**7. Diseño, especificación, documentación y estilo.**

En este curso queremos que aprendas a escribir un script que te resuelva un problema computacional. Pero también queremos que puedas escribir adecuadamente programas más grandes, que los puedas compartir y volver a usar vos misme unos años más tarde.

Por eso insistimos con algunos temas de estilo, documentación, especificiación y diseño que ya hemos comentado anteriormente y sobre los que volveremos en esta clase. Uno de ellos es que es conveniente administrar los errores; seguiremos hablando sobre las formas adecuadas de hacerlo y porqué no conviene hacerlo de más. También se vuelve indispensable estructurar adecuadamente el código y aprender a definir una función *main*. Vamos a continuar con nuestras discusiones sobre el diseño de algoritmos y sus estructuras de datos asociadas. También queremos que aprendas algunos conceptos elementales sobre especificación de problemas. Son procesos de abstracción que nos ayudan a pensar con mayor claridad. Al especificar un problema con precondiciones y poscondiciones estamos definiendo qué es lo que debe pasar en una función, por ejemplo (aunque en ningún momento decimos cómo debe pasar esto). Una especificación es como un contrato y podemos definir varias funciones que cumplan el contrato, y cada una puede resolverlo a su manera.

Finalmente, daremos un poco más sistemáticamente algunos conceptos de la biblioteca matplotlib, incluyendo el manejo de figuras y subplots.

Ésta es la última clase antes del primer parcial. El miércoles dentro de dos semanas acordate de estar atente de 14 a 16hs que tomaremos el parcial on-line. Sabemos que hay gente que no está haciendo la materia por los créditos, sino para aprender los contenidos. Les pedimos que igual rindan los exámenes y soliciten el certificado final de aprobación. Para nosotros es importante que los que hayan seguido el curso figuren formalmente para que esta expericiencia pueda tener continuidad en el tiempo.

* [7.1 Repaso de temas pasados](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/01_Repaso.md)
* [7.2 Control de errores](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/02_Excepciones.md)
* [7.3 El módulo principal](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/03_Modulo_principal.md)
* [7.4 Cuestiones de diseño](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/04_Flexibilidad.md)
* [7.5 Contratos: Especificación y Documentación](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/05_Especificacion_y_Documentacion.md)
* [7.6 Estilos de codeo](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/06_Estilo.md)
* [7.7 La biblioteca matplotlib](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/07_Matplotlib.md)
* [7.8 Cierre de la clase](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/08_Cierre.md)

**7.1 Repaso de temas pasados**

A continuación dejamos unos links a unos videos retomando temas de la clase anterior e introduciendo los nuevos mediante ejemplos.

1. En [este video](https://youtu.be/O65J5n6yA1g) hablamos sobre complejidad de algoritmos. Búsqueda secuencial y búsqueda binaria. El problema de la mochila.
2. En [este video](https://youtu.be/Cc9bJ5ACWks) seguimos hablando sobre complejidad de la búsqueda secuencial y la búsqueda binaria. Mencionamos las pre y post-condiciones y los invariantes de ciclo que veremos más adelante en esta misma clase.
3. En [este video](https://youtu.be/uJq3_EJ6MRI) resolvemos el Ejercicio [Ejercicio 6.14](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/05_BusqBinaria.md" \l "ejercicio-614-b%C3%BAsqueda-binaria) de insertar en una lista ordenada. De paso ilustramos los conceptos de pre y post-condiciones de una función y de varios temas de complejidad de algoritmos.
4. Finalmente, en [este video](https://youtu.be/6cCJ0RmDXSY) hablamos sobre la importancia de documentar para poder compartir y reutilizar el código].

**7.2 Control de errores**

Aunque ya hablamos de *excepciones*, en esta sección hablaremos de administración de excepciones y control de errores con mayor detalle. Dejamos [este video](https://youtu.be/1yX-uFioZ8w) con una introducción breve a esta sección.

**Formas en que los programas fallan**

Python no hace ningún control ni validación sobre los tipos de los argumentos que las funciones reciben ni los valores de estos argumentos. Las funciones trabajarán sobre todo dato que sea compatible con las instrucciones dentro de la función.

def add(x, y):

return x + y

add(3, 4) # 7

add('Hola', 'mundo') # 'Holamundo'

add('3', '4') # '34'

Si existen errores en una función, serán evidentes durante la ejecución de la función (en forma de una excepción).

def add(x, y):

return x + y

>>> add(3, '4')

Traceback (most recent call last):

...

TypeError: unsupported operand type(s) for +::

'int' and 'str'

>>>

Python acusa los errores en inglés. El error acusado acá puede traducirse como:

Recapitulando (llamada más reciente al final)

...

Error de tipo (de datos): tipo de argumento no admitido para +: 'int' y 'str'.

Es decir: la función intentó aplicar el operador + (suma) a dos argumentos de tipos distintos (entero y cadena) y no supo hacerlo. Por eso levantó una excepción.

**Excepciones**

Como ya dijimos, las excepciones son una forma de señalar errores en tiempo de ejecución. Acordate de que podés levantar una excepción usando la instrucción raise .

if nombre not in autorizados:

raise RuntimeError(f'{nombre} no autorizado')

Para *atrapar* una excepción, usá un bloque try-except.

try:

authenticate(nusuario)

except RuntimeError as e:

print(e)

**Administración de excepciones**

Una excepción se propagará hasta el primer except que coincida con ella.

def grok():

...

raise RuntimeError('Epa!') # Levanta una excepción acá

def spam():

grok() # Esta llamada va a levantar una excepción

def bar():

try:

spam()

except RuntimeError as e: # Acá atrapamos la excepción

...

def foo():

try:

bar()

except RuntimeError as e: # Por lo tanto la excepción no llega acá

...

foo()

Para administrar la excepción, usá instrucciones en el bloque except. Cualquier instrucción hará que Python considere a la excepción como administrada, incluso un pass pero es pertinente realizar acciones relacionadas con la excepción específica a administrar.

def grok(): ...

raise RuntimeError('Epa!')

def bar():

try:

grok()

except RuntimeError as e: # Excepción atrapada

instrucciones # Ejecuta estos comandos

instrucciones

...

bar()

Una vez atrapada la excepción, la ejecución continúa en la primera instrucción a continuación del try-except.

def grok(): ...

raise RuntimeError('Epa !')

def bar():

try:

grok()

except RuntimeError as e: # Excepción atrapada

instrucciones

instrucciones

...

instrucciones # La ejecución del programa

instrucciones # continúa acá

...

bar()

**Excepciones integradas**

Hay más de una veintena de tipos de excepciones ya integradas en Python. Normalmente, el nombre de la excepción indica qué anduvo mal (por ejemplo, se levanta un ValueError si el valor suministrado no es adecuado). La siguiente no es una lista completa. Vas a encontrar más en la [documentación del lenguaje](https://docs.python.org/3/library/exceptions.html#bltin-exceptions).

ArithmeticError

AssertionError

EnvironmentError

EOFError

ImportError

IndexError

KeyboardInterrupt

KeyError

MemoryError

NameError

ReferenceError

RuntimeError

SyntaxError

SystemError

TypeError

ValueError

**Valores asociados a excepciones**

Usualmente las excepciones llevan valores asociados, que te dan más información sobre la causa precisa del error. Este valor puede ser una cadena (*string*) o una tupla con valores diversos (por ejemplo un código de error y un texto explicando ese código).

raise RuntimeError('Nombre de usuario inválido')

La instancia de la variable suministrada a except (en nuestros ejemplos e) lleva asociado este valor.

try:

...

except RuntimeError as e:

# `e` contiene la excepción lanzada con su mensaje específico

...

e es una instancia del mismo tipo que la excepción, aunque si la imprimís suele tener aspecto de una cadena de caracteres.

except RuntimeError as e:

print('Fracasé. Motivo:', e)

**Podés atrapar múltiples excepciones**

Es posible atrapar diferentes tipos de excepciones en la misma porción de código, si incluís varios except en tu try:.

try:

...

except LookupError as e:

...

except RuntimeError as e:

...

except IOError as e:

...

except KeyboardInterrupt as e:

...

Como alternativa, si las vas a procesar a todas de la misma manera, las podés agrupar:

try:

...

except (IOError, LookupError, RuntimeError) as e:

...

**Todas las excepciones**

Para atrapar todas y cualquier excepción, se usa Exception así:

try:

...

except Exception: # PELIGRO. (ver abajo)

print('Hubo un error')

En general es mala idea "administrar" las excepciones de este modo, porque no te da ninguna pista de por qué falló el programa. Sólo sabés que "Hubo un error".

**Así NO se atrapan excepciones.**

Así es como NO debe hacerse la administración de excepciones.

try:

hacer\_algo()

except Exception:

print('Hubo un error.')

Esto atrapa todos los errores posibles, y puede complicar mucho el debugging cuando el código falla por algún motivo que no esperabas (por ejemplo, falta algún módulo de Python y lo único que te dice es "Hubo un error").

**Así es un poco mejor.**

Si vas a atrapar todas las excepciones, acá hay un modo algo más decente:

try:

hacer\_algo()

except Exception as e:

print('Hubo un error. Porque...', e)

Exception incluye toda excepción posible, de modo que no sabés cuál atrapaste. Al menos esta versión te informa el motivo específico del error. Siempre es bueno tener alguna forma de ver o informar errores cuando atrapás todas las excepciones posibles.

Sin embargo, por lo general es mejor atrapar errores específicos, y sólo aquellos que podés administrar. Errores que no sepas como manejar adecuadamente, déjalos correr (tal vez alguna otra porción de código los atrape y administre correctamente o tal vez lo mejor sea detener la ejecución).

**Re-lanzar una excepción**

Si necesitás hacer algo en respuesta a una excepción pero no querés atraparla, podés usar raise para volver a lanzar la misma excepción.

try:

hacer\_algo()

except Exception as e:

print('Hubo un error. Porque...', e)

raise

Esto te permite, por ejemplo, llevar un registro de las excepciones (*log*) sin administrarla, y re-lanzarla para administrarla adecuadamente más tarde.

**Buenas prácticas al administrar excepciones**

No atrapes excepciones que no vayas a manejar adecuadamente. Dejalas caer ruidosamente. Si es importante, alguien se va a encargar del problema. Sólo atrapá excepciones si *sos ese "alguien"*. Es decir: sólo atrapá aquellos errores que podés administrar elegantemente de forma que permita que el programa se siga ejecutando.

**La instrucción finally.**

finally especifica que esa porción de código debe ejecutarse sin importar si una excepción fue atrapada o no.

lock = Lock()

...

lock.acquire()

try:

...

finally:

lock.release() # esto SIEMPRE se ejecuta. Haya o no haya excepciones.

Una estructura como ésa resulta en un manejo seguro de los recursos disponibles (seguros, archivos, hardware, etc.)

**Ejercicios**

Trabajá siempre con las últimas versiones de tus archivos. En esta clase vamos a trabajar con el archivo fileparse.py y también con una nueva versión de informe.py. Por favor, copiá informe\_funciones.py a informe.py, que trabajaremos sobre este último archivo.

**Lancemos excepciones**

La función parse\_csv() que escribiste en el [Ejercicio 6.9](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/03_Funciones.md#ejercicio-69-trabajando-sin-encabezados) admite seleccionar algunas columnas por le usuarie, pero eso sólo funciona si el archivo de entrada tiene encabezados.

Modifcá tu código para que lance una excepción en caso que ambos parámetros select y has\_headers = False sean pasados juntos. Y que resulte:

>>> parse\_csv('../Data/precios.csv', select = ['nombre','precio'], has\_headers = False)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "fileparse.py", line 9, in parse\_csv

raise RuntimeError("Para seleccionar, necesito encabezados.")

RuntimeError: Para seleccionar, necesito encabezados.

>>>

Ahora que agregaste este control, te estarás preguntando si no deberías comprobar otras cosas también en tu función. Por ejemplo, ¿deberías comprobar que nombre\_archivo sea una cadena, que tipos sea una lista y otras cosas de ese estilo?

Como regla general, es mejor no controlar esas cosas, y dejar que el programa dé un error ante entradas inválidas. El mensaje de error va a darte una idea del origen del problema y te va ayudar a solucionarlo.

El motivo principal para agregar controles de calidad sobre los parámetros de entrada es evitar que tu programa sea ejecutado en condiciones que no tienen sentido. Si le pedís que haga algo que requiere encabezados y simultáneamente le decís que no existen encabezados implica estás usando la función incorrectamente. La idea general es estar protegido contra situaciones que "no deberían suceder" pero podrían.

**Atrapemos excepciones**

La función parse\_csv() que escribiste está destinada a procesar un archivo completo. Pero en una situacion real, es posible que los archivos CSV de entrada estén "rotos", ausentes, o que su contenido no se adecúe al formato esperado. Probá esto:

>>> camion = parse\_csv('../Data/missing.csv', types = [str, int, float])

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "fileparse.py", line 36, in parse\_csv

row = [func(val) for func, val in zip(types, row)]

ValueError: invalid literal for int() with base 10: ''

>>>

El error es: el texto '' es inválido para la función int()

Modificá la función parse\_csv() de modo que atrape todas las excepciones de tipo ValueError generadas durante el armado de los registros a devolver e imprima un mensaje de advertencia para las filas que no pudieron ser convertidas. Estas filas no deben ser procesadas (ya que no se puede hacer adecuadamente), y deben ser omitidas en el output de la función.

Este mensaje deberá incluir el número de fila que causó el problema y el motivo por el cual falló la conversión. Para probar tu nueva función, intentá procesar Data/missing.csv. Debería darte algo así:

>>> camion = parse\_csv('../Data/missing.csv', types = [str, int, float])

Fila 4: No pude convertir ['Mandarina', '', '51.23']

Fila 4: Motivo: invalid literal for int() with base 10: ''

Fila 7: No pude convertir ['Naranja', '', '70.44']

Fila 7: Motivo: invalid literal for int() with base 10: ''

>>>

>>> camion

[{'cajones': 100, 'nombre': 'Lima', 'precio': 32.2},

{'cajones': 50, 'nombre': 'Naranja', 'precio': 91.1},

{'cajones': 150, 'nombre': 'Caqui', 'precio': 103.44},

{'cajones': 95, 'nombre': 'Durazno', 'precio': 40.37},

{'cajones': 50, 'nombre': 'Mandarina', 'precio': 65.1}]

>>>

**Ejercicio 7.1: Errores silenciados**

Modificá parse\_csv() de modo que le usuarie pueda silenciar los informes de errores en el parseo de los datos que agregaste antes.Por ejemplo:

>>> camion = parse\_csv('../Data/missing.csv', types = [str,int,float], silence\_errors = True)

>>> camion

[{'cajones': 100, 'nombre': 'Lima', 'precio': 32.2},

{'cajones': 50, 'nombre': 'Naranja', 'precio': 91.1},

{'cajones': 150, 'nombre': 'Caqui', 'precio': 103.44},

{'cajones': 95, 'nombre': 'Durazno', 'precio': 40.37},

{'cajones': 50, 'nombre': 'Mandarina', 'precio': 65.1}]

>>>

Guardá estos cambios que los vamos a usar más adelante.

**Comentarios**

Lograr un buen manejo o administración de errores es una de las partes más difíciles en la mayoría de los programas. Estás intentando prever imprevistos. Como regla general, no silencies los errores. Es mejor informar los problemas y darle al usuarie la opción de silenciarlos explícitamente. Un buen diálogo entre el código y el usuarie facilita el debugging y el buen uso del programa.

**7.3 El módulo principal**

En esta sección introducimos el concepto de **módulo principal**. Dejamos [este video](https://youtu.be/TsWym9qYs4M) con una introducción breve a esta sección y la siguiente.

**Función principal**

En muchos lenguajes de programación existe el concepto de método o función *principal*.

// c / c++

int main(int argc, char \*argv[]) {

...

}

// java

class myprog {

public static void main(String args[]) {

...

}

}

Se refiere a la primera función que es ejecutada cuando corremos un programa.

**Módulo principal en Python**

Python no tiene una función o método principal. En su lugar existe un *módulo principal* y éste será el archivo con código fuente que se ejecuta primero.

bash % python3 prog.py

...

El archivo que le pases al intérprete al invocarlo será el módulo principal. No importa cómo lo llames.

**Chequear \_\_main\_\_**

Es una práctica estándar usar la siguiente convención en módulos que son ejecutados como scripts principales:

# prog.py

...

if \_\_name\_\_ == '\_\_main\_\_':

# Soy el programa principal ...

comandos

...

Los comandos dentro del if constituyen el *programa principal*

**Módulo principal vs. módulo importado**

Cualquier archivo .py puede ejecutarse ya sea como el programa principal o como un módulo importado:

bash % python3 prog.py # Corriendo como principal

import prog # Corriendo como módulo importado

La variable \_\_name\_\_ es el nombre del módulo. Sin embargo, esta variable \_\_name\_\_ valdrá \_\_main\_\_ si ese módulo está siendo ejecutado como el script principal.

Normalmente deseamos que los comandos que son parte del comportamiento del script en modo *principal* sólo se ejecuten si efectivamente el script es el módulo principal. No queremos que esos comandos se ejecuten si el módulo fue importado.

Por lo tanto es común escribir una condición if que decida cómo se va a portar el código cuando éste puede ser usado de ambas maneras.

if \_\_name\_\_ == '\_\_main\_\_':

# Esto no se ejecuta en un módulo importado ...

**Modelo de programa**

Éste es un modelo usual para escribir un programa en Python:

# prog.py

# Comandos import (bibliotecas o módulos)

import modules

# Funciones

def spam():

...

def blah():

...

# Función principal

def main():

...

if \_\_name\_\_ == '\_\_main\_\_':

main()

**Herramientas para la consola**

Python se usa muy frecuentemente para correr herramientas desde la línea de comandos. En clase vimos algún ejemplo:

bash % python3 informe.py camion.csv precios.csv

Esto permite que los scripts sean ejecutados desde la terminal para correr ciertos procesos automáticos, ejecutar tareas en segundo plano, etc.

**Argumentos en la línea de comandos**

Python interpreta una línea de comandos como una lista de cadenas de texto.

bash % python3 informe.py camion.csv precios.csv

Como el script informe.py no está preparado para leer parámetros, no los va a usar. Igual, podés acceder a esta lista de cadenas usando sys.argv. Por ejemplo, si usas el parámetro -i para invocar a python de modo que el intérprete interactivo no termine luego de llamar a informe.py con los parámetros anteriores

bash % python3 -i informe.py camion.csv precios.csv

luego podrás ver el contenido de esta lista:

# Llamado como recién, sys.argv contiene

import sys

sys.argv # ['informe.py, 'camion.csv', 'precios.csv']

Ahora vamos a hacer que los tenga en cuenta. El siguiente es un ejemplo de script simple para procesar los argumentos recibidos al invocarlo desde la terminal. Te permite usar tu script para generar el informe con archivos de diferentes camiones o precios, pasados como parámetros por la línea de comandos:

import sys

if len(sys.argv) != 3:

raise SystemExit(f'Uso adecuado: {sys.argv[0]} ' 'archivo\_camion archivo\_precios')

camion = sys.argv[1]

precios = sys.argv[2]

...

Para ir un poco más allá, podés mirar el módulo [argparse](https://docs.python.org/3/library/argparse.html" \l "module-argparse) de Python permite escribir interfaces para programas que corren por linea de comandos de una manera amigables y profesional.

**Standard I/O**

Los archivos de entrada y salida estándard (Standard Input / Output (stdio)) son archivos que se portan como archivos normales, pero están definidos por el sistema operativo.

sys.stdout

sys.stderr

sys.stdin

Por omisión, la salida impresa es dirigida a sys.stdout (usualmente la pantalla), la entrada se lee de sys.stdin (usualmente el teclado), y la recapitulación de errores es dirigida a sys.stderr (usualmente, la pantalla otra vez).

Las entradas y salidas de *stdio* pueden estar ligadas al teclado, a la pantalla, a una impresora, a diferentes archivos o incluir cosas más extrañas como pipes, etc.

bash % python3 prog.py > resultados.txt

# o si no

bash % cmd1 | python3 prog.py | cmd2

Esta sintaxis se llama "piping" o redireccionamiento y significa: ejecutar cmd1, enviar su salida como entrada a prog.py invocado desde la terminal, y la salida de éste será la entrada para cmd2.

**Terminación del programa**

La terminación y salida del programa se administran a través de excepciones.

raise SystemExit

raise SystemExit(codigo\_salida)

raise SystemExit('Mensaje informativo')

O, alternativamente:

import sys

sys.exit(codigo\_salida)

Es estándar que un codigo de salida de 0 indica que no hubo problemas y otro valor, que los hubo.

**El comando #!**

Bajo Unix (Linux es un Unix) una línea que comienza con #! ejecutará un script en el intérprete Python. Por ejemplo, si agregás la siguiente línea al comienzo de tu script podés ejecutar directamente el script (sin invocar manualmente a Python en la misma línea).

#!/usr/bin/env python3

# prog.py

...

Para porder ser ejecutado, el archivo prog.py requiere permiso de ejecución asignado. Podés asignarle este permiso así:

bash % chmod +x prog.py

# Ahora lo podés ejecutar

bash % ./prog.py

... salida ...

*Observación: Al iniciar un script Python en Windows, se lee la línea que comienza con #! dentro del script para saber qué versión del intérprete invocar.*

**Modelo de script con parámetros**

Para terminar, éste es un modelo usual de programa en Python que se ejecuta invocado desde la terminal.

#!/usr/bin/env python3

# prog.py

# Import (bibliotecas)

import modules

# Funciones

def spam():

...

def blah():

...

# Funcion principal

def main(parametros):

# Analizar la línea de comandos,

# usando la variable parámetros en lugar

# de sys.argv, donde corresponda

...

if \_\_name\_\_ == '\_\_main\_\_':

import sys

main(sys.argv)

*Observación: Este modelo es flexible en el sentido que te permite escribir programas que podés llamar desde la terminal pasándole parámetros o ejecutar directamente dentro de un intérprete usando import y llamando a su función main como veremos en los siguientes ejercicios.*

**Ejercicios**

Recordá trabajar siempre con las últimas versiones de tus archivos.

**Ejercicio 7.2: Función main()**

Usando estas ideas, agregá a tu programa informe.py una función main() que tome una lista de parámetros en la línea de comandos y produzca la misma salida que antes.

bash % python3 informe.py ../Data/camion.csv ../Data/precios.csv

También deberías poder ejecutarlo del siguiente modo dentro del intérprete interactivo de Python:

>>> import informe

>>> informe.main(['informe.py', '../Data/camion.csv', '../Data/precios.csv'])

Nombre Cajones Precio Cambio

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

Lima 100 $32.2 8.02

Naranja 50 $91.1 15.18

Caqui 150 $103.44 2.02

Mandarina 200 $51.23 29.66

Durazno 95 $40.37 33.11

Mandarina 50 $65.1 15.79

Naranja 100 $70.44 35.84

>>>

Análogamente, modificá el archivo costo\_camion.py para que incluya una función similar main() que te permita hacer esto:

>>> import costo\_camion

>>> costo\_camion.main(['costo\_camion.py', '../Data/camion.csv'])

Total cost: 47671.15

>>>

**Ejercicio 7.3: Hacer un script**

Finalmente, modificá tus programas informe.py y costo\_camion.py para que puedan ser ejecutados como scripts desde la línea de comandos:

bash $ python3 informe.py ../Data/camion.csv ../Data/precios.csv

Nombre Cajones Precio Cambio

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

Lima 100 $32.2 8.02

Naranja 50 $91.1 15.18

Caqui 150 $103.44 2.02

Mandarina 200 $51.23 29.66

Durazno 95 $40.37 33.11

Mandarina 50 $65.1 15.79

Naranja 100 $70.44 35.84

bash $ python3 costo\_camion.py Data/camion.csv

Costo total: 47671.15

*Aclaración:* En el ejercicio anterior ya agregaste una función main() a tu código. En este simplemente deberías verificar si \_\_name\_\_ == '\_\_main\_\_' y llamar a esa función para que se ejecute automáticamente cuando llames a tu programa desde la línea de comandos.

**7.4 Cuestiones de diseño**

En esta breve sección, volvemos a discutir algunas decisiones de diseño que tomamos antes.

**Archivos versus iterables**

Compará estos dos programas que resultan en la misma salida.

# Necesita el nombre de un archivo

def read\_data(nombre\_archivo):

records = []

with open(nombre\_archivo) as f:

for line in f:

...

records.append(r)

return records

d = read\_data('file.csv')

# Necesita líneas de texto

def read\_data(lines):

records = []

for line in lines:

...

records.append(r)

return records

with open('file.csv') as f:

d = read\_data(f)

* ¿Cuál de las funciones read\_data() preferís y por qué?
* ¿Cuál de las funciones permite mayor flexibilidad?

**Una idea profunda: "Duck Typing" (Identificación de patos)**

[Duck Typing](https://en.wikipedia.org/wiki/Duck_typing) del inglés o en español ["Test del pato"](https://es.wikipedia.org/wiki/Duck_typing) es un concepto usado en programación para determinar si un objeto puede ser usado para un propósito en particular. Se trata de una aplicación particular del [test del pato](https://en.wikipedia.org/wiki/Duck_test) que puede resumirse así:

Si algo se parece a un pato, nada como un pato, y hace el mismo ruido que un pato, entonces probablemente se trate de un pato.

Mientras que la primera versión de read\_data() requiere específicamente líneas de un archivo de texto, la segunda versión funciona con *cualquier* iterable.

def read\_data(lines):

records = []

for line in lines:

...

records.append(r)

return records

Esto implica que la podemos usar con otro tipo de *líneas*, no necesariamente archivos. Veamos algunos ejemplos.

# Un archivo .csv

lines = open('data.csv')

data = read\_data(lines)

# Un archivo zipeado

lines = gzip.open('data.csv.gz','rt')

data = read\_data(lines)

# La entrada estándar (Standard Input), por teclado

lines = sys.stdin

data = read\_data(lines)

# Una lista de cadenas

lines = ['Quinoto,50,91.1','Naranja,75,123.45', ... ]

data = read\_data(lines)

Esto nos lleva nuevamente a la identificación de patos: es suficiente con saber que grazna como pato, camina como pato y vuela como pato para saber que podés usarlo como pato. Volveremos a esta idea al hablar de diseño de objetos, dentro de un par de clases. En este caso en particular, todos nuestros "patos" ...

lines = open('data.csv')

lines = gzip.open('data.csv.gz','rt')

lines = sys.stdin

lines = ['Quinoto,50,91.1','Naranja,75,123.45', ... ]

son iterables de texto, por lo tanto los usaremos como "patos" en la función read\_data().

Si el concepto te resulta oscuro, tal vez [esta explicación](https://youtu.be/pxzLSMqU_7U) te ayude. La flexibilidad que este diseño permite es considerable. *Pregunta: ¿Debemos favorecer u oponernos a esta flexibilidad?*

**Buenas prácticas en el diseño de bibliotecas**

Las bibliotecas de código suelen ser más útiles si son flexibles. No restrinjas las opciones innecesariamente. Con mayor flexibilidad suele venir asociada una mayor potencia.

**Ejercicio**

**Ejercicio 7.4: De archivos a "objetos cual archivos"**

>>> import fileparse

>>> camion = fileparse.parse\_csv('../Data/camion.csv', types=[str,int,float])

>>>

Actualmente la función solicita el nombre de un archivo, pero podés hacer el código más flexible. Modificá la función de modo que funcione con cualquier objeto o iterable que se comporte como un archivo. Por ejemplo:

>>> import fileparse

>>> import gzip

>>> with gzip.open('../Data/camion.csv.gz', 'rt') as file:

... camion = fileparse.parse\_csv(file, types=[str,int,float])

...

>>> lines = ['nombre,cajones,precio', 'Lima,100,34.23', 'Naranja,50,91.1', 'Mburucuya,75,45.1']

>>> camion = fileparse.parse\_csv(lines, types=[str,int,float])

>>>

Atención: Si lo hacés con delicadeza, los cambios a hacer en tu código son realmente mínimos. Fijate en la documentación de la función csv.reader (por ejemplo con help(csv.reader)) que lo que le pasás debe ser un objeto iterable, pero no necesariamente un archivo.

Una vez que hayas incorporado esta modificación, ¿qué pasa si le pasás un nombre de archivo como antes?

>>> camion = fileparse.parse\_csv('../Data/camion.csv', types=[str,int,float])

>>> camion

... mirá la salida (debería ser un lío) ...

>>>

Sí, hay que tener cuidado.

**Ejercicio 7.5: Arreglemos las funciones existentes**

Arreglá las funciones leer\_camion() y leer\_precios() en el archivo informe.py de modo que funcionen con la nueva versión de parse\_csv(). Con una pequeña modificación es suficiente. Después de esto tus programas informe.py y costo\_camion.py deberían funcionar tan bien como antes.

Por ahora dejamos estos archivos y pasamos a otras discusiones. Dejá estos archivos listos para entregar al final de la clase.

**7.5 Contratos: Especificación y Documentación**

En esta unidad formalizamos algunos temas que ya mencionamos brevemente en las clases anteriores sobre la especificación y documentación de funciones. Dejamos [este video](https://youtu.be/cKVP5-Z4_RE) con una introducción breve a esta sección y la siguiente.

Trabajaremos informalmente con conceptos formales. Por ejemplo, trataremos de responder en algunos casos concretos: ¿qué condiciones debe cumplir una función al comenzar? ¿Qué condiciones se mantinen durante su ejecución? ¿Qué debemos garantizar cuando se termina de ejecutar? Y veremos algunas técnicas para tener en cuenta estas condiciones.

**Documentación**

Comenzamos formalizando un poco más algunos conceptos relacionados con la documentación, cuál es su objetivo y las distintas formas de documentar.

**Comentarios vs documentación**

En Python tenemos dos convenciones diferentes para documentar nuestro código: la *documentación* propiamente dicha (lo que ponemos entre ' o ''' al principio de cada función o módulo se llama *docstring*), y los *comentarios* (#). En la mayoría de los lenguajes de programación hay convenciones similares. ¿Por qué tenemos dos formas diferentes de documentar?

La *documentación* tiene como objetivo explicar *qué* hace el código. La documentación está dirigida a cualquier persona que desee utilizar la función o módulo, para que pueda entender cómo usarla sin necesidad de leer el código fuente. Esto es útil incluso cuando quien implementó la función es la misma persona que la va a utilizar, ya que permite separar responsabilidades.

Los *comentarios* tienen como objetivo explicar *cómo* funciona el código, y *por qué* se decidió implementarlo de esa manera. Los comentarios están dirigidos a quien esté leyendo el código fuente.

Podemos ver la diferencia entre la documentación y los comentarios en la función elegir\_codigo:

def elegir\_codigo():

'''Devuelve un codigo de 4 digitos elegido al azar'''

digitos = ('0','1','2','3','4','5','6','7','8','9')

codigo = ""

for i in range(4):

candidato = random.choice(digitos)

# Debemos asegurarnos de no repetir digitos

while candidato in codigo:

candidato = random.choice(digitos)

codigo = codigo + candidato

return codigo

**¿Por qué documentamos?**

Muchas veces se plantea el siguiente interrogante: ¿Para qué repetir con palabras lo que ya está estipulado en el código? La documentación es algo que muy a menudo se deja *para después* por resultar tedioso y quizás aburrido en el momento de escribir el código. Pero en ese *después*, está el *yo* del futuro, u otre del futuro que quiere volver a usar el código y que agradecerá esas líneas que le evitarán varios dolores de cabeza.

Es muy frecuente que durante el desarrollo de un proyecto el código evolucione con el tiempo. Si nos olvidamos de actualizar la documentación para reflejar los cambios, entonces tendremos documentación de mala calidad, ya que posiblemente esté incompleta e incluso incorrecta.

Una buena documentación es componente esencial de cualquier proyecto exitoso (NumPy, matplotlib, etc. tienen buena documentación). Esto en parte se debe a que el código fuente transmite en detalle las operaciones individuales que componen un algoritmo o programa, pero no suele transmitir en forma transparente cosas como la *intención* del programa, el *diseño* de alto nivel, las *razones* por las que se decidió utilizar un algoritmo u otro, etc. También se pueden incluir ejemplos para [clarificar su uso](https://numpy.org/doc/stable/reference/generated/numpy.resize.html).

**Código autodocumentado**

En teoría, si nuestro código pudiera transmitir en forma eficiente todos esos conceptos, la documentación sería menos necesaria. De hecho, existe una técnica de programación llamada *código autodocumentado*, en la que la idea principal es elegir los nombres de funciones y variables de forma tal que la documentación sea menos indispensable.

Tomemos como ejemplo el siguiente código:

a = 9.81

b = 5

c = 0.5 \* a \* b\*\*2

Leyendo esas tres líneas de código podemos razonar cuál será el valor final de las variables a, b y c, pero no hay nada que nos indique qué representan esas variables, o cuál es la intención del código. Una opción para mejorarlo sería utilizar comentarios para aclarar la intención:

a = 9.81 # Valor de la constante G (aceleración gravitacional), en m/s²

b = 5 # Tiempo en segundos

c = 0.5 \* a \* b\*\*2 # Desplazamiento (en metros)

Otra opción, según la técnica de código autodocumentado, es simplemente asignar nombres descriptivos a las variables:

aceleracion\_gravitacional = 9.81

tiempo\_segundos = 5

desplazamiento\_metros = 0.5 \* aceleracion\_gravitacional \* tiempo\_segundos \*\* 2

De esta manera logramos que la intención del código esté más clara, y que se reduzca la necesidad de comentarios y documentación para comprenderlo.

La técnica de código autodocumentado presenta varias limitaciones. Entre ellas:

* Elegir buenos nombres es una tarea difícil, que requiere tener en cuenta cosas como: qué tan descriptivo es el nombre (cuanto más, mejor), la longitud del identificador (no debe ser excesivamente largo), el alcance del identificador (cuánto más grande, más descriptivo debe ser el nombre), y convenciones (i para índices, c para caracteres, etc).
* La documentación de todas formas termina siendo necesaria, ya que por muy bien que elijamos los nombres, muchas veces la única forma de explicar la intención del código y todos sus detalles es en lenguaje coloquial.
* En ciertos contextos sigue siendo deseable, o imprescindible, que quien quiera utilizar nuestra función o módulo pueda entender su funcionamiento sin necesidad de leer el código fuente.

**Un error común: la sobredocumentación**

Si bien la ausencia de documentación suele ser perjudicial, el otro extremo también lo es: la *sobredocumentación*. Después de todo, en la vida diaria no necesitamos carteles que nos recuerden cosas como "esta es la puerta", "este es el picaporte" y "empujar hacia abajo para abrir". De la misma manera, podríamos decir que el siguiente código peca de ser sobredocumentado:

def buscar\_elemento(lista\_de\_numeros, numero):

'''Esta función devuelve el índice (contando desde 0) en el que se

encuentra el número `numero` en la lista `lista\_de\_numeros`.

Si el elemento no está en la lista devuelve -1.

'''

# Recorremos todos los índices de la lista, desde 0 (inclusive) hasta N

# (no inclusive)

for indice in range(len(lista\_de\_numeros)):

# Si el elemento en la posicion `indice` es el buscado

if lista\_de\_numeros[indice] == numero:

# Devolvemos el índice en el que está el elemento

return indice

# No lo encontramos, devolvemos -1

return -1

Algunas cosas que podemos mejorar:

* En la firma de la función los nombres buscar\_elemento, lista\_de\_numeros y numero se pueden simplificar a indice, secuencia y elemento. Cambiamos lista\_de\_numeros por lista, ya que la función puede recibir secuencias de cualquier tipo, con elementos de cualquier tipo, y no hay ninguna razón para limitar a que sea una lista de números.
* Las variable interna indice también se puede simplificar: por convención podemos usar i.
* "Esta función" es redundante: cuando alguien lea la documentación ya va a saber que se trata de una función.
* "contando desde 0" es redundante: en Python siempre contamos desde 0.
* Los comentarios son excesivos: la función es suficientemente simple y cualquier persona que sepa programación básica podrá entender el algoritmo.

Corrigiendo todos estos detalles resulta:

def indice(lista, elemento):

'''Devuelve el índice en el que se encuentra el `elemento` en la `lista`,

o -1 si no está.

'''

for i in range(len(lista)):

if lista[i] == elemento:

return i

return -1

**Contratos**

Cuando hablamos de **contratos** o *programación por contratos*, nos referimos a una forma de estipular tanto lo que nuestro código asume sobre los parámetros, como lo que devuelve. En los contratos se establecen compromisos que garantizan que si se cumplen los requisitos estipulados, la función devolverá cierto resultado. Es bueno que el contrato de una función esté incluido en su documentación.

Algunos ejemplos de cosas que deben ser estipuladas como parte del contrato son: cómo deben ser los parámetros recibidos, qué va a ser lo que se devuelve, y si la función provoca algún efecto secundario (como por ejemplo modificar alguno de los parámetros recibidos).

Las condiciones que se deben cumplir al momento de ejecutar el código o función se llaman *precondiciones*. Si se cumplen las precondiciones, el código se ejecutará de manera que al finalizar su ejecución el estado final de las variables y de valor de retorno, estarán caracterizados en una *poscondición*.

**Precondiciones**

La precondición de una función debe cumplirse antes de ejecutarla para que se comporte correctamente: cómo deben ser los parámetros que recibe, cómo debe ser el estado global, etc. Si no se cumplen, no hay garantías del funcionamiento del código (podría colgarse, o dar error, o peor aún dar resultados erróneos).

Por ejemplo, en una función que realiza la división entre dos números, la precondición debe decir que ambos parámetros deben ser números, y que el divisor debe ser distinto de 0.

Si incluimos las precondiciones como parte de la documentación, en el cuerpo de la función podremos asumir que son ciertas, y no será necesario escribir código para lidiar con los casos en los que no se cumplan.

**Poscondiciones**

La poscondición caracterizará cómo será el valor de retorno y cómo se modificarán las variables de entrada (en caso de que corresponda) al finalizar la ejecución siempre asumiendo que se cumplió la precondición al inicio.

En el ejemplo anterior, la función división nos garantiza que si se satisface la precondición, la función devolverá un número y éste será el cociente solicitado.

**El qué, no el cómo**

Notar que al especificar un problema con pre y poscondición estamos definiendo qué es lo que debe suceder. En ningún momento decimos cómo es que esto sucede. Para una misma especificación podemos definir varias funciones que cumplan el contrato, y cada una puede resolverlo a su manera.

Por ejemplo, volviendo a los ejemplos que trabajamos antes, mirá la documentación del csv.reader. Deberías nota que la precondición dice que el primer parámetro debe ser un interable y no un archivo específicamente.

**Aseveraciones**

Retomamos acá el concepto de aseveración que introdujimos en la [Sección 4.1](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/01_Debugger.md#aseveraciones-assert). Tanto las precondiciones como las poscondiciones pueden pensarse como *aseveraciones* (en inglés *assertions*). Es decir, afirmaciones realizadas en un momento particular de la ejecución sobre el estado computacional. Si una aseveración llegara a ser falsa, se levanta una excepción interrumpiendo la normal ejecución del programa.

En algunos casos puede ser útil incorporar estas afirmaciones desde el código, y para eso podemos utilizar la instrucción assert. Esta instrucción recibe una condición a verificar (o sea, una expresión booleana). Si la condición es True, la instrucción no hace nada; en caso contrario se produce un error.

>>> assert True

>>> assert False

(Traceback (most recent call last):

File '<stdin>', line 1, in <module>

AssertionError^)

Opcionalmente, la instrucción assert puede recibir un mensaje de error que se mostrará en caso de que la condición no se cumpla.

>>> x = 0

>>> assert x != 0, 'El divisor no puede ser 0'

(^Traceback (most recent call last):

File '<stdin>', line 1, in <module>

AssertionError: El divisor no puede ser 0)

**Atención:** Es importante tener en cuenta que assert está pensado para ser usado en la etapa de desarrollo. Un programa terminado nunca debería dejar de funcionar por este tipo de errores.

**Ejemplos**

Usando los ejemplos anteriores, la función division nos quedaría de la siguiente forma:

def division(dividendo, divisor):

'''Cálculo de la división

Pre: Recibe dos números, divisor debe ser distinto de 0.

Pos: Devuelve un número real, con el cociente de ambos.

'''

assert divisor != 0, 'El divisor no puede ser 0'

return dividendo / divisor

o directamente

def division(dividendo, divisor):

'''Cálculo de la división

Pre: Recibe dos números, divisor debe ser distinto de 0.

Pos: Devuelve un número real, con el cociente de ambos.

'''

return dividendo / divisor

Es interesante discutir un poco en detalle este ejemplo. La función *asume* que el divisor es no nulo. Esto tiene sentido, ya que no podemos dividir por cero. Podríamos atrapar el error y, si nos pasan un divisor nulo, devolver por ejemplo *cero*. De esta forma evitamos que se termine el programa. Pero ¿tiene sentido esto? ¿Nos ahorra un problema o nos genera un nuevo problema? No es una buena práctica atrapar errores que no sabemos manejar. Que 1/0 devuelva cero en principio **no es correcto**. Como ya mencionamos en la [Sección 7.2](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/02_Excepciones.md#buenas-pr%C3%A1cticas-al-administrar-excepciones), es mejor que los errores generen excepciones ruidosamente y no atraparlas si no sabemos exactamente cómo manejarlas.

Veamos otro ejemplo, tal vez más interesante. Consideremos una función sumar\_enteros(desde, hasta) que implementa la sumatoria *sum\_i=desde^hasta i*.

En este caso, analizando los parámetros que recibirá la función, podemos definir la precondición para indicar lo que éstos deberán cumplir.

La función sumar\_enteros tomará un valor desde y un valor hasta. Es decir que recibe dos parámetros.

def sumar\_enteros(desde, hasta):

Tanto desde como hasta deben ser números enteros, y dependiendo de la implementación a realizar o de los requisitos con los que nos enfrentamos al escribir la precondición puede ser necesario exigir que hasta sea mayor o igual a desde.

La declaración de la función, incluyendo documentación, precondición y poscondición queda de la siguiente manera.

def sumar\_enteros(desde, hasta):

'''Calcula la sumatoria de los números entre desde y hasta.

Si hasta < desde, entonces devuelve cero.

Pre: desde y hasta son números enteros

Pos: Se devuelve el valor de sumar todos los números del intervalo

[desde, hasta]. Si el intervalo es vacío se devuelve 0

'''

Prestá atención a que tanto la pre como la pos no dicen cómo hace la función para resolver el problema, sino que caracterizan el resultado. La implementación (o código) serán el cómo. En este caso puede ser con un ciclo que emule los pasos de dichas sumas, podría utilizarse una fórmula cerrada que calcule el valor sin utilizar un ciclo, entre otras opciones. Lo importante es ver que a fines de la especificación, esto no importa.

En definitiva, la estipulación de pre y poscondiciones dentro de la documentación de las funciones es una forma de definir claramente el comportamiento del código. Son, en efecto, un *contrato* entre el código invocante (o usuarie) y el código invocado (o función).

**Ejercicio 7.6: Sumas**

En este ejercicio vas a realizar dos implementaciones correspondientes a la función sumar\_enteros definida recién.

1. En la primera implementación te pedimos que uses un ciclo.
2. En la segunda te pedimos que lo hagas sin ciclos: implementá la función de manera que trabaje en tiempo constante (es decir, usando una cantidad de operaciones que no depende de las entradas a la función.

*Ayuda: Estas sumas se pueden escribir como diferencia de dos*[*números triangulares*](https://es.wikipedia.org/wiki/N%C3%BAmero_triangular)*.*

**Invariantes de ciclo**

Los invariantes se refieren a estados o condiciones que no cambian dentro de un contexto o porción de código. Hay invariantes de ciclo, que son los que veremos a continuación, e invariantes de estado, que se verán más adelante.

Un invariante de ciclo es una aseveración que debe ser verdadera al comienzo de cada iteración del ciclo y al salir del mismo.

Por ejemplo, si el problema es ir desde el punto A al punto B, la precondición dice que tenemos que estar parados en A y la poscondición que al terminar estaremos parados en B. En este caso las siguientes aseveraciones son invariantes: "estamos en algún punto entre A y B", "estamos en el punto más cercano a B que estuvimos hasta ahora". Son aseveraciones que podría tener nuestro código (y dependen exclusivamente de cómo lo programamos).

Pensar en términos de invariantes de ciclo nos ayuda a reflexionar y comprender mejor qué es lo que debe realizar nuestro código y nos ayuda a desarrollarlo.

Por ejemplo, para la función maximo, que busca el valor más grande de una lista desordenada, podemos enunciar:

* precondición: la lista contiene elementos que tienen una relación de orden (son comparables con <)
* poscondición: se devolverá el elemento máximo de la lista, si es que tiene elementos, y si no se devolverá None.

def maximo(lista):

'Devuelve el elemento máximo de la lista o None si está vacía.'

if not lista:

return None

max\_elem = lista[0]

for elemento in lista:

if elemento > max\_elem:

max\_elem = elemento

return max\_elem

En este caso, el invariante del ciclo es que max\_elem contiene el valor máximo de la porción de lista que ya fue analizada.

Los invariantes son de gran importancia al momento de demostrar formalmente que un algoritmo funciona, pero aún cuando no hagamos una demostración formal resulta útil tener los invariantes a la vista, ya que de esta forma es más fácil entender cómo funciona un algoritmo y encontrar posibles errores.

Los invariantes, además, son útiles a la hora de determinar las condiciones iniciales de un algoritmo, ya que también deben cumplirse para ese caso. Por ejemplo, consideremos el algoritmo para obtener la potencia n de un número.

def potencia(base, exp):

'Calcula la potencia exp del número base, con exp entero mayor que 0.'

resultado = 1

for i in range(exp):

resultado \*= base

return resultado

En este caso, el invariante del ciclo es que la variable resultado contiene el valor de la potencia correspondiente al índice i de la iteración. Teniendo en cuenta esta condición, es fácil ver que resultado debe comenzar el ciclo con un valor de 1, ya que ese es el valor correspondiente a base \*\* 0.

De la misma manera, si la operación que se quiere realizar es sumar todos los elementos de una lista, el invariante será que una variable suma contenga la suma de todos los elementos ya recorridos. Antes de empezar a recorrer la lista, según lo expresado en este invariante, esta suma debe ser 0 ya que no recorrió ningún elemento.

def suma(lista):

'Devuelve la suma de todos los elementos de la lista.'

suma = 0

for elemento in lista:

suma += elemento

return suma

En resumen, el concepto de invariante de ciclo es una herramienta que nos permite comprender (explicitar) mejor cómo funciona un algoritmo. Resulta fundamental en la teoría de algoritmos, donde es necesario para *demostrar* que:

* un algoritmo es correcto, es decir que realiza la tarea descripta por la pre y poscondición.
* un algoritmo termina (y no se cuelga).

**Ejercicio 7.7: Invariante en sumas**

En el [Ejercicio 7.6](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/05_Especificacion_y_Documentacion.md#ejercicio-76-sumas), escribiste una función sumar\_enteros(desde, hasta) que utiliza un ciclo. ¿Cuál es el invariante de este ciclo?

**Parámetros mutables e inmutables**

Las funciones reciben parámetros que pueden ser mutables o inmutables.

Si dentro del cuerpo de la función se modifica uno de estos parámetros para que *referencie* a otro valor, este cambio no se verá reflejado fuera de la función. Si, en cambio, se modifica el *contenido* de alguno de los parámetros mutables, este cambio *sí* se verá reflejado fuera de la función.

A continuación vemos un ejemplo en el cual se asigna la variable recibida a un nuevo valor. Esa asignación sólo tiene efecto dentro de la función.

>>> def no\_cambia\_lista(lista):

... lista = [0, 1, 2, 3]

... print('Dentro de la funcion lista =', lista)

...

>>> lista = [10, 20, 30, 40]

>>> no\_cambia\_lista(lista)

Dentro de la funcion lista = [0, 1, 2, 3]

>>> lista

[10, 20, 30, 40]

A continuación un ejemplo en el cual se modifica la variable recibida. En este caso, los cambios realizados tienen efecto tanto dentro como fuera de la función.

>>> def cambia\_lista(lista):

... for i in range(len(lista)):

... lista[i] = lista[i] \*\* 3

...

>>> lista = [1, 2, 3, 4]

>>> cambia\_lista(lista)

>>> lista

[1, 8, 27, 64]

**Atención:** Salvo que sea explícitamente aclarado, una función no debe modificar los valores de sus parámetros. En el caso en que por una decisión de diseño o especificación se modifiquen los parámetros mutables recibidos, esto debe estar claramente documentado como parte de las poscondiciones.

**Repaso**

Dejamos un par de videos. En el [primer video](https://youtu.be/k-QC8WAmWr0) discutimos brevemente la importancia de establecer contratos y explicamos cómo estos y los invariantes pueden ayudarnos a *demostrar matemáticamente* que una función hace lo que creemos que hace. En el [segundo video](https://youtu.be/uRgd9y0QNpM) damos algunos ejemplos de invariantes de ciclos discutidos paso a paso y hablamos de pre- y post-condiciones.

**Resumen**

* El **contrato** de una función especifica qué condiciones se deben cumplir para que la función pueda ser invocada (**precondición**), y qué condiciones se garantiza que serán válidas cuando la función termine su ejecución (**poscondición**).
* La **documentación** tiene como objetivo explicar *qué* hace el código, y está dirigida a quien desee utilizar la función o módulo.
* Es una buena práctica incluir el contrato en la documentación.
* Si una función modifica un valor mutable que recibe por parámetro, eso debe estar explícitamente aclarado en su documentación.
* Los **comentarios** tienen como objetivo explicar *cómo* funciona el código y *por qué* se decidió implementarlo de esa manera, y están dirigidos a quien esté leyendo el código fuente.
* Los **invariantes de ciclo** son las condiciones que deben cumplirse al comienzo de cada iteración de un ciclo.

**Ejercicios**

**Ejercicio 7.8: Funciones y documentación**

Para cada una de las siguientes funciones:

* Pensá cuál es el contrato de la función.
* Agregale la documentación adecuada.
* Comentá el código si te parece que aporta.
* Detectá si hay invariantes de ciclo y comentalo al final de la función

Guardá estos códigos con tus modificaciones en el archivo documentacion.py.

def valor\_absoluto(n):

if n >= 0:

return n

else:

return -n

def suma\_pares(l):

res = 0

for e in l:

if e % 2 ==0:

res += e

else:

res += 0

return res

def veces(a, b):

res = 0

nb = b

while nb != 0:

#print(nb \* a + res)

res += a

nb -= 1

return res

def collatz(n):

res = 1

while n!=1:

if n % 2 == 0:

n = n//2

else:

n = 3 \* n + 1

res += 1

return res

**7.6 Estilos de codeo**

**PEP 8 - La guía de estilo para Python**

La comunidad de usuaries de Python ha adoptado una guía de estilo que facilita la lectura del código y la consistencia entre programas de distintos usuaries. Esta guía no es de seguimiento obligatorio, pero es altamente recomendable. El documento completo se denomina PEP 8 y está escrito originalmente en [inglés](https://www.python.org/dev/peps/pep-0008/), aunque hay alguna traducción al [castellano](http://recursospython.com/pep8es.pdf).

A continuación presentamos un resumen con solo algunas recomendaciones.

**Indentación**

Utilizar siempre 4 espacios y nunca mezclar tabuladores y espacios.

Si se continúa una línea hay dos opciones aceptables:

# Correcto

# opción 1, indentar a la apertura del paréntesis:

foo = funcion\_que\_crea\_bar(variable\_1, variable2,

variable\_3, variable\_4)

# opcion 2, agregar 4 espacios:

foo = funcion\_que\_crea\_bar(

variable\_1, variable2,

variable\_3)

# Incorrecto, en cualquier lado.

foo = funcion\_que\_crea\_bar(variable\_1, variable2,

variable\_3)

**Tamaño máximo de línea**

Las líneas deben limitarse a un máximo de 79 caracteres.

**Líneas en blanco**

Separar las definiciones de las clases y funciones con dos líneas en blanco. Los métodos dentro de clases se separan con una línea en blanco. Se recomienda utilizar líneas en blanco para separar partes del código, por ejemplo dentro de una función, que realizan tareas diferenciadas.

**Imports**

Los imports de distintos módulos deben estar en líneas diferentes:

# Sí:

import os

import sys

# No:

import os, sys

Sí se pueden poner en una línea los elementos que se importan de un mismo módulo:

from subprocess import Popen, PIPE

Los imports deben ponerse siempre al principio del archivo, justo después de los comentarios y de la documentación del archivo y antes de la definición de las variables globales y las constantes.

Los imports deben agruparse en el siguiente orden:

1. bibliotecas o módulos estándar.
2. bibliotecas o módulos de terceros.
3. bibliotecas o módulos locales o propios.

Cada grupo de imports debe estar separado por una línea en blanco.

**Espacios en blanco en expresiones**

Evitar espacios en blanco extra en:

Dentro de paréntesis, corchetes o llaves.

# Sí:

spam(ham[1], {eggs: 2})

# No:

spam( ham[ 1 ], { eggs: 2 })

Antes de una coma.

# Sí:

if x == 4: print x, y; x, y = y, x

# No:

if x == 4 : print x , y ; x , y = y , x

Antes del paréntesis de una llamada a una función.

# Sí:

spam(1)

# No, ese espacio es espantoso

spam (1)

Antes del corchete de un índice o clave.

# Sí:

dict['key'] = list[index]

# No, ese espacio es igual de espantoso que el anterior

dict ['key'] = list [index]

Siempre separá los operadores binarios con un espacio simple a ambos lados: asignación (=), asignación aumentada (+=, -= , etc.), comparación (==, <, >, !=, <>, <=, >=, in, not in, is, is not), booleanos (and, or, not).

Usá espacios alrededor de operadores artiméticos:

# Sí:

i = i + 1

submitted += 1

x = x \* 2 - 1

hypot2 = x \* x + y \* y

c = (a + b) \* (a - b)

# No:

i=i+1

submitted +=1

x = x\*2 - 1 #no es recomendado pero a veces lo usamos

hypot2 = x\*x + y\*y

c = (a+b) \* (a-b)

**Convenciones de nombres**

Las convenciones de nombres en Python son un lío y probablemente nunca lograremos que todo sea consistente. Sin embargo, te damos algunas de las recomendaciones actuales sobre nombres. Los nuevos módulos deberían ser escritos respetándolos, aunque la consistencia interna es preferible para bibliotecas que ya tengan partes hechas...

**Estilos de nombres**

Hay muchos estilos para nombrar variable, funciones, etc. Es útil reconocer qué estilo se está usando, independientemente de para qué se está usando.

Éstos son algunos estilos:

* b (una sola letra, en minúscula)
* B (una sola letra, en mayúscula)
* minusculas
* minusculas\_con\_guiones\_bajos
* MAYUSCULAS
* MAYUSCULAS\_CON\_GUIONES\_BAJOS
* PalabrasConMayusculas (también llamado estilo camello por las jorobas)
* mixedCase (difiere del camello en la inicial)
* Con\_Mayusculas\_Y\_Guiones\_Bajos (horrible!)

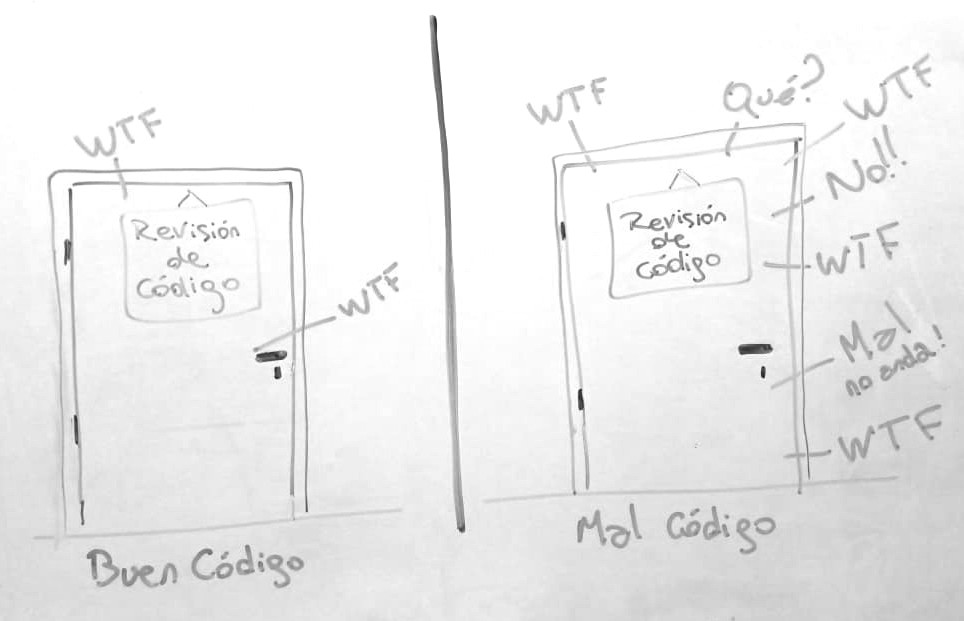
Se recomienda no usar acentos ni caracteres especiales de ningún tipo para evitar problemas de compatibilidadd. Los nombres de funciones y variables deberían estar escritos en minúsculas, eventualmente usando guiones bajos para mejorar la legibilidad.

**Hay mucho más!**

Esto es solo un breve resumen, mirá el [PEP 8](https://www.python.org/dev/peps/pep-0008/) para tener toda la información sobre estilo recomendado en Python.

**El código no es solo leído: el código de percibe**

Más allá de los gustos por los diferentes estilos posibles, es importante recordar que hay una métrica que no falla nunca para evaluar la calidad de un código: *la cantidad de puteadas por minuto de un lector externo*.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/wtf.jpg)

Las reglas de estilo pueden ser discutidas, acatadas o no, en diferentes contextos. Es importante entender que un programador enfrentado a un código en Python no solo lee el código sino que percibe su diseño en el espacio, el uso de bloques y espacios, de indentaciones y mayúsculas. El diseño gráfico del código es una parte importante de las herramientas de transmisión de la información que permite nuestro archivo.

Creeamos estas ideas con tres citas en inglés, extraídas de [este excelente artículo](https://optimal-codestyle.github.io/) de Aleksandr Skobelev que expresan la convicción de que el código en un lenguaje de programación no se escribe solamente para ser ejecutado por una computadora sino principalmente para ser leído y *percibido* por otras personas:

Usually, when evaluating the text of a program in terms of its *ease of perception*, the term *readability* is used. Strictly speaking, they are not exactly the same thing, because, as will be shown later, the process of *perceiving a program* is more than just reading. -- Aleksandr Skobelev, Fundamentals of Optimal Code Style.

First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. -- Harold Abelson and Gerald Jay Sussman, Structure and Interpretation of Computer Programs.

Indeed, the ratio of time spent reading vs. writing is well over 10:1… Because this ratio is so high, we want the reading of code to be easy, even if it makes the writing harder. -- Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship.

**Zen de Pyhton**

Ya que estamos hablando de los PEPs, queremos mencionar el PEP 20 (PEP viene de Python Enhancement Proposals), también conocido como el [Zen de Python](https://es.wikipedia.org/wiki/Zen_de_Python)

El Zen de Python es una colección de principios de software que influyen en el diseño del lenguaje. El texto, que copiamos a continuación se puede encontrar en el sitio oficial de Python y también se incluye como sorpresa en el intérprete de Python al escribir la instrucción import this.​

**Zen de Pyhton**

Bello es mejor que feo.

Explícito es mejor que implícito.

Simple es mejor que complejo.

Complejo es mejor que complicado.

Plano es mejor que anidado.

Espaciado es mejor que denso.

La legibilidad es importante.

Los casos especiales no son lo suficientemente especiales como para romper las reglas.

Sin embargo la practicidad le gana a la pureza.

Los errores nunca deberían pasar silenciosamente.

A menos que se silencien explícitamente.

Frente a la ambigüedad, evitá la tentación de adivinar.

Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.

A pesar de que esa manera no sea obvia a menos que seas Holandés.

Ahora es mejor que nunca.

A pesar de que nunca es muchas veces mejor que *justo* ahora.

Si la implementación es difícil de explicar, es una mala idea.

Si la implementación es fácil de explicar, puede que sea una buena idea.

Los espacios de nombres son una gran idea, ¡hagamos más de ellos!

**7.7 La biblioteca matplotlib**

Matplotlib es probablemente la biblioteca de Python más usada para crear gráficos en 2D, también llamados plots. Provee una forma rápida de graficar datos en varios formatos de alta calidad que pueden ser compartidos y/o publicados. En esta sección vamos a ver los usos más comunes de matplotlib. En [este video](https://youtu.be/c7fR1KkvKFQ) encontrarán una introducción breve a esta sección.

**pyplot**

*pyplot* proporciona una interfase a la biblioteca de matplotlib. Pyplot está diseñada siguiendo el estilo de Matlab y la mayoría de los comandos para graficar en pyplot tienen análogos en Matlab con argumentos similares. Explicaremos las instrucciones más importantes con ejemplos interactivos.

from matplotlib import pyplot as plt

**Un plot simple**

Para empezar, vamos a plotear las funciones *seno* y *coseno* en el mismo gráfico. Partiendo de la configuración básica, vamos a ir cambiando el gráfico paso por paso para que quede como queremos.

Primero hay que obtener los datos para graficar:

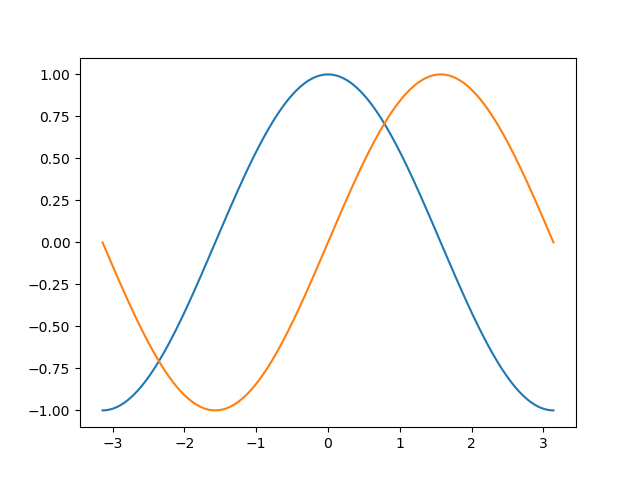
import numpy as np

X = np.linspace(-np.pi, np.pi, 256)

C, S = np.cos(X), np.sin(X)

Ahora tenemos un array de numpy con 256 valores que van desde -π a +π (incluído). C tiene los valores del coseno (256 valores) y S tiene los valores del seno (256 valores).

**El ploteo estándar**

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_exercise_1_001.png)

En Matplotlib los gráficos tienen una configuración por omisión. Cambiándolas podés configurar muchas propiedades del gráfico. Podés cambiar el tamaño de la figura, los DPI (viene de dots per inch, puntos por pulgada, y determina la resolución), el tamaño, color y estilo del trazo, las propiedades de los ejes y el cuadriculado, los textos y sus propiedades, etc.

import numpy as np

import matplotlib.pyplot as plt

X = np.linspace(-np.pi, np.pi, 256)

C, S = np.cos(X), np.sin(X)

plt.plot(X, C)

plt.plot(X, S)

plt.show()

**Un gráfico básico**

En el siguiente script, hemos explicitado y comentado todas las propiedades de una figura que influyen en la apariencia de un gráfico.

Cada propiedad se configuró a un valor típico y cercano al valor por omisión. Podés modificarlos y jugar con ellos para ver sus efectos sobre el gráfico. Sobre propiedades y estilos de las líneas hablaremos luego.

import numpy as np

import matplotlib.pyplot as plt

# Crea una figura nueva, de 8x6 pulgadas, con 80 puntos por pulgada

plt.figure(figsize=(8, 6), dpi=80)

# Crea un nuevo subplot, en una grilla de 1x1

plt.subplot(1, 1, 1)

X = np.linspace(-np.pi, np.pi, 256)

C, S = np.cos(X), np.sin(X)

# Plotea el coseno con una línea azul contínua de ancho 1 (en pixeles)

plt.plot(X, C, color="blue", linewidth=1.0, linestyle="-")

# Plotea el seno con una línea verde contínua de ancho 1 (en pixeles)

plt.plot(X, S, color="green", linewidth=1.0, linestyle="-")

# Rango del eje x

plt.xlim(-4.0, 4.0)

# Ponemos marcas (ticks) en el eje x

plt.xticks(np.linspace(-4, 4, 9))

# Rango del eje y

plt.ylim(-1.0, 1.0)

# Ponemos marcas (ticks) en el eje y

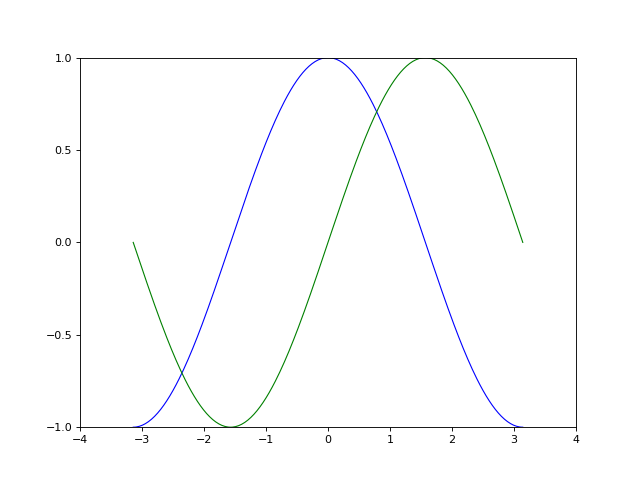
plt.yticks(np.linspace(-1, 1, 5))

# Podemos grabar el gráfico (con 72 dpi)

# plt.savefig("ejercicio\_2.png)", dpi=72)

# Mostramos el resultado en pantalla

plt.show()

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_exercise_2_001.png)

Los gráficos que genera matplotlib son muy flexibles, te dejamos [un machete](https://github.com/matplotlib/cheatsheets/blob/master/cheatsheets.pdf) resumiendo las variaciones más usuales.

A continuación presentamos detalles técnicos de esta biblioteca tan útil. No hace falta que te los aprendas (igual te los vas a olvidar), ni que pruebes todas las combinaciones. Podés volver a esta página o a la [documentación](https://matplotlib.org/) cuando lo necesites. Iguál mirá los ejercicios al final de esta sección, te pediremos que entregues el segundo.

**Detalles de un plot simple**

**Cómo cambiar los colores y ancho de los trazos**

Ahora vamos a modificar el gráfico para que quede un poco mejor. Primero, queremos trazar el coseno en azul y el seno en rojo, y ambos con una línea algo más gruesa. Además, vamos a cambiar un poco el tamaño de la figura para hacerla apaisada. Corré el siguiente código y compará el resultado con la figura anterior.

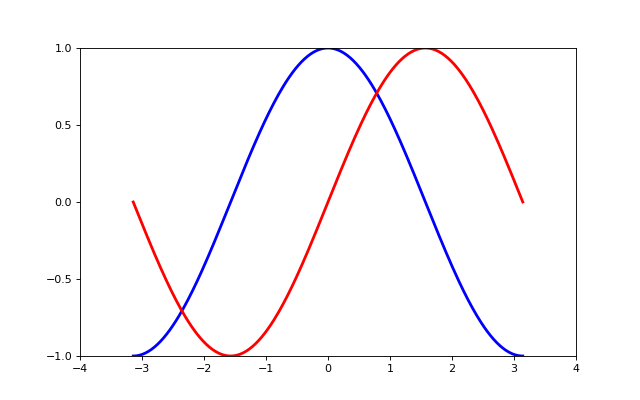
...

plt.figure(figsize=(10, 6), dpi=80)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")

plt.plot(X, S, color="red", linewidth=2.5, linestyle="-")

...

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_exercise_3_001.png)

**Límites de los ejes**

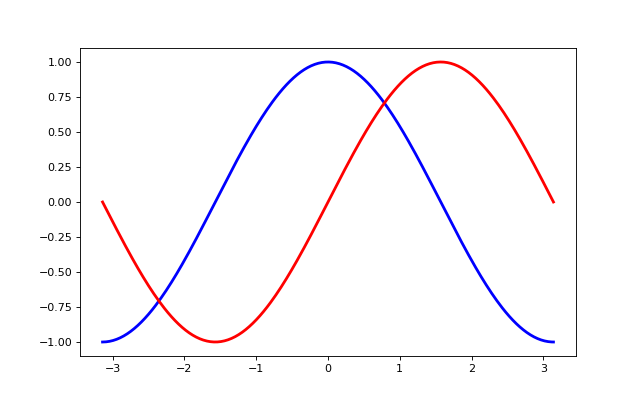
El rango de valores de los ejes es un poco angosto y necesitamos más espacio alrededor para ver claramente todos los puntos.

...

plt.xlim(X.min() \* 1.1, X.max() \* 1.1)

plt.ylim(C.min() \* 1.1, C.max() \* 1.1)

...

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_exercise_4_001.png)

**Marcas en los ejes**

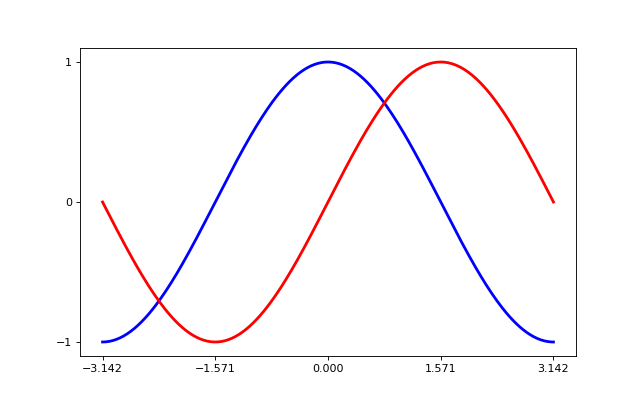
Así como están, las marcas sobre los ejes no son lo más útil. Sería bueno destacar los valores interesantes para seno y coseno (+/-π,+/-π/2). Cambiémoslos para mostrar únicamente esos valores.

...

plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi])

plt.yticks([-1, 0, +1])

...

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_exercise_5_001.png)

**Texto de las marcas en los ejes**

Las marcas en los ejes ahora están donde los queremos, pero el texto no es muy explícito. Aunque podemos darnos cuenta que 3.142 es π sería mejor dejarlo explícito.

Al definir un valor para las marcas en los ejes podemos proveer un texto en la segunda lista de argumentos para usar como etiqueta. Fijate que vamos a usar [*LaTeX*](https://es.wikipedia.org/wiki/LaTeX) para hacer que los símbolos tengan mejor pinta (otro de los geniales inventos de Donald Knuth, el mismo acuñó el término *análisis de algoritmos*).

...

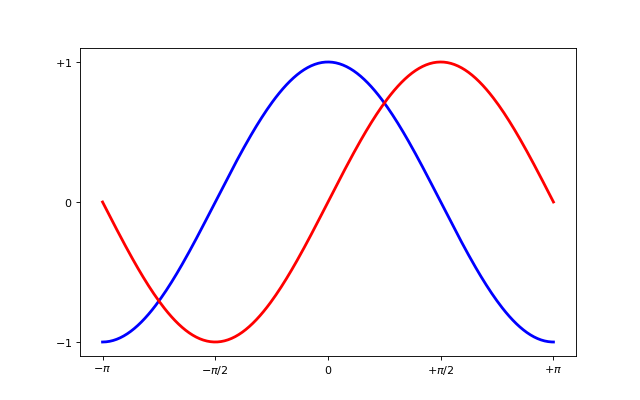
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],

[r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])

plt.yticks([-1, 0, +1],

[r'$-1$', r'$0$', r'$+1$'])

...

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_exercise_6_001.png)

**Movamos el contorno**

El contorno es el conjunto de líneas que delimitan el área de graficación y que unen todas las marcas en los ejes. Podemos ubicarlas en cualquier posición y, hasta ahora, han estado en el extremo de cada eje. Cambiemos eso, así las ubicamos en el centro. Como hay cuatro (arriba, abajo, izquierda y derecha) vamos a esconder dos de ellas dándoles color none y vamos a mover la de abajo y la de la izquierda a la posición 0 del espacio de coordenadas.

...

ax = plt.gca() # gca es 'get current axis' ó 'tomar eje actual'

ax.spines['right'].set\_color('none')

ax.spines['top'].set\_color('none')

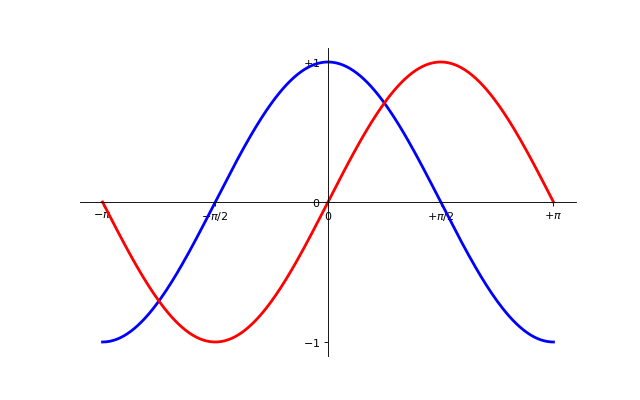
ax.xaxis.set\_ticks\_position('bottom')

ax.spines['bottom'].set\_position(('data',0))

ax.yaxis.set\_ticks\_position('left')

ax.spines['left'].set\_position(('data',0))

...

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_exercise_7_001.png)

**Pongámosle un título**

Pongámosle nombres a los trazos al gráfico en la esquina superior izquierda. Para esto alcanza con agregar a la instrucción plot la palabra clave label y ese texto será usado para el recuadro con los nombres.

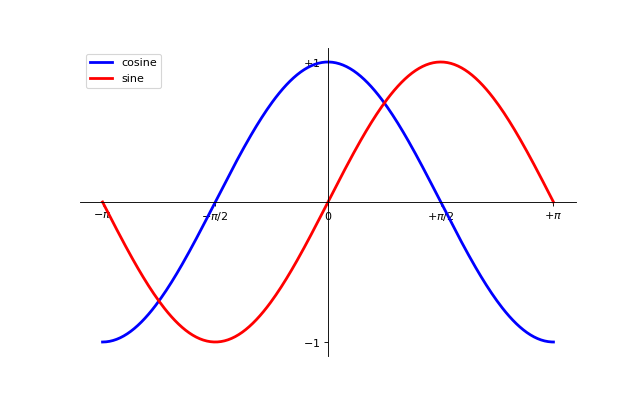
...

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="coseno")

plt.plot(X, S, color="red", linewidth=2.5, linestyle="-", label="seno")

plt.legend(loc='upper left')

...

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_exercise_8_001.png)

**Algunos puntos interesantes**

Vamos a marcar algunos puntos interesantes usando el comando annotate. Elegimos el valor 2π/3 y queremos marcar tanto el seno como el coseno. Vamos a dibujar una marca en la curva y una línea recta punteada. Además, vamos a usar annotate para mostrar texto y una flecha para destacar el valor de las funciones.

...

t = 2 \* np.pi / 3

plt.plot([t, t], [0, np.cos(t)], color='blue', linewidth=2.5, linestyle="--")

plt.scatter([t, ], [np.cos(t), ], 50, color='blue')

plt.annotate(r'$cos(\frac{2\pi}{3})=-\frac{1}{2}$',

xy=(t, np.cos(t)), xycoords='data',

xytext=(-90, -50), textcoords='offset points', fontsize=16,

arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))

plt.plot([t, t],[0, np.sin(t)], color='red', linewidth=2.5, linestyle="--")

plt.scatter([t, ],[np.sin(t), ], 50, color='red')

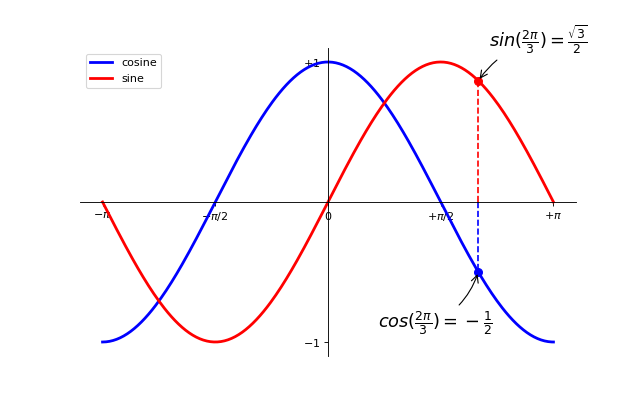
plt.annotate(r'$sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$',

xy=(t, np.sin(t)), xycoords='data',

xytext=(+10, +30), textcoords='offset points', fontsize=16,

arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))

...

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_exercise_9_001.png)

**El diablo está en los detalles**

Notá (vas a tener que mirar muy de cerca) que los ejes tapan los trazos de las funciones seno y coseno, y éstas tapan los valores escritos sobre los ejes. Si esto fuera una publicación quedaría feo.

Podemos hacer más grandes las marcas y los textos y ajustar sus propiedades de modo que tengan sean semi-transparentes. Esto nos permitirá ver un poco mejor los datos y los textos.

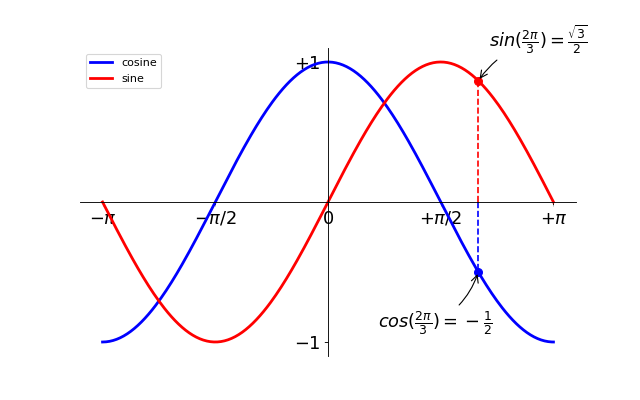
...

for label in ax.get\_xticklabels() + ax.get\_yticklabels():

label.set\_fontsize(16)

label.set\_bbox(dict(facecolor='white', edgecolor='None', alpha=0.65))

...

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_exercise_10_001.png)

**Figuras, subplots, ejes y marcas (ticks)**

En matplotlib el término "figura" se refiere a toda la ventana que conforma la interfase al usuarie. Dentro de esta ventana o figura pueden existir diversos subplots.

Hasta acá dibujamos gráficos y creamos sus ejes de forma implícita. Esto es bueno para obtener ploteos rápidos. Pero podemos controlar mejor la apariencia de la figura que generamos si definimos todo en forma explícita. Podemos definir la figura, los subplots y los ejes.

Mientras que *subplot* ubica a sus plots en posiciones espaciadas regularmente (una grilla) uno puede ubicar los *ejes* libremente en la figura. Ambas cosas pueden ser útiles, depende de qué estés buscando.

Aunque trabajamos con figuras y subplots sin llamarlos explicitamente, es bueno saber que al invocar plot() matplotlib llama a gca() (get current axes) para obtener acceso a los ejes, y gca() a su vez llama a gcf() (get current figure) para obtener acceso a la figura. Si no existe tal figura, llama a figure() para crearla o más estrictamente hablando, para crear un un único subplot (el número 1 en una grilla de 1x1). Aunque no pidamos explícitamente crear una figura, ésta es creada cuando la necesitamos. Veamos un poco los detalles.

**Figuras**

Una "figura" es la ventana en la interfase al usuarie que lleva como título "Figura #". Las figuras se enumeran comenzando en 1. Varios parámetros determinan la pinta que tiene una figura:

| **Argumento** | **Por Omisión** | **Descripción** |
| --- | --- | --- |
| num | 1 | número de figura |
| figsize | figure.figsize | tamaño de figura en pulgadas (ancho, alto) |
| dpi | figure.dpi | resolución en puntos por pulgada |
| facecolor | figure.facecolor | color del fondo |
| edgecolor | figure.edgecolor | color del borde rodeando el fondo |
| frameon | True | dibujar un recuadro para la figura ? |

Si estás trabajando en una interfaz gráfica podés cerrar una figura clickeando en la X de la ventana. Tambien podés cerrar una ventana desde tu programa llamando al método close(). Dependiendo del parámetro que le pases va a cerrar la figura con que estás trabajando (sin argumentos), una figura específica (como argumento le pasás el número de figura) o todas las figuras (el argumento es "all").

plt.close(1) # Cierra la figura 1

A pesar de que en casi todo el mundo usamos el sistema métrico, increíblemente el imperialismo llega al punto que no hay un modo directo de especificar distancias o tamaños en centímetros en matplotlib. Podemos usar una función auxiliar como ésta para convertir una distancia de *cm* a *pulgadas*:

def cm2inch(value):

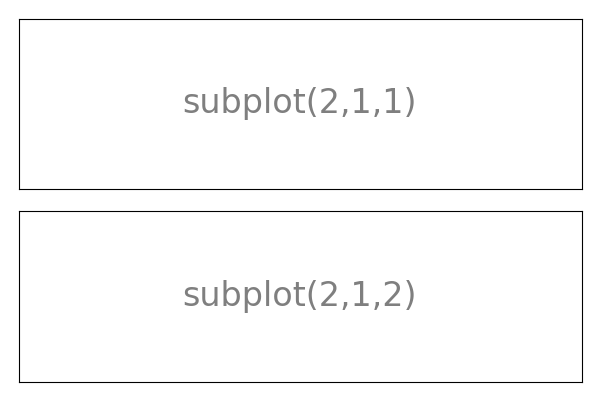
return value/2.54

fig = plt.figure(figsize=(cm2inch(12.8), cm2inch(9.6)))

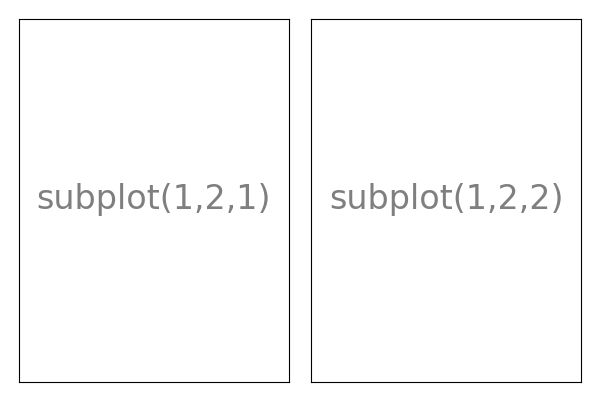
**Subplots**

Podés disponer tus plots en una grilla de intervalos regulares si usás subplots. Sólo tenés que especificar el número de filas, el de columnas y finalmente el número de subplot para activar el subplot correspondiente.

Ejemplo:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_subplot-horizontal_001.png)

Ejemplo:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_subplot-vertical_001.png)

Ejemplo:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_subplot-grid_001.png)

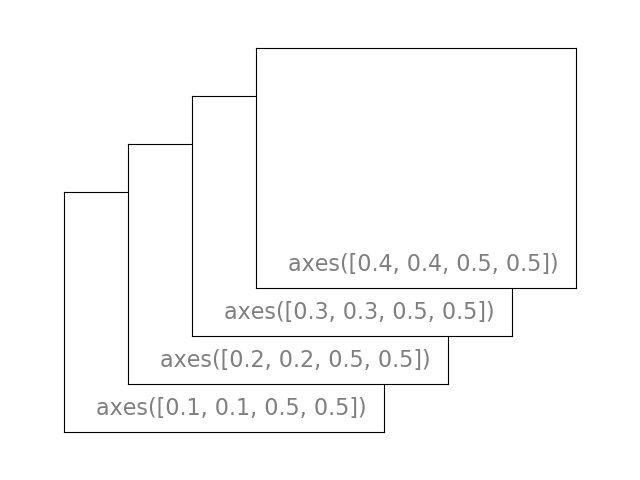
**Ejes**

Podés usar los ejes para ubicar los plots en cualquier lugar de la figura. Si queremos poner un pequeño gráfico como inserto en uno más grande, lo podemos hacer moviendo sus ejes.

Ejemplo:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_axes_001.png)

Ejemplo:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_axes-2_001.png)

**Ejercicios:**

Solo te pedimos que entregues el segundo ejercicio, los otros son optativos.

**Ejercicio 7.9: Subplots fuera de una grilla**

Modificá el siguiente código para reproducir el gráfico que se muestra. Prestá atención a cómo se numeran los subplots.

import matplotlib.pyplot as plt

fig = plt.figure()

plt.subplot(2, 1, 1) # define la figura de arriba

plt.plot([0,1,2],[0,1,0]) # dibuja la curva

plt.xticks([]), plt.yticks([]) # saca las marcas

plt.subplot(2, 2, 3) # define la primera de abajo, que sería la tercera si fuera una grilla regular de 2x2

plt.plot([0,1],[0,1])

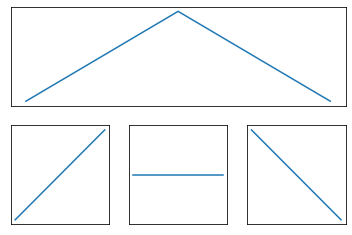
plt.xticks([]), plt.yticks([])

plt.subplot(2, 2, 4) # define la segunda de abajo, que sería la cuarta figura si fuera una grilla regular de 2x2

plt.plot([0,1],[1,0])

plt.xticks([]), plt.yticks([])

plt.show()

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/ex_subplots.png)

**Ejercicio 7.10: Caminatas al azar**

Una [caminata al azar](https://es.wikipedia.org/wiki/Camino_aleatorio) o *random walk* es una formalización matemática de la trayectoria que resulta de hacer sucesivos pasos aleatorios. Por ejemplo, la ruta trazada por una molécula mientras viaja por un líquido o un gas, el camino que sigue un animal en su búsqueda de comida, el precio de una acción fluctuante y la situación financiera de un jugador pueden tratarse, bajo ciertas hipótesis, como una caminata aleatoria.

El siguiente código genera una caminata al azar de N pasos de largo y la grafica.

import numpy as np

import matplotlib.pyplot as plt

def randomwalk(largo):

pasos=np.random.randint (-1,2,largo)

return pasos.cumsum()

N = 100000

plt.plot(randomwalk(N))

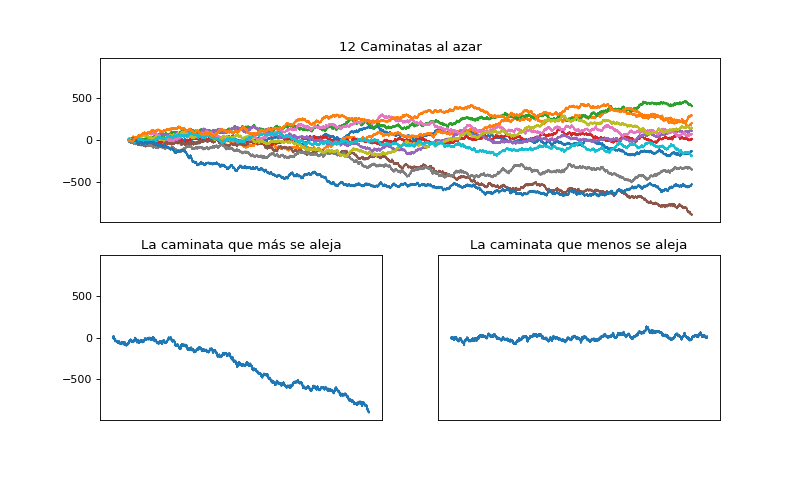
plt.show()

En este ejercicio te pedimos:

1. Modificá el código anterior para ponerles nombres a los ejes ("tiempo" y distancia al origen") y al gráfico.
2. Graficá 12 trayectorias en la misma figura, con diferentes colores.
3. Usá la estructura de subplots sugerida en el [Ejercicio 7.9](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/07_Matplotlib.md#ejercicio-79-subplots-fuera-de-una-grilla) para graficar tres pubplots en una figura:
   * Arriba, grande, 12 trayectorias aleatorias como en el inciso anterior
   * Abajo a la izquierda la trayectoria que más se aleja del origen.
   * Abajo a la derecha la trayectoria que menos se aleja del origen.

Ojo, cuando decimos la que más o menos se aleja, nos referimos a *en algún momento*, no necesariamente a la que termina más cerca o más lejos.

Guardá tu solución del inciso 3 en el archivo random\_walk.py. Debería verse aproximadamente como este plot:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/RW.png)

**Optativos:**

Los siguientes ejercicios profundizan en algunos estilos particulares y son optativos. Si querés ver las soluciones exactas a algunos de estos ejercicios y otros problemas más, podés consultar [acá](https://scipy-lectures.org/intro/matplotlib/index.html#other-types-of-plots-examples-and-exercises).

**Ejercicio 7.11: Gráficos de barras**

Modificá el siguiente código para generar un gráfico similar al que se muestra: tenés que agregar etiquetas para las barras rojas cuidando la alineación del texto.

n = 12

X = np.arange(n)

Y1 = (1 - X / float(n)) \* np.random.uniform(0.5, 1.0, n)

Y2 = (1 - X / float(n)) \* np.random.uniform(0.5, 1.0, n)

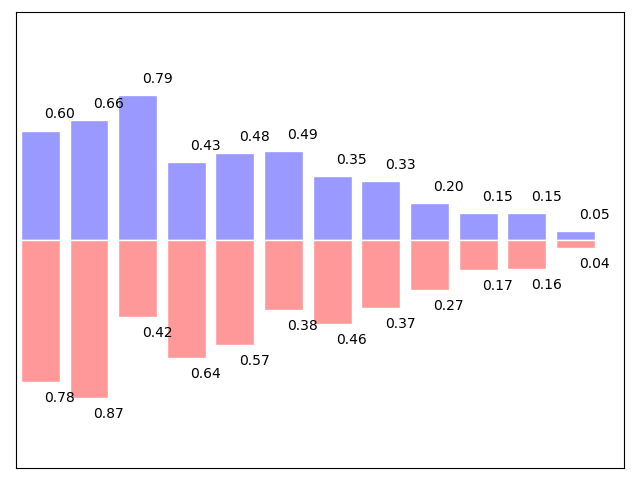
plt.bar(X, +Y1, facecolor='#9999ff', edgecolor='white')

plt.bar(X, -Y2, facecolor='#ff9999', edgecolor='white')

for x, y in zip(X, Y1):

plt.text(x + 0.4, y + 0.05, '%.2f' % y, ha='right', va='bottom')

plt.ylim(-1.25, +1.25)

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_bar_001.png)

**Ejercicio 7.12: Coordenadas polares**

A partir de este código, generá un gráfico como el siguiente.

plt.axes([0, 0, 1, 1])

N = 20

theta = np.arange(0., 2 \* np.pi, 2 \* np.pi / N)

radii = 10 \* np.random.rand(N)

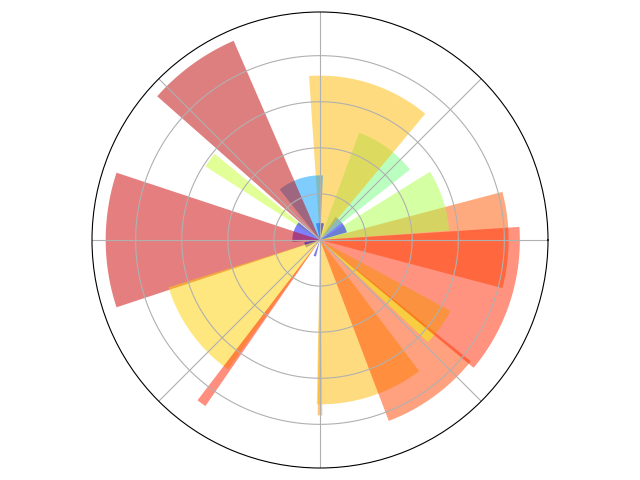
width = np.pi / 4 \* np.random.rand(N)

bars = plt.bar(theta, radii, width=width, bottom=0.0)

for r, bar in zip(radii, bars):

bar.set\_facecolor(plt.cm.jet(r / 10.))

bar.set\_alpha(0.5)

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_polar_001.png)

*Pista*: sólo necesitás modifcar los ejes en la primera línea. Fijate que hay un parámetro polar que tiene por omisión valor False.

**Ejercicio 7.13: Setear el color de un scatter plot**

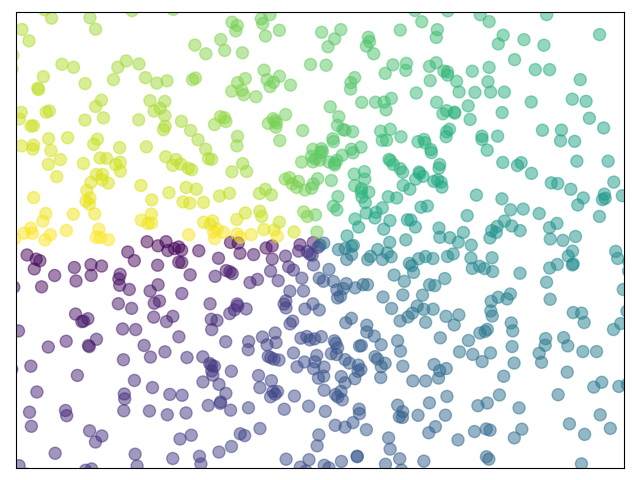
Modificá el código que sigue para generar un gráfico similar al que se muestra, prestando atención a los límites, el tamaño de las marcas, el color, y la transparencia de los trazos.

n = 1024

X = np.random.normal(0,1,n)

Y = np.random.normal(0,1,n)

plt.scatter(X,Y)

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/sphx_glr_plot_scatter_001.png)

*Pista*: El color depende del ángulo que forma el vector (X,Y) con los ejes. Hay diversas formas de calcularlo.

**7.8 Cierre de la clase**

En esta clase vimos cómo se hace una administración eficiente de errores, cómo atrapar excepciones, cómo lanzarlas, y cuándo conviene hacer o no hacer estas cosas.

Vimos que un archivo .py correctamente escrito puede usarse como un módulo, como un programa en sí mismo, o como ambas cosas dependiendo del caso, y mostramos que aunque uno esté escribiendo una pequeña función para solucionar un pequeño problema, es bueno pensar en grande y no imponer restricciones innecesarias.

Aprendimos a documentar y comentar de manera útil, y mostramos el paradigma de contratos. Además vimos algo sobre estilo código estándard.

También estudiamos diversos estilos de gráficos, como obtener un vistazo rápido de los datos y como ajustar cada elemento para obtener un gráfico que pueda ser publicado.

Para cerrar esta clase te pedimos dos cosas:

* Que recopiles las soluciones de los siguientes ejercicios:

1. El archivo fileparse.py del [Ejercicio 7.4](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/04_Flexibilidad.md#ejercicio-74-de-archivos-a-objetos-cual-archivos).
2. El archivo informe.py del [Ejercicio 7.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/04_Flexibilidad.md#ejercicio-75-arreglemos-las-funciones-existentes).
3. El archivo documentacion.py del [Ejercicio 7.8](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/05_Especificacion_y_Documentacion.md#ejercicio-78-funciones-y-documentaci%C3%B3n).
4. El archivo random\_walk.py del [Ejercicio 7.10](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/07_Matplotlib.md#ejercicio-710-caminatas-al-azar).

* Que completes [este formulario](https://docs.google.com/forms/d/1LmeHVZqiEznUUD7FjhTbi6eRQFzvb8Er4dOF3mOy9oo) usando como identificación tu dirección de mail. Al terminar vas a obtener un link para enviarnos tus ejercicios y podrás participar de la revisión de pares.

¡Gracias!