, pues es el que se encargará de graficar la transformada de Fourier. Para ello, generamos la información referida a ese nuevo eje fuera del loop, llevando sus límites a los valores usuales:

```
fig, (ax1, ax2) = plt.subplots(2) # Inicializo dos ejes
ax1.set_xlim(0, CHUNK) # Eje del gráfico temporal
ax1.set_ylim(-1, 1)
ax2.set_xlim(20, 20000) # Frecuencias de 20 Hz a 20 kHz
ax2.set_ylim(0, 1)
```

El contenido inicial del gráfico es inicializado mediante `np.fft.rfftfreq`, según se hizo en el Capítulo 4. Es importante notar que `np.fft.rfft` devuelve `CHUNK//2 + 1` muestras, y por ello necesitamos esa cantidad de ceros. Observar que estamos usando un eje para las frecuencias:

```
x = np.arange(CHUNK) # Eje x, temporal
x_fft = np.fft.rfftfreq(CHUNK, 1/RATE) # Eje x, frecuencial
```

```
# Ceros en el eje x temporal
line, = ax1.plot(x, np.zeros(CHUNK))
# Ceros en el eje x frecuencial
line_fft, = ax2.semilogx(x_fft, np.zeros(CHUNK//2 + 1),
                           linewidth=2, color='r')
```

Finalmente, actualizamos el loop para que incorpore la transformada de Fourier:

```
print(50 * '#')
print('Apretá Ctrl + C para parar el monitoreo')
print(50 * '#')
while True:
    monitoreo = stream.read(CHUNK)

    decodificada = (np.frombuffer(monitoreo, dtype='int16')
                     / 32768)
    line.set_ydata(decodificada)

    y_fft = np.fft.rfft(decodificada)
    espectro = np.abs(y_fft) / (CHUNK // 2)

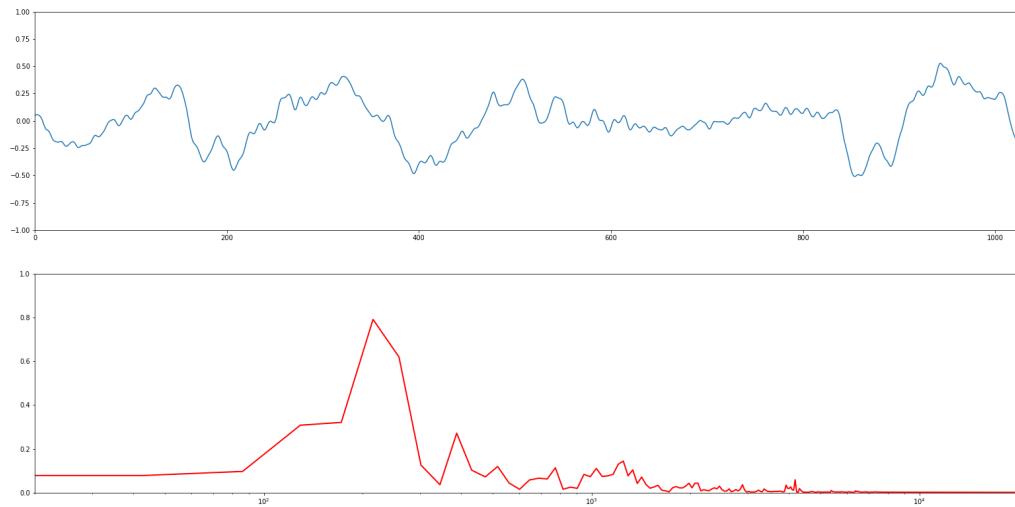
    line_fft.set_ydata(espectro)

    fig.canvas.draw()
    fig.canvas.flush_events()

stream.close()
p.terminate()
```

Lo único novedoso en este loop es que a cada buffer que ingresa, además de decodificarlo para enviar su información al gráfico temporal, se le calcula la transformada de Fourier, y esta se normaliza y se grafica en el segundo eje. Finalmente el gráfico se dibuja y actualiza. Observar que a diferencia del capítulo 4, no normalizamos el espectro por el máximo sino por la longitud de la transformada. Si normalizamos con el máximo, como el máximo se irá actualizando para cada buffer, siempre tendríamos algún valor llegando a 1, aún con silencio en la entrada de audio.

El gráfico dinámico resultante debería verse como este:



Es un poco molesto eso de tener que generar un error en el programa para detenerlo. Mejor, capturemos la excepción generada con el comando `try` y reformulemos el loop de modo que, si se encuentra una interrupción de la consola a través de `KeyboardInterrupt`, se salga del `while` en forma elegante y prevista, y se cierre la figura. El código completo quedará así:

```
import numpy as np
import pyaudio
import matplotlib.pyplot as plt

p = pyaudio.PyAudio()

FORMAT = pyaudio.paInt16
RATE = 44100
CHANNELS = 1
CHUNK = 1024

fig, (ax1, ax2) = plt.subplots(2) # Inicializo dos ejes
ax1.set_xlim(0, CHUNK) # Eje del gráfico temporal
ax1.set_ylim(-1, 1)
ax2.set_xlim(20, 20000) # Frecuencias de 20 Hz a 20 kHz
ax2.set_yscale('log')
ax2.set_ylim(0, 1)

stream = p.open(rate=RATE,
                 channels=CHANNELS,
                 format=FORMAT,
                 input=True,
```

```

        output=True,
        frames_per_buffer=CHUNK)

x = np.arange(CHUNK) # Eje x, temporal
x_fft = np.fft.rfftfreq(CHUNK, 1/RATE) # Eje x, frecuencial

# Ceros en el eje x temporal
line, = ax1.plot(x, np.zeros(CHUNK))
# Ceros en el eje x frecuencial
line_fft, = ax2.semilogx(x_fft, np.zeros(CHUNK//2 + 1),
                           linewidth=2, color='r')

print(50 * '#')
print('Apretá Ctrl + C para parar el monitoreo')
print(50 * '#')
while True:
    try:
        monitoreo = stream.read(CHUNK)

        decodificada = (np.frombuffer(monitoreo, dtype='int16') /
                         32768)
        line.set_ydata(decodificada)

        y_fft = np.fft.rfft(decodificada)
        espectro = np.abs(y_fft) / (CHUNK // 2)

        line_fft.set_ydata(espectro)

        fig.canvas.draw()
        fig.canvas.flush_events()
    except KeyboardInterrupt:
        print(' *** Gracias por usar el Pynalizador de '
              'Espectro :) *** ')
        break

plt.close(fig)
stream.close()
p.terminate()

```

Ello deja el código funcional. Puede ajustarse el valor de CHUNK para mejorar el desempeño, o normalizar el espectro del eje frecuencial de otra forma (o

bien no normalizarlo y reajustar su límites en el eje  $y$ ). Queda a los lectores la importante tarea de complementar este capítulo con el capítulo 6, y construir una interfaz gráfica que permita al usuario iniciar y detener el monitoreo con un botón, así como dar al usuario una interfaz bella, bonita e interactiva que le permita ajustar distintos parámetros del gráfico. Puede ser interesante explorar también otras alternativas gráficas, como `PyQtGraph`, que si bien no son tan estándar como `matplotlib`, tienen otras alternativas visuales y, en ocasiones, trabajan en forma mucho más rápida<sup>3</sup>.

## 7.5. Modo *Callback*

Terminamos este capítulo con otra forma de utilizar `PyAudio` para procesar una señal en tiempo real. En particular, esta podría no ser la mejor para realizar un gráfico como el que hicimos anteriormente, pero es algo más ordenada y eficaz para procesar una señal en tiempo real y escuchar el resultado, y es fácil de incluir en una aplicación con interfaz gráfica. Esto es el denominado *modo callback* para utilizar el stream, y consiste en definir una función (llamada `callback` en nuestro ejemplo) que se aplicará a cualquier buffer que ingrese al stream. La función que utilizará el stream se especifica en el argumento `stream_callback` de la función `p.open`, la función de `PyAudio` que abre el stream. Las diferencias en el modo de uso son menores:

```
import numpy as np
import pyaudio

p = pyaudio.PyAudio()

FORMAT = pyaudio.paInt16
RATE = 44100
CHANNELS = 2
CHUNK = 1024

def callback(in_data, frame_count, time_info, status_flags):
    data = in_data
    data = np.frombuffer(data, dtype='int16') / 32768
```

---

<sup>3</sup>La velocidad de graficado es un limitante importante para aplicaciones animadas con `matplotlib`. Hemos tomado ideas de este artículo para generar un gráfico lo suficientemente ágil, pero su lectura completa de seguro permitirá alcanzar mayores velocidades. La idea del analizador fue tomada de este video, al que se han hecho algunas correcciones significativas en la conversión de formatos de audio digital.

```

# ¡ACÁ SE PROCESA DATA!
print(data)

return (data, pyaudio.paContinue)

stream = p.open(rate=RATE,
                 channels=CHANNELS,
                 format=FORMAT,
                 input=True,
                 output=True,
                 frames_per_buffer=CHUNK,
                 stream_callback=callback)

stream.start_stream()

while stream.is_active():
    input('***** Detener monitoreo con Enter *****')
    stream.stop_stream()
print('Y colorín colorado, este libro se ha terminado.')
stream.close()
p.terminate()

```

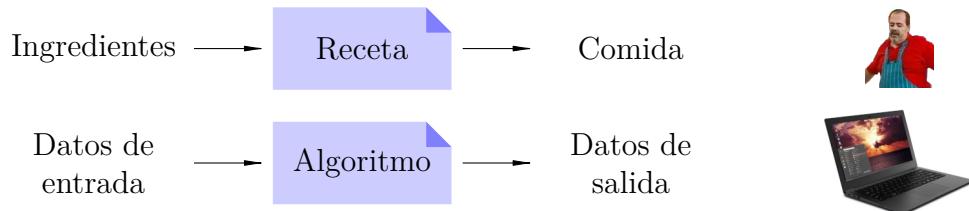
Nótese que las diferencias son menores. En este caso, `stream` se inicializa con el argumento `stream_callback` asignado a la función `callback`, que hemos definido oportunamente. Ello implica que el `stream` ya tendrá asignado qué proceso aplicar a la señal, esto es, el que sea que se haya definido en `callback`. Por ello, basta con inicializar el `stream` con `stream.start_stream` para que `callback` comience a aplicarse sobre los buffers que fluyen por el `stream`. Utilizamos el booleano `stream.is_active` para mantener el `stream` activo en tanto el usuario no presione Enter o Return, lo cual ejecutaría el `input` y, seguidamente, detendría el `stream` con `stream.stop_stream`. La función `callback` se define siempre con los mismos argumentos: de ellos obtenemos la información de audio que ingresa al `stream` (en la variable `in_data`). `frame_count` contiene un `int` con el tamaño del buffer, en tanto `time_info` y `status_flags` nos dan información del hardware que PyAudio obtiene de PortAudio, y que si bien no suele ser necesaria, puede llegar a servir para conocer las limitaciones del sistema o si se encuentran errores de procesamiento. En este caso, no estamos realizando ningún procesamiento muy especial, simplemente transformando la información de entrada a un formato vectorial e imprimiéndola en la consola a medida que atraviesa el `stream`, pero puede hacerse lo que se quiera con esta información, **en**

**tanto esta sea eventualmente reconvertida a un formato binario y devuelta al stream.** Como mostramos en el ejemplo, `callback` necesariamente debe devolver una tupla que contenga dos elementos. El primero debe ser un bloque que contenga `frame_count` muestras de información codificada (de otro modo el stream encontraría una inconsistencia entre lo que entra y sale) y el flag `pyaudio.paContinue`, que indica si aún quedan muestras de audio que deban ser grabadas o reproducidas. Con ello, dejamos un código preparado para aplicar cualquier proceso y escuchar la respuesta, esperando que los lectores lo encuentren tan divertido como nosotros. Por supuesto, ese proceso podría venir de alguno de los ejemplos del capítulo 2, y nuestro código de PyAudio puede ser integrado dentro de una interfaz gráfica diseñada a la manera del capítulo 6.

# Apéndice A

## Computadoras y algoritmos

Programar es como escribir una receta de cocina.



La receta es un ejemplo de *conocimiento imperativo*. Es una serie de pasos que el cocinero debe seguir, para preparar correctamente la comida a partir de los ingredientes. Lo mismo pasa con un *algoritmo*: es una serie de *instrucciones que la computadora debe seguir*, para generar correctamente datos de salida, en base a los datos de entrada. **Programar es pensar e implementar algoritmos.** En otras palabras,

**Programar es darle instrucciones a la computadora para que lea y escriba datos de una determinada manera.**

De esta afirmación, surgen inmediatamente dos preguntas:

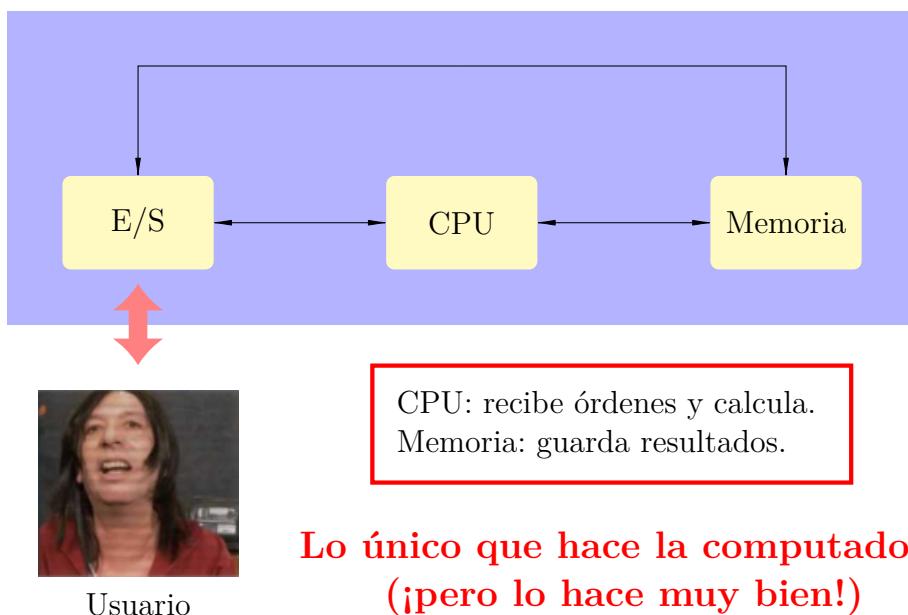
1. ¿Cómo hace la computadora para seguir instrucciones?
2. ¿Cómo hace una persona para darle las instrucciones?

El objetivo del presente capítulo es responder estas preguntas.

## A.1. ¿Qué es una computadora?

Comenzaremos esta sección con una frase de John Guttag: “una computadora hace dos cosas, y sólo dos: **realiza cálculos y recuerda los resultados de esos cálculos**. Pero lo hace extremadamente bien” [1].

El siguiente es un diagrama súper simplificado, pero resume las características fundamentales de una computadora cualquiera.



CPU es *unidad central de procesamiento* (por “Core Processor Unit”). E/S significa *Entrada/Salida*, e involucra todo lo necesario para interactuar con el usuario, como p. ej. el mouse y el teclado.

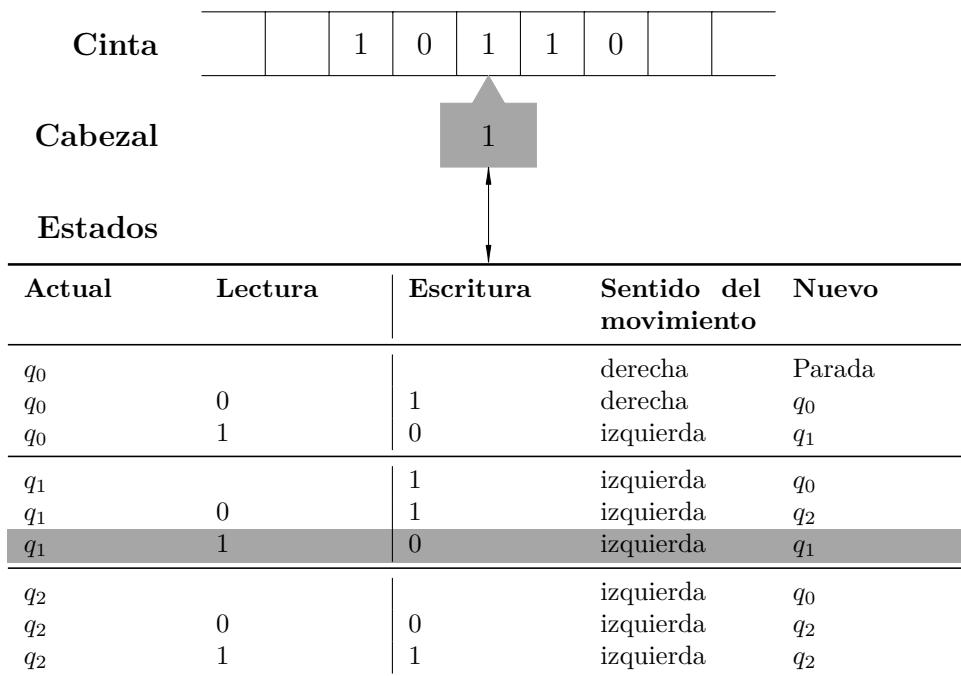
El dibujo está basado en un modelo de John von Neumann. En 1945, este brillante señor sentó las bases de la arquitectura de computadoras moderna, al concebir el concepto de “programa almacenado” [2]. Por ese entonces, las computadoras eran máquinas que realizaban siempre una misma operación. Por ejemplo, Atanasoff y Berry construyeron una que resolvía sistemas lineales de ecuaciones mediante tarjetas perforadas y válvulas [3]. Sólo resolvía sistemas lineales: era una computadora de *programa fijo*. La novedad del modelo de von Neumann, entre otras cosas, fue categorizar a la computadora como un dispositivo *programable*. Bajo estas condiciones, las computadoras ya no necesariamente calculaban la misma cosa cada vez, sino que recibían instrucciones para saber qué calcular.

En resolución, una **computadora** es un dispositivo que **puede programarse dando instrucciones a su CPU para hacer cuentas, cuyos resultados se guardan en la Memoria**.

## A.2. ¿Qué instrucciones recibe?

Un procesador Intel o AMD de 64 bits, como la CPU que probablemente tiene nuestra computadora en casa, posee una gran cantidad de instrucciones fundamentales, sobre las cuales se construyen los algoritmos en su forma final. Algunas de ellas son las operaciones matemáticas elementales (sumar, restar, multiplicar y dividir), pero también hay instrucciones más oscuras del estilo “mover 4 bytes de una posición en la memoria a otra”, “calcular el complemento a dos”, “cargar los bits de una bandera de registro”, etc [4].

La descomposición de un determinado algoritmo como secuencia de instrucciones más básicas no es un concepto precisamente nuevo. En 1937, un joven muchacho llamado Alan Turing definió teóricamente un concepto que denominó “a-machine” [5] (máquina automática, hoy llamada “máquina de Turing”). La máquina de Turing es el primer modelo matemático de computadora conocido. Consta de tres componentes:



La **cinta** es idealmente infinita y contiene símbolos. En particular, los símbolos pueden ser número binarios y espacios en blanco, como en el ejemplo del dibujo. El **cabezal**, ubicado bajo la cinta en el dibujo, solamente puede *leer y escribir símbolos de la cinta, y moverse a la izquierda y a la derecha*. La máquina decide cuál de estas acciones realizar, en base a la lectura del cabezal y al **estado** en el que se encuentra. En el dibujo, la máquina tiene el estado  $q_1$ , y la lectura del cabezal es igual a 1, como indica la fila sombreada

de la tabla. En consecuencia, la máquina debe seguir los siguientes pasos.

1. Escribir 0 en la cinta.
2. Mover el cabezal a la izquierda.
3. Quedarse en el estado  $q_1$ .

En general, la tabla asocia los valores en el primer par de columnas (**Estado actual**, **Lectura**) a los valores de las últimas tres (**Escritura**, **Sentido del movimiento**, **Nuevo estado**).

El funcionamiento general de la máquina es el siguiente.

1. Iniciar en el **estado**  $q_0$ .
2. Tomar la **lectura** del cabezal.
3. Ubicar la fila de la tabla que se corresponde con el **estado actual** y la **lectura** obtenida en el paso anterior.
4. Según la fila ubicada, realizar los siguientes pasos.
  - a) Escribir en la cinta el valor que indique la columna **Escritura**.
  - b) Mover el cabezal en el sentido que indique la columna **Sentido del movimiento** correspondiente.
  - c) Cambiar al estado que indique la columna **Nuevo**.
5. Repetir los pasos 2–4 hasta alcanzar el estado “Parada”.

A priori, es imposible saber si una máquina dada llegará al estado de “Parada” en una cantidad finita de pasos, lo cual fue demostrado por Turing en su trabajo [5]. Esta cuestión, llamada *problema de parada* (“halting problem”), tiene suma importancia para la informática, puesto que determina una **limitación de las computadoras**. Una reformulación moderna del problema de parada podría ser la siguiente.

**No puede construirse un algoritmo que decida si otro termina en una cantidad finita de pasos, para todas sus entradas posibles.**

Sólo un programador humano con cierta experiencia puede predecir o estimar si terminará, en algún momento, la ejecución de un algoritmo determinado. ¡Punto para la humanidad!

De todas formas, las computadoras son una herramienta poderosa. Es un hecho que todos comprobamos día tras día. En particular, se supone que

**una máquina de Turing puede reproducir cualquier algoritmo [5]**, siempre y cuando dicho algoritmo pueda ser expresado como sucesión de números binarios y espacios en blanco. Así, los algoritmos se reducen a una secuencia de instrucciones fundamentales simples: leer, escribir, mover el cabezal, y cambiar de estado.

Si bien las computadoras modernas tienen más instrucciones que las de una “máquina automática”, el aporte de Turing a las Ciencias de la Computación tiene suma importancia, por ser una de las primeras representaciones abstractas de conceptos ampliamente utilizados hoy en día: las nociones de “computadora” y “algoritmo”, junto con sus limitaciones. Además, *lograr que la máquina realice una tarea de cierta complejidad a partir de instrucciones básicas es, precisamente, programar.*

### A.3. ¿Cómo se dan las instrucciones?



En el fondo, la **computadora sólo entiende números binarios**. Si yo le digo “haceme un sánduche”, no lo va a hacer. Si en cambio le digo 10010111010001, y esa resulta ser la secuencia binaria que trae incorporado el CPU para encender una cocina y hacer sándwiches automáticamente, entonces me lo hace. *La secuencia binaria se denomina código máquina.* Programar de esta forma es, para una persona normal, una locura. Roza el delirio místico. Para evitar esto, se usan *lenguajes de programación* y, en particular, *python*.

El **lenguaje de programación** es el **intermediario** entre el **código máquina** y el lenguaje que usamos los humanos para comunicarnos entre nosotros, llamado **lenguaje natural**.

Los lenguajes de programación suelen clasificarse de la siguiente forma [1].

- Según el tipo de *ejecución*, en lenguaje **interpretado** y **compilado**.
- Según el *nivel*, en **alto nivel** y **bajo nivel**.
- Según el *campo de aplicación*, en **general** y **particular**.

Las instrucciones generadas con un lenguaje interpretado son ejecutadas directamente por un intérprete. Las instrucciones en lenguaje compilado requieren ser traducidas primero a código máquina por un compilador.

Lenguaje  
interpretado

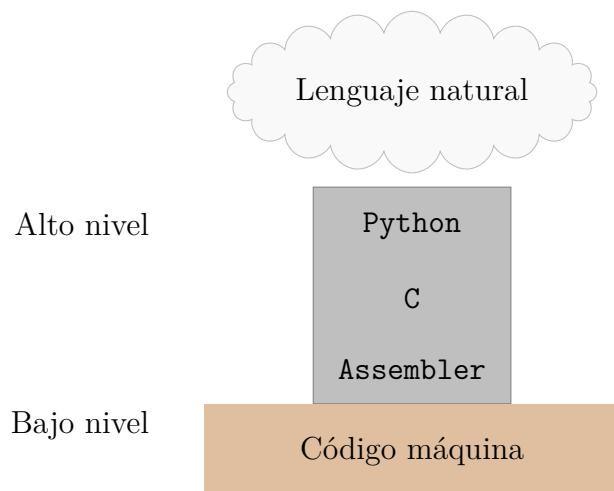
Algoritmo → Intérprete

Lenguaje  
compilado

Algoritmo → Compilador → Código máquina

Es más fácil hacer pruebas con lenguaje interpretado: escribo, ejecuto, cambio, ejecuto de nuevo, etc. El proceso de compilación suele consumir mayor tiempo, lo cual demora las pruebas. No obstante, estos programas, una vez compilados, suelen funcionar más rápido y consumir menos memoria que los interpretados.

El *nivel* del lenguaje mide las distancias que tiene con el código máquina y el lenguaje natural. Un lenguaje de bajo nivel está mas cerca del código máquina. En cambio, uno de alto nivel es más parecido al lenguaje natural.



Un lenguaje de programación de *aplicación particular* está hecho para resolver cierto tipo de problemas, p. ej.: mySQL para bases de datos, Prolog para inteligencia artificial, PSEInt para enseñanza de la programación, etc. Otros lenguajes, como C++, java, y lua, hacen de comodines para resolver un amplio abanico de problemas de diversa índole. Son de *propósito general*.

Particularmente

**python es un lenguaje de programación interpretado, de alto nivel, y de propósito general.**

Además, es **Turing-completo: resuelve los mismos problemas que puede resolver una máquina de Turing**. *Todo lenguaje de programación moderno y decente es Turing-completo*. Así, cualquier problema que resuelve python también lo resuelve lua o java, por ejemplo, que también son Turing-completos. Lo que cambia entre un lenguaje y otro es la facilidad que traen para resolver determinados tipos de problemas. Esto depende, generalmente, del campo de aplicación del lenguaje.

Es importante destacar que *ninguno de los lenguajes de programación mencionados puede resolver el problema de parada*. Como dijimos, esta es una limitación de las computadoras en general. Cuando un programa se trabe, será responsabilidad nuestra entender por qué lo hace.

## Referencias

- [1] J. V. Guttag, *Introduction to computation and programming using Python*, 2.<sup>a</sup> ed. United States of America: Massachusetts Institute of Technology Press, 2016 (vid. págs. 145, 148).
- [2] J. von Neumann, «First Draft of a Report on the EDVAC», University of Pennsylvania, Pennsylvania, 30 de mayo de 1945 (vid. pág. 145).
- [3] A. R. Burks y A. W. Burks, *The First Electronic Computer: The Atanasoff Story*, 1.<sup>a</sup> ed. United States of America: The University of Michigan Press, 1989 (vid. pág. 145).
- [4] J. V. Hoey, *Beginning x64 Assembly Programming: From Novice to AVX Professional*, 1.<sup>a</sup> ed. Hamme, Belgium: Apress Media, 2019 (vid. pág. 146).
- [5] A. Turing, «On Computable Numbers, with an Application to the Entscheidungsproblem», en *Proceedings of the London Mathematical Society*, ép. 2, vol. 42, 1937 (vid. págs. 146-148).