

1. Introducción a Python	20
1.1 Python	20
¿Qué es Python?	20
¿Dónde conseguir Python?	20
¿Para qué fue creado Python?	20
¿Cómo ejecuto Python en mi máquina?	21
Ejercicios	21
Ejercicio 1.1: Python como calculadora	21
Ejercicio 1.2: Obtener ayuda	22
Ejercicio 1.3: Copy-paste	22
1.2 Un primer programa	23
Ejecutando Python	23
Modo interactivo	24
Crear programas	25
Ejecutar programas	25
Un ejemplo de programa	26
Comandos	27
Comentarios	27
Variables	27
Tipos	27
Python distingue mayúsculas y minúsculas	28
Ciclos	28
Indentación	28
Indentando adecuadamente	29
Condicionales	29
Imprimir en pantalla	30
Ingreso de valores por teclado	30
El comando pass	31
Ejercicios	31
Ejercicio 1.4: Debuguear	31
Ejercicio 1.5: La pelota que rebota	32
Ejercicio 1.6: Saludos	33
1.3 Números	33
Tipos de números	33
Booleanos (bool)	33
Enteros (int)	34
Punto flotante (float)	36
Comparaciones	37
Conversión de números	37
Ejercicios	38

Ejercicio 1.7: La hipoteca de David	38
Ejercicio 1.8: Adelantos	38
Ejercicio 1.9: Calculadora de adelantos	39
Ejercicio 1.10: Tablas	39
Ejercicio 1.11: Bonus	40
Ejercicio 1.12: Un misterio	40
Ejercicio 1.13: El volúmen de una esfera	40
1.4 Cadenas	40
Representación de textos	41
Código de escape	41
Representación en memoria de las cadenas	42
Indexación de cadenas	42
Operaciones con cadenas	42
Métodos de las cadenas	43
Mutabilidad de cadenas	44
Conversión de cadenas	44
f-Strings	44
Ejercicios	44
Ejercicio 1.14: Extraer caracteres individuales y subcadenas	45
Ejercicio 1.15: Concatenación de cadenas	46
Ejercicio 1.16: Testeo de pertenencia (test de subcadena)	46
Ejercicio 1.17: Iteración sobre cadenas	47
Ejercicio 1.18: Geringoso rústico	47
Ejercicio 1.19: Métodos de cadenas	47
Ejercicio 1.20: f-strings	48
Ejercicio 1.21: Expresiones regulares	48
Commentario	49
1.5 Listas	50
Creación de Listas	50
Operaciones con listas	50
Iteradores de listas y búsqueda	51
Borrar elementos	52
Ordenar una lista	52
Listas y matemática	52
Ejercicios	53
Ejercicio 1.22: Extracción y reasignación de elementos.	53
Ejercicio 1.23: Ciclos sobre listas	54
Ejercicio 1.24: Test de pertenencia	54
Ejercicio 1.25: Adjuntar, insertar y borrar elementos	55
Ejercicio 1.26: Sorting	56
Ejercicio 1.27: Juntar múltiples cadenas	56

Ejercicio 1.28: Listas de cualquier cosa	57
Ejercicio 1.29: Traductor (rústico) al lenguaje inclusivo	58
1.6 Cierre de la primer clase	58
2. Trabajando con datos	60
2.1 Manejo de archivos	60
Archivos de entrada y salida	60
Comandos usuales para leer un archivo	61
Comandos usuales para escribir un archivo	62
Ejercicios	62
Ejercicio 2.1: Preliminares sobre lectura de archivos	62
Ejercicio 2.2: Lectura de un archivo de datos	64
Ejercicio 2.3: Archivos comprimidos	65
Comentario: ¿No deberíamos estar usando Pandas para esto?	65
2.2 Funciones	65
Funciones a medida	66
Funciones de la biblioteca	66
Errores y excepciones	66
Atrapar y administrar excepciones	67
Generar excepciones	68
Ejercicios	68
Ejercicio 2.4: Definir una función	69
Ejercicio 2.5: Transformar un script en una función	69
Ejercicio 2.6: Administración de errores	70
Ejercicio 2.7: Funciones de la biblioteca	71
Ejercicio 2.8: Ejecución desde la línea de comandos con parámetros	71
2.3 Tipos y estructuras de datos	73
Tipos de datos primitivos	73
Tipo None	73
Estructuras de datos	73
Tuplas	74
Empaquetar tuplas	74
Desempaquetar tuplas	74
Tuplas vs. Listas	75
Diccionarios	75
Operaciones usuales	75
¿Por qué diccionarios?	76
Ejercicios	76
Ejercicio 2.9: Tuplas	76
Ejercicio 2.10: Diccionarios como estructuras de datos	78
Ejercicio 2.11: Más operaciones con diccionarios	79

2.4 Contenedores	80
Panorama	81
Listas como contenedores	81
Construcción de una lista	81
Diccionarios como contenedores	82
Construcción de diccionarios	82
Búsquedas en un diccionario	83
Claves compuestas	83
Conjuntos	83
Ejercicios	84
Ejercicio 2.12: Lista de tuplas	84
Ejercicio 2.13: Lista de diccionarios	86
Ejercicio 2.14: Diccionarios como contenedores	87
Ejercicio 2.15: Balances	89
2.5 Secuencias	89
Tipo de secuencias	89
Rebanadas (slicing)	90
Reasigación de rebanadas	91
Reducciones de secuencias	91
Iterar sobre una secuencia	91
El comando break	92
El comando continue	92
Ciclos sobre enteros	92
La función enumerate()	93
Tuplas y ciclos for	94
La función zip()	94
Ejercicios	94
Ejercicio 2.16: Contar	95
Ejercicio 2.17: Más operaciones con secuencias	95
Ejercicio 2.18: Un ejemplo práctico de enumerate()	96
Ejercicio 2.19: La función zip()	97
Ejercicio 2.20: Invertir un diccionario	99
2.6 Contadores del módulo collections	100
Ejemplo: Contar cosas	100
Contadores	101
Ejercicios	101
Ejercicio 2.21: Contadores	101
2.7 Arbolado porteño	102
Ejercicios	103
Descripción de la base	103

Ejercicio 2.22: Lectura de los árboles de un parque	105
Ejercicio 2.23: Determinar las especies en un parque	105
Ejercicio 2.24: Contar ejemplares por especie	106
Ejercicio 2.25: Alturas de una especie en una lista	106
Ejercicio 2.26: Inclinación promedio por especie de una lista	107
Ejercicio 2.27: Especie con el ejemplar más inclinado	107
Ejercicio 2.28: Especie con más inclinada en promedio	107
2.8 Impresión con formato	108
Formato de cadenas	108
Códigos de formato	109
Formato a diccionarios	109
El método format()	109
Formato estilo C	110
Ejercicios	110
Ejercicio 2.29: Formato de números	110
Ejercicio 2.30: Recolectar datos	111
Ejercicio 2.31: Imprimir una tabla con formato	112
Ejercicio 2.32: Agregar encabezados	113
Ejercicio 2.33: Un desafío de formato	114
Ejercicio 2.34: Tablas de multiplicar	114
2.9 Cierre de la segunda clase	115
3. Algoritmos sobre listas y comprensión de listas	115
3.1 Entorno de desarrollo integrado	116
3.2 Errores	117
Tres tipos de errores:	117
Debuggear a mano	118
¿Qué dice un traceback?	118
Usá el modo REPL de Python	119
Debuggear con print	119
Debuggear con lápiz y papel	120
Ejercicios:	120
Ejercicio 3.1: Semántica	121
Ejercicio 3.2: Sintaxis	122
Ejercicio 3.3: Tipos	122
Ejercicio 3.4: Alcances	122
Ejercicio 3.5: Pisando memoria	123
3.3 Listas y búsqueda lineal	123
Búsqueda lineal	124
El problema de la búsqueda	124

¿Cuántas comparaciones hace este programa?	125
Búsqueda lineal	125
¿Cuántas comparaciones hace este programa?	127
Ejercicios	127
Ejercicio 3.6: Búsquedas de un elemento	127
Ejercicio 3.7: Búsqueda de máximo y mínimo	128
Ejercitación con iteradores y listas	129
Ejercicio 3.8: Invertir una lista	129
Ejercicio 3.9: Propagación	129
3.4 Comprensión de listas	130
Crear listas nuevas	130
Filtros	130
Casos de uso	131
Sintaxis general	131
Digresión histórica	131
Ejercicios	132
Ejercicio 3.10: Comprensión de listas	132
Ejercicio 3.11: Reducción de secuencias	132
Ejercicio 3.12: Consultas de datos	133
Ejercicio 3.13: Extracción de datos	134
Ejercicio 3.14: Extraer datos de una arhcivo CSV	135
Comentario	136
3.5 Objetos	136
Asignaciones	136
Ejemplo de asignación	137
Reasignar valores	138
Peligros	138
Identidad y referencias	138
Copias superficiales	139
Copias profundas	140
Nombre, valores y tipos	140
Verificación de tipos	141
Todo es un objeto	141
Ejercicios	142
Ejercicio 3.15: Datos de primera clase	142
Ejercicio 3.16: Diccionarios	144
Ejercicio 3.17: Fijando ideas	145
3.6 Arbolado porteño y comprensión de listas	146
Ejercicios	146
Ejercicio 3.18: Lectura de todos los árboles	146
Ejercicio 3.19: Lista de altos de Jacarandá	146

Ejercicio 3.20: Lista de altos y diámetros de Jacarandá	147
Ejercicio 3.21: Diccionario con medidas	147
3.7 Cierre de la tercera clase	148
4. Aleatoriedad	148
4.1 Debuggear programas	149
Testear es genial, debuggear es horrible.	149
Aseveraciones (assert)	150
Programación por contratos	151
El debugger de Python (pdb)	151
Ejercicios	156
Ejercicio 4.1: Debugger	156
Ejercicio 4.2: Más debugger	156
Análisis de alternativas para propagar	157
Ejercicio 4.3: Propagar por vecinos	157
Ejercicio 4.4: Propagar por como el auto fantástico	158
Ejercicio 4.5: Propagar con cadenas	159
4.2 Random	160
Valores discretos	160
Ejercicios:	161
Ejercicio 4.6: Generala servida	161
Ejercicio 4.7: Generala no necesariamente servida	162
Semillas	162
Elecciones con reposición	162
Elecciones sin reposición	163
Ejercicio 4.8: Envido	164
Mezclar	164
Valores continuos	164
Ejercicio 4.9: Calcular pi	165
Ejercicio 4.10: Gaussiana	166
4.3 NumPy	168
Instalar e importar numpy	168
¿Cuál es la diferencia entre listas y arreglos?	169
Arreglos n-dimensionales	169
Más información sobre arreglos	170
Crear un arreglo básico	171
Ejercicio 4.11: arange() y linspace()	172
Agregar, borrar y ordenar elementos	172
Conocer el tamaño, dimensiones y forma de un arreglo	173
Cambiar la forma de un arreglo	174
Agregar un nuevo eje a un arreglo	175

Índices y rebanadas	175
Crear arreglos usando datos existentes	178
Operaciones básicas sobre arreglos	179
Broadcasting	181
Operaciones un poco más complejas	181
Crear matrices	182
Fórmulas matemáticas	186
Guardar y cargar objetos de numpy	187
Ejercicio 4.12: Guardar temperaturas	188
Ejercicio 4.13: Empezando a plotear	189
4.4 El album de Figuritas	189
Las figuritas del mundial	189
Datos:	190
Herramientas útiles de Python	190
El modelo del álbum de figuritas	190
Primera simplificación	191
Ejercicios con figus sueltas	191
Ejercicio 4.14: Crear	191
Ejercicio 4.15: Incompleto	191
Ejercicio 4.16: Comprar	192
Ejercicio 4.17: Cantidad de compras	192
Ejercicio 4.18:	192
Ejercicio 4.19:	192
Ahora con paquetes	193
Ejercicios con paquetes	193
Ejercicio 4.20:	193
Ejercicio 4.21:	193
Ejercicio 4.22:	193
Ejercicio 4.23:	193
Gráficar el llenado del álbum	193
Ejercicios un toque más estadísticos:	194
Ejercicio 4.24:	194
Ejercicio 4.25: Plotear el histograma	195
Ejercicio 4.26:	195
Ejercicio 4.27:	195
Ejercicio 4.28: Cooperar vs competir	195
4.5 Gráficos del Arbolado porteño	195
Ploteando datos reales	195
Ejercicio 4.29: Histograma de altos de Jacarandás	195
Ejercicio 4.30: Scatterplot (diámetro vs alto) de Jacarandás	196
Ejercicio 4.31: Scatterplot para diferentes especies	197

4.6 Cierre de la cuarta clase	198
5. Complejidad y Organización de programas	198
5.1 Scripting	199
¿Qué es un script?	199
Un problema	199
Definir nombres	199
Definir funciones	200
¿Qué es una función?	200
Dónde definir funciones	201
El estilo Bottom-Up	201
Diseño de funciones	202
Doc-strings	202
Notas sobre el tipo de datos	202
Ejercicios	203
Ejercicio 5.1: Estructurar un programa como una colección de funciones	204
Ejercicio 5.2: Crear una función de alto nivel para la ejecución del programa.	204
Comentario	205
5.2 Funciones	205
Llamando a una función	205
Argumentos por omisión	206
Si un argumento es opcional, dale un nombre.	206
Buenas prácticas de diseño	207
Devolver un resultado	207
Devolver múltiples resultados	208
Alcance de variables	208
Variables locales	208
Variables globales	209
Modificar el valor de una variable global	209
Pasaje de argumentos	210
Reasignar versus modificar	210
Ejercicios	211
Ejercicio 5.3: Parsear un archivo CSV	211
Ejercicio 5.4: Selector de Columnas	213
Ejercicio 5.5: Conversión de tipo	215
Ejercicio 5.6: Trabajando sin encabezados	216
Comentario	217
5.3 Módulos	217
Módulos y la instrucción import	217
Namespaces	218

Definiciones globales	218
Módulos como entornos	218
Ejecución de módulos	219
El comando import as	219
from módulo import nombre	219
Notas sobre import	220
Carga de módulos	220
Ejercicios	220
Ejercicio 5.7: Importar módulos	221
Ejercicio 5.8: Usemos tu módulo	222
Ejercicio 5.9: Un poco más allá	223
Comentario	223
5.4 Búsqueda binaria	223
Búsqueda sobre listas ordenadas	223
Ejercicio 5.10: Búsqueda lineal sobre listas ordenadas.	223
Búsqueda binaria	224
¿Cuántas comparaciones hace este programa?	227
Comparación entre ambos métodos	227
Ejercicio 5.11: Búsqueda binaria	228
5.5 Complejidad de algoritmos	228
Resumen de algoritmos de Búsqueda	228
Complejidad de algoritmos	229
Un algoritmo cuadrático	230
Complejidad en el peor caso	231
Estructuras de datos y Tipos Abstractos de Datos	231
Ejercicios:	231
Ejercicio 5.12: Insertar un elemento en una lista	232
Ejercicio 5.13: Cálcular la complejidad de dos resoluciones de propagar	232
Secuencias binarias	232
Ejercicio 5.14: Complejidad de incrementar()	233
Ejercicio 5.15: Un ejemplo más complejo	233
5.6 Gráficos de complejidad	233
Contar la cantidad de operaciones de un algoritmo	233
Ejercicio 5.16: Contar comparaciones en la búsqueda binaria	234
Gráficoar la cantidad de comparaciones promedio	234
Ejercicio 5.17: Búsqueda binaria vs. búsqueda secuencial	237
5.7 Cierre de la quinta clase	238
6. Diseño, especificación, documentación y estilo.	238
6.1 Control de errores	239

Formas en que los programas fallan	239
Excepciones	240
Administración de excepciones	241
Excepciones integradas	242
Valores asociados a excepciones	242
Podés atrapar múltiples excepciones	243
Todas las excepciones	243
Así NO se atrapan excepciones.	244
Así es un poco mejor.	244
Re-lanzar una excepción	244
Buenas prácticas al administrar excepciones	245
La instrucción finally.	245
Ejercicios	245
Lancemos excepciones	245
Atrapemos excepciones	246
Ejercicios:	247
Ejercicio 6.1: Errores silenciados	247
Comentarios	248
6.2 El módulo principal	248
Función principal	248
Módulo principal en Python	248
Chequear <code>__main__</code>	249
Módulo principal vs. módulo importado	249
Modelo de programa	249
Herramientas para la consola	250
Argumentos en la línea de comandos	250
Standard I/O	251
Terminación del programa	251
El comando <code>#!</code>	252
Modelo de script con parámetros	252
Ejercicios	253
Ejercicio 6.2: Función main()	253
Ejercicio 6.3: Hacer un script	254
6.3 Cuestiones de diseño	254
Archivos versus iterables	254
Una idea profunda: "Duck Typing" (Identificación de patos)	255
Buenas prácticas en el diseño de bibliotecas	256
Ejercicio	256
Ejercicio 6.4: De archivos a "objetos cual archivos"	256
Ejercicio 6.5: Arreglemos las funciones existentes	257
6.4 Contratos: Especificación y Documentación	257

Documentación	258
Comentarios vs documentación	258
¿Por qué documentamos?	259
Código autodocumentado	259
Un error común: la sobredocumentación	260
Contratos	261
Precondiciones	262
Poscondiciones	262
El qué, no el cómo	262
Aseveraciones	262
Ejemplos	263
Ejercicio 6.6: Sumas	265
Invariantes de ciclo	265
Ejercicio 6.7: Invariante en sumas	267
Parámetros mutables e inmutables	267
Resumen	268
Ejercicios	268
Ejercicio 6.8: Funciones y documentación	268
6.5 Estilos de codeo	269
PEP 8 - La guía de estilo para Python	270
Indentación	270
Tamaño máximo de línea	270
Líneas en blanco	270
Imports	270
Espacios en blanco en expresiones	271
Convenciones de nombres	272
Estilos de nombres	272
Hay mucho más!	273
Zen de Python	273
6.6 La biblioteca matplotlib	274
pyplot	274
Un plot simple	274
6.7 Cierre de la sexta clase	275
6.7 Cierre de la sexta clase	276
6.7 Cierre de la sexta clase	277
El ploteo estándar	278
Un gráfico básico	279
Detalles de un plot simple	280
Cómo cambiar los colores y ancho de los trazos	280
Límites de los ejes	281

Marcas en los ejes	282
Texto de las marcas en los ejes	283
Movamos el contorno	284
Pongámosle un título	285
Algunos puntos interesantes	286
El diablo está en los detalles	287
Figuras, subplots, ejes y marcas (ticks)	287
Figuras	288
Subplots	290
Ejes	290
Ejercicios:	292
Ejercicio 6.9: Subplots fuera de una grilla	293
Ejercicio 6.10: Caminatas al azar	294
Optativos:	295
Ejercicio 6.11: Gráficos de barras	295
Ejercicio 6.12: Coordenadas polares	296
Ejercicio 6.13: Setear el color de un scatter plot	297
6.7 Cierre de la sexta clase	298
7. Fechas, Carpetas y Pandas	299
7.1 Manejo de fechas y horas	300
El módulo datetime	300
Ejemplo: Obtener fecha y hora actuales	300
Ejemplo: Obtener fecha actual	300
La clase datetime.date	301
Ejemplo: Un objeto para representar una fecha	301
Ejemplo: Obtener la fecha a partir de un timestamp	301
Ejemplo: Obtener el año, el mes y el día por separado.	302
La clase datetime.time	302
Ejemplo: Representar la hora con un objeto time	302
Ejemplo: Obtener horas, minutos, segundos y micro-segundos	303
La clase datetime.datetime	303
Ejemplo: Objeto datetime	303
Ejemplo: Obtener año, mes, día, hora, minutos, timestamp de un datetime	304
La clase datetime.timedelta	304
Ejemplo: Diferencia entre fechas y horarios	304
Ejemplo: Diferencia entre objetos timedelta	305
Ejemplo: Imprimir objetos timedelta negativos	305
Ejemplo: Duración en segundos	305
Formato para fechas y horas	306
Python strftime() - convertir un objeto datetime a string	306
Ejemplo: Formato de fecha usando strftime()	306

Python strftime() - convertir una cadena a un objeto datetime	307
Ejemplo: strftime()	307
Ejercicios:	307
Ejercicio 7.1: Segundos vividos	307
Ejercicio 7.2: Cuánto falta	308
Ejercicio 7.3: Fecha de reincorporación	308
Ejercicio 7.4: Días hábiles	308
7.2 Manejo de archivos y carpetas	309
Manejo de archivos y directorios	309
Obtener el directorio actual	309
Cambiar el directorio de trabajo	309
Listar directorios y archivos	310
Crear un nuevo directorio	311
Renombrar un directorio o un archivo	311
Eliminar un directorio o un archivo	312
Recorriendo directorios con os.walk()	313
Ejemplo	313
Cambiar atributos de un archivo	314
7.3 Ordenar archivos en Python	314
Ejercicio 7.5: Recorrer el árbol de archivos	315
Ejercicio 7.6: Ordenar el árbol de archivos (optativo)	315
7.4 Introducción a Pandas	316
Lectura de datos	316
Selección	318
Filtros booleanos	319
Scatterplots	320
Filtros por índice y por posición	320
Selección de una columna	322
Series temporales en Pandas	322
Caminatas al azar	323
Ejemplo: 12 personas caminando 8 horas	323
Guardando datos	324
Incorporando el Arbolado lineal	324
Ejercicio 7.7: Lectura y selección	324
Ejercicio 7.8: Boxplots	325
Ejemplo de pairplot	325
Ejercicio 7.9: Comparando especies en parques y en veredas	326
7.5 Series temporales	327
Análisis y visualización de series temporales.	328
Lectura de archivos temporales	328

Ondas de marea en el Río de la Plata	329
Vientos y ondas de tormenta en el Río de la Plata	330
Ejercicio 7.10:	333
Parte optativa	334
Análisis por medio de transformadas de Fourier	335
Preparación de módulos y datos	336
Espectro de potencia y de ángulos para San Fernando	338
Espectro de potencia y de ángulos para Buenos Aires	341
Ejercicio 7.11: Desfasajes	343
Un poco más avanzados:	343
Ejercicio 7.12: Otros puertos	343
Ejercicio 7.13: Otros períodos	343
7.6 Cierre de la séptima	344
8. Clases y objetos	344
8.1 Clases	345
Programación orientada a objetos (POO)	345
La instrucción class	346
Instancias	346
Datos de una instancia	347
Métodos de una instancia.	347
Visibilidad en clases (Scoping)	348
Ejercicios	349
Ejercicio 8.1: Objetos como estructura de datos.	349
Ejercicio 8.2: Agregá algunos métodos	350
Ejercicio 8.3: Lista de instancias	351
Ejercicio 8.4: Usá tu clase	351
8.2 Herencia	352
Introducción	352
Extensiones	352
Ejemplo	352
Agregar un método nuevo	353
Redefinir un método existente	353
Utilizar un método prevalente	354
El método <code>__init__</code> y herencia.	354
Usos de herencia	355
Relación "isinstance"	355
La clase base object	356
Herencia múltiple.	356
Ejercicios	356
Ejercicio 8.5: Un problema de extensibilidad	357

Ejercicio 8.6: Usemos herencia para cambiar la salida	359
Ejercicio 8.7: Polimorfismo en acción	362
Ejercicio 8.8: Volvamos a armar todo	363
Discusión	364
8.3 Métodos especiales	365
Introducción	365
Métodos especiales para convertir a strings	365
Métodos matemáticos especiales	367
Métodos especiales para acceder a elementos	367
Invocar métodos	368
Métodos ligados	368
Acceso a atributos	369
Ejercicios	369
Ejercicio 8.9: Mejor salida para objetos	370
Ejercicio 8.10: Ejemplo de getattr()	370
8.4 Objetos, pilas y colas	371
Un ejercicio geométrico	371
Ejercicio 8.11: Canguros buenos y canguros malos	372
Colas	374
Ejercicio 8.12: Torre de Control	375
Pilas	375
Ejercicio 8.13: implementar el TAD pila	377
8.5 Teledetección	378
Ejercicio 8.14: Optativo de teledetección	378
Ejercicios:	380
Ejercicio 8.15: Ver una banda	380
Ejercicio 8.16: Histogramas	380
Ejercicio 8.17: Máscaras binarias	380
Ejercicio 8.18: Clasificación manual	381
Ejercicio 8.19: Clasificación automática	382
8.6 Cierre de la octava clase	383
Cierre de clase	383
9. Generadores e iteradores	384
9.1 El protocolo de iteración	385
Iteraciones por doquier	385
El protocolo de iteración	385
Iterable	386
Ejercicios	387
Ejercicio 9.1: Iteradores, un ejemplo	387

Ejercicio 9.2: Iteración sobre objetos	388
Ejercicio 9.3: Un iterador adecuado	390
9.2 Iteración a medida	391
Un problema de iteración	391
Generadores	392
Ejercicios	393
Ejercicio 9.4: Un generador simple	393
Ejercicio 9.5: Monitoreo de datos en tiempo real.	394
Ejercicio 9.6: Uso de un generador para producir datos	395
Ejercicio 9.7: Cambios de precio de un camión	396
Discusión	397
9.3 Productores, consumidores y cañerías.	397
Sistemas productor-consumidor	397
Pipelines con generadores	398
Ejercicios	399
Ejercicio 9.8: Configuremos un pipeline simple	399
Ejercicio 9.9: Un pipeline más en serio	400
Ejercicio 9.10: Un pipeline más largo	401
Ejercicio 9.11: Filtremos los datos	402
Ejercicio 9.12: El pipeline ensamblado	403
Discusión	403
9.4 Más sobre generadores	403
Expresiones generadoras	404
Motivos para usar generadores	405
El módulo itertools	405
Ejercicios	406
Ejercicio 9.13: Expresiones generadoras	406
Ejercicio 9.14: Expresiones generadoras como argumentos en funciones.	406
Ejercicio 9.15: Código simple	407
Ejercicio 9.16: Volviendo a ordenar imágenes	407
9.5 Predador Presa	407
Clases del modelo inicial:	409
Animales	409
Constructor	409
Consultas	409
Modificadores	410
León	411
Ejemplo: Un León	411
Antílope	412
Ejemplo: Un León y dos Antílopes	412

Ejemplo: Un León y una Leona	413
El tablero	414
Constructor	414
Modificadores	414
Getters	414
Modificadores complejos	415
Consultas	415
Auxiliares	416
El mundo	416
Constructor	417
Modelado de la dinámica	417
Probando el modelo completo	419
Ejercicio 9.17: Etapa de reproducción	419
Ejercicio 9.18: Acotando la reproducción	419
Ejercicio 9.19: Alcanzando la madurez	419
Extensiones del modelo	419
Exploraciones	420
9.6 Cierre de la novena clase	420
Cierre de clase	420
10. Recursión y regresión	421
10.1 Intro a la Recursión	422
La recursión y cómo puede ser que funcione	422
Una función recursiva matemática	423
Ejercicio 10.1:	424
Algoritmos recursivos y algoritmos iterativos	424
Un ejemplo de recursión elegante	425
Un ejemplo de recursión poco eficiente	427
10.2 Diseño de algoritmos recursivos	430
Un primer diseño recursivo	430
Recursión de cola	432
Modificación de la firma	433
Limitaciones	435
Sabías que	436
Resumen	436
10.3 Práctica de Recursión	437
Ejercicios	437
Ejercicio 10.2: Números triangulares	437
Ejercicio 10.3: Dígitos	437
Ejercicio 10.4: Potencias	437

Ejercicio 10.5: Subcadenas	438
Ejercicio 10.6: Paridad	438
Ejercicio 10.7: Máximo	438
Ejercicio 10.8: Replicar	438
Ejercicio 10.9: Pascal	438
Ejercicio 10.10: Combinatorios	439
Ejercicio 10.11: Búsqueda binaria	439
Ejercicio 10.12: Envolviendo a Fibonacci	440
Ejercicio 10.13: Hojas ISO y recursión	441
10.4 Regresión lineal	441
Regresión lineal simple	441
Criterio de cuadrados mínimos	443
Ejemplo: el modelo de cuadrados mínimos	444
Ajuste del modelo de cuadrados mínimos	445
Ejemplo: datos sintéticos	445
Ejercicio 10.14: precio_alquiler ~ superficie	447
Ejemplo: relación cuadrática	447
Parte optativa:	449
Ejemplo: precómputo de atributos adecuados	449
Scikit-Learn	450
Regresión Lineal Múltiple	451
Ejemplo: superficie y antigüedad	452
Ejercicio 10.15: Peso específico	452
Ejercicio 10.16: Modelo cuadrático	453
Navaja de Ockham	454
Ejercicio 10.17: Modelos polinomiales para una relación cuadrática	454
Ejercicio 10.18: selección de modelos	455
Ejercicio 10.19: Datos para la evaluación	456
Ejercicio 10.20: Altura y diámetro de árboles.	457
Ejercicio 10.21: Gráficos de ajuste lineal con Seaborn	458
10.5 Cierre de la clase de Recursión y Regresión	458
Cierre de la clase	459
11. Ordenamiento	459
11.1 Ordenamientos sencillos de listas	460
Ordenamiento por selección	460
Invariante en el ordenamiento por selección	463
¿Cuánto cuesta ordenar por selección?	464
Ordenamiento por inserción	464
Invariante del ordenamiento por inserción	467
¿Cuánto cuesta ordenar por inserción?	467

Inserción en una lista ordenada	467
Resumen	468
Ejercicios	468
Ejercicio 11.1:	468
Ejercicio 11.2: burbujeo	468
Ejercicio 11.3: ordernar a mano	469
Ejercicio 11.4: experimento con 3 métodos	470
Ejercicio 11.5: comparar métodos gráficamente	470
11.2 Divide y reinarás	471
El algoritmo merge sort (u ordenamiento por mezcla)	471
Ejemplo: Árbol de recursión	474
¿Cuánto cuesta el Merge sort?	475
Resumen	476
Ejercicios:	477
Ejercicio 11.6:	477
Ejercicio 11.7:	477
El módulo timeit	477
Ejemplo: evaluar el método de selección con timeit.	479
Ejercicio 11.8:	480
Ejercicio 11.9:	481
11.3 Algoritmos de clasificación supervisada	481
Veamos los datos	482
Visualización de los datos	484
Ejercicio 11.10: Seaborn	485
Training y testing	485
Modelar	486
Evaluación del modelo	488
Pasando en limpio todo	489
Ejercicios:	489
Ejercicio 11.11:	489
Ejercicio 11.12:	490
11.4 Cierre de la clase de Ordenamiento	490
Cierre clase	490

1. Introducción a Python

El objetivo de este primera clase es introducir algunos conceptos básicos de Python. Comenzando desde cero vas a aprender a editar, a ejecutar y a debuguear pequeños programas. Vas a aprender sobre diferentes tipos de datos en Python: número enteros, números de punto flotante, cadenas y listas.

- [1.1 Python](#)
- [1.2 Un primer programa](#)
- [1.3 Números](#)
- [1.4 Cadenas](#)
- [1.5 Listas](#)
- [1.6 Cierre de la primer clase](#)

1.1 Python

¿Qué es Python?

Python es un lenguaje interpretado de alto nivel. Frecuentemente se lo clasifica como lenguaje de "scripting". La sintaxis del Python tiene elementos de lenguaje C de programación.

Python fue creado por Guido van Rossum a principios de la década del '90 y lo nombró así en honor de Monty Python.

¿Dónde conseguir Python?

Te recomendamos instalar Python 3.6 o más nuevo. En la documentación previa hablamos sobre [cómo instalar Python para este curso](#).

¿Para qué fue creado Python?

El objetivo original de su autor fue crear un lenguaje de programación con el que pudiera realizar las tareas de administración de un sistema fácilmente. En algún sentido los scripts de la terminal no eran suficientemente poderosos y programar esas tareas en C resultaba demasiado tedioso. Python fue creado para llenar ese hueco en el medio.

¿Cómo ejecuto Python en mi máquina?

Existen diferentes entornos en los que podés correr Python en tu computadora. Es importante saber que Python está instalado normalmente como un programa que se ejecuta desde la consola. Desde la terminal deberías poder ejecutar `python` así:

```
bash $ python
Python 3.8.1 (default, Feb 20 2020, 09:29:22)
[Clang 10.0.0 (clang-1000.10.44.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world")
hello world
>>>
```

Si es la primera vez que ves una consola o terminal, sería conveniente que pares aquí, leas [un tutorial corto](#) sobre cómo usar la consola de tu sistema operativo y luego vuelvas para seguir leyendo.

Existen diversos entornos fuera de la terminal en los que se puede escribir y ejecutar código Python. Pero para nosotros es importante que primero aprendas a usarlo desde la terminal: si lo sabés usar bien desde la terminal (que es su entorno natural) lo podrás usar en cualquier otro entorno. Ya en la próxima clase usarás Python dentro de un entorno de desarrollo. Por ahora, te recomendamos usarlo de esta manera que acabamos de explicar.

Ejercicios

Ejercicio 1.1: Python como calculadora

En tu máquina, iniciá Python y usalo como calculadora para resolver el siguiente problema:

- ¿Cuántas horas son 105 minutos?
- ¿Cuántos kilómetros son 20 millas? (un kilómetro corresponde a 0,6214 millas)

```
>>> 105/60
1.75
>>> 20 / 0.6214
32.1853878339234
```

tip: Usá el guión bajo (underscore, `_`) para referirte al resultado del último cálculo.

- Si alguien corre una carrera de 20 millas en 105 minutos, ¿cuál fue tu velocidad promedio en km/h?

```
>>> _/1.75  
18.391650190813372
```

Ejercicio 1.2: Obtener ayuda

Ver [Ejercicio 1.1](#) y [Ejercicio 1.2](#) y [Sección 1.1](#)

Usá el comando `help()` para obtener ayuda sobre la función `abs()`. Luego, usá el `help()` para obtener la ayuda sobre la función `round()`. Tipeá `help()` sólo para entrar en la ayuda en modo interactivo.

El `help()` no funciona con los comandos básicos de Python como `for`, `if`, `while`, etc. Si tipeás `help(for)` vas a obtener un error. Podés probar usando comillas como en `help("for")`, en algunos entornos funciona bien. Si no, siempre podés hacer una búsqueda en internet.

La documentación oficial en inglés de Python se encuentra en <http://docs.python.org>. Por ejemplo, encontrá ahí la documentación sobre la función `abs()` (ayuda: está dentro de "library reference" y relacionado a las "built-in functions").

Ejercicio 1.3: Copy-paste

Este curso está estructurado como una serie de páginas web tradicionales en las que los incentivamos a probar interactivamente fragmentos de código en sus intérpretes de Python escribiéndolos a mano. Si estás aprendiendo Python por primera vez, esta forma "lenta" de hacer las cosas es la que recomendamos. Vas a entender mejor yendo lento y escribiendo los comandos vos mismo mientras pensás en lo que estás tipeando.

Es importante que tipées los comandos a mano. Para usar copy-paste quizás mejor ni hacerlos. Parte del objetivo de los ejercicios es entrenar tus manos, tus ojos y tu cabeza en leer, escribir y mirar código tal como dice [Zed Shaw en su libro](#). Usar copy-paste excesivamente es como hacerte trampa a vos mismo. Es como tratar de aprender a tocar la guitarra escuchando discos: es probable que no aprendas nunca.

Si en algún momento necesitás hacer "copy-paste" de fragmentos de código, seleccioná el código que viene luego del símbolo `>>>` y hasta la siguiente linea en blanco o el siguiente `>>>` (el que aparezca primero). Seleccioná "copy" en el navegador (Ctrl-C), andá al intérprete de Python y poné "paste" (Ctrl-V o Ctrl-shift-V)

para pegarlo. Para ejecutar el código es posible que tengas que apretar "Enter" luego de pegarlo.

Usá copy-paste para ejecutar los siguientes comandos:

```
>>> 12 + 20
32
>>> (3 + 4
      + 5 + 6)
18
>>> for i in range(5):
    print(i)

0
1
2
3
4
>>>
```

Advertencia: No es posible pegar más de un comando de Python (comandos que aparecen luego del símbolo >>>) por vez en el entorno básico de Python.

1.2 Un primer programa

En esta sección vas a crear tu primer programa en Python, ejecutarlo y debuguearlo.

Ejecutando Python

Los programas en Python siempre son ejecutados en un intérprete de Python.

El intérprete es una aplicación que funciona en la consola y se ejecuta desde la terminal.

```
python3
Python 3.6.1 (v3.6.1:69c0db5050, Mar 21 2017, 01:21:04)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Los programadores no suelen tener problemas en usar el intérprete de esta forma, aunque no es la más cómoda para principiantes. Más adelante vamos a porponerles

usar entornos de desarrollo más sofisticados, pero por el momento quedémosnos con la incomodidad que nos va a enseñar cosas útiles.

Modo interactivo

Cuando ejecutás Python, entrás al modo *interactivo* en el que podés experimentar.

Si escribís un comando, se va a ejecutar inmediatamente. No hay ningún ciclo de edición-compilación-ejecución-debug en esto, como en otros lenguajes.

```
>>> print('hello world')
hello world
>>> 37*42
1554
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>>
```

Esta forma de escribir código (en una consola del lenguaje) que se evalúa inmediatamente e imprime el resultado, se denomina *bucle de Lectura-Evaluación-Impresión* (REPL por las siglas en inglés de «Read-Eval-Print-Loop»). Asegurate de poder interactuar con el intérprete antes de seguir.

Veamos en mayor detalle cómo funciona este REPL:

- >>> es el símbolo del intérprete para comenzar un nuevo comando.
- ... es el símbolo del intérprete para continuar con un comando comenzado antes. Dejá una línea en blanco para terminar lo que ya ingresaste.

El símbolo ... puede mostrarse o no dependiendo de tu entorno. En este curso lo mostraremos como líneas en blanco para facilitar el copy-paste de fragmentos de código.

Antes vimos que el guion bajo _ guarda el último resultado.

```
>>> 37 * 42
1554
>>> _ * 2
3108
>>> _ + 50
```

3158

>>>

Esto solo es válido en el modo interactivo que estamos viendo. No uses el guión bajo en un programa.

Crear programas

Los programas se guardan en archivos .py.

```
# hello.py
print('hello world')
```

Podés crear estos archivos con tu editor de texto favorito. Más adelante vamos a proponerles usar el `spyder` que es un entorno de desarrollo integrado (IDE) que permite tener en la pantalla un editor y un intérprete al mismo tiempo, entre otras cosas. Pero por ahora usemos el `block de notas`, o el `gedit` para seguir con estos ejemplos.

Ejecutar programas

Para ejecutar un programa, correlo en la terminal con el comando `python` seguido del nombre del archivo a ejecutar. Por ejemplo, en una línea de comandos Unix (por ejemplo Ubuntu):

```
bash % python hello.py
hello world
bash %
```

O en una terminal de Windows:

```
C:\SomeFolder>hello.py
hello world

C:\SomeFolder>c:\python36\python hello.py
hello world
```

Observación: En Windows puede ser necesario especificar el camino (path) completo al intérprete de Python como en `c:\python36\python`. Sin embargo, si Python está instalado del modo usual, podría alcanzar con que tipées el nombre del programa como en `hello.py`.

Tené en cuenta que con estos comandos estás corriendo el código de Python desde la línea de comandos de tu sistema operativo. El código se ejecuta, Python termina y el control vuelve a la terminal, saliendo de Python. Si necesitás ejecutarlo y seguir dentro del intérprete de Python podés usar `python -i hello.py`.

Si estás dentro del intérprete de Python y querés salir y volver a la línea de comandos, podés hacerlo mediante el comando `exit()`.

Un ejemplo de programa

Resolvamos el siguiente problema:

Una mañana ponés un billete en la vereda al lado del obelisco porteño. A partir de ahí, cada día vas y duplicás la cantidad de billetes, apilándolos prolijamente. ¿Cuánto tiempo pasa antes de que la pila de billetes sea más alta que el obelisco?

Acá va una solución:

```
# obelisco.py
grosor_billete = 0.11 * 0.001 # 0.11 mm escrito en metros
altura_obelisco = 67.5          # altura en metros
num_billetes = 1
dia = 1

while num_billetes * grosor_billete <= altura_obelisco:
    print(dia, num_billetes, num_billetes * grosor_billete)
    dia = dia + 1
    num_billetes = num_billetes * 2

print('Cantidad de días', dia)
print('Cantidad de billetes', num_billetes)
print('Altura final', num_billetes * grosor_billete)
```

Cuando lo ejecutás, la salida será la siguiente:

```
bash % python3 obelisco.py
1 1 0.00011
2 2 0.00022
3 4 0.00044
4 8 0.00088
5 16 0.00176
6 32 0.00352
...
19 262144 28.83584
20 524288 57.67168
Cantidad de días 21
Cantidad de billetes 1048576
Altura final 115.34336
```

A continuación vamos a usar este primer programa como ejemplo para aprender algunas cosas fundamentales sobre Python.

Comandos

Un programa de Python es una secuencia de comandos:

```
a = 3 + 4  
b = a * 2  
print(b)
```

Cada comando se termina con una nueva línea. Los comandos son ejecutados uno luego del otro hasta que el intérprete llega al final del archivo.

Comentarios

Los comentarios son texto que no será ejecutado.

```
a = 3 + 4  
# Esto es un comentario  
b = a * 2  
print(b)
```

Los comentarios comienzan con # y siguen hasta el final de la línea.

Variables

Una variable es un nombre para un valor. Estos nombres pueden estar formados por letras (minúsculas y mayúsculas) de la a a la z. También pueden incluir el guión bajo, y se pueden usar números, salvo como primer carácter.

```
altura = 442 # válido  
_altura = 442 # válido  
altura2 = 442 # válido  
2altura = 442 # inválido
```

Tipos

El tipo de las variables no debe ser declarado como en otros lenguajes. El tipo es asociado con el valor del lado derecho.

```
height = 442           # Entero  
height = 442.0         # Punto flotante
```

```
height = 'Really tall' # Cadena de caracteres
```

Decimos que Python tiene tipado dinámico. El tipo percibido por el intérprete puede cambiar a lo largo de la ejecución dependiendo del valor asignado a la variable.

Python distingue mayúsculas y minúsculas

Mayúsculas y minúsculas son diferentes para Python. Por ejemplo, todas las siguientes variables son diferentes.

```
name = 'David'  
Name = 'Diego'  
NAME = 'Rosita'
```

Los comandos de Python siempre se escriben con minúsculas.

```
while x < 0:    # OK  
WHILE x < 0:    # ERROR
```

Ciclos

El comando `while` ejecuta un ciclo o *loop*.

```
while num_billetes * grosor_billete <= altura_obelisco:  
    print(dia, num_billetes, num_billetes * grosor_billete)  
    dia = dia + 1  
    num_billetes = num_billetes * 2  
  
print('Cantidad de días', dia)
```

Los comandos indentados debajo del `while` se van a ejecutar mientras que la expresión luego del `while` sea verdadera (`true`).

Indentación

La indentación se usa para marcar grupos de comandos que van juntos. Considerá el ejemplo anterior:

```
while num_billetes * grosor_billete <= altura_obelisco:  
    print(dia, num_billetes, num_billetes * grosor_billete)  
    dia = dia + 1  
    num_billetes = num_billetes * 2  
  
print('Cantidad de días', dia)
```

La indentación agrupa los comandos siguientes como las operaciones a repetir:

```
print(dia, num_billetes, num_billetes * grosor_billete)
dia = dia + 1
num_billetes = num_billetes * 2
```

Como el comando `print()` del final no está indentado, no pertenece al ciclo. La línea en blanco que dejamos entre ambos solo está para facilitar la lectura y no afecta la ejecución.

Indentando adecuadamente

Algunas recomendaciones sobre cómo indentar:

- Usá espacios y no el tabulador.
- Usá 4 espacios por cada nivel.
- Usá un editor de textos que entienda que estás escribiendo en Python.

El único requisito del intérprete de Python es que la indentación dentro de un mismo bloque sea consistente. Por ejemplo, esto es un error:

```
while num_billetes * grosor_billete <= altura_obelisco:
    print(dia, num_billetes, num_billetes * grosor_billete)
        dia = dia + 1 # ERROR
    num_billetes = num_billetes * 2
```

Condicionales

El comando `if` es usado para ejecutar un condicional:

```
if a > b:
    print('Gana a')
else:
    print('Gana b')
```

Podés verificar múltiples condiciones agregando condiciones extras con `elif`.

```
if a > b:
    print('Gana a')
elif a == b:
    print('Empate!')
else:
    print('Gana b')
```

El comando `elif` viene de `else`, `if` y puede traducirse como "si no se da la condición del `if` anterior, verificá si se da la siguiente".

Imprimir en pantalla

La función `print` imprime una línea de texto con el valor pasado como parámetro.

```
print('Hello world!') # Imprime 'Hello world!'
```

Podés imprimir variables. El texto impreso en ese caso será el valor de la variable y no su nombre.

```
x = 100
print(x) # imprime el texto '100'
```

Si le pasás más de un valor al `print` los separa con espacios.

```
name = 'Juana'
print('Mi nombre es', name) # Imprime el texto 'Mi nombre es Juana'
print() siempre termina la línea impresa pasando a la siguiente.
print('Hola')
print('Mi nombre es', 'Juana')
```

This prints:

```
Hola
Mi nombre es Juana
```

El salto de línea entre ambos comandos puede ser suprimido:

```
print('Hola', end=' ')
print('Mi nombre es', 'Juana')
```

Este código va a imprimir:

```
Hola Mi nombre es Juana
```

Ingreso de valores por teclado

Para leer un valor ingresado por el usuario, usá la función `input()`:

```
name = input('Ingresá tu nombre:')
```

```
print('Tu nombre es', name)
```

`input` imprime el texto que le pases como parámetro y espera una respuesta. Es útil para programas pequeños, ejercitarse o para debuguear un código. Casi no se lo usa en programas reales.

El comando `pass`

A veces es conveniente especificar un bloque de código que no haga nada. El comando `pass` se usa para eso.

```
if a > b:  
    pass  
else:  
    print('No ganó a')
```

Este comando no hace nada. Sirve para guardar el lugar para comando que querramos agregar luego.

Ejercicios

Ejercicio 1.4: Debuguear

El siguiente fragmento de código está relacionado con el problema del obelisco. Tiene un bug, es decir, un error.

```
# obelisco.py  
  
grosor_billete = 0.11 * 0.001 # 0.11 mm en metros  
altura_obelisco = 67.5          # altura en metros  
num_billetes = 1  
dia = 1  
  
while num_billetes * grosor_billete <= altura_obelisco:  
    print(dia, num_billetes, num_billetes * grosor_billete)  
    dia = dia + 1  
    num_billetes = num_billetes * 2  
  
print('Cantidad de días', dia)  
print('Cantidad de billetes', num_billetes)  
print('Altura final', num_billetes * grosor_billete)
```

Copíá y pegá el código que aparece arriba en un nuevo archivo llamado `obelisco.py`. Cuando ejecutes el código vas a obtener el siguiente mensaje de error que hace que el programa se detenga:

```
Traceback (most recent call last):
  File "obelisco.py", line 10, in <module>
    dia = dias + 1
NameError: name 'dias' is not defined
```

Aprender a leer y entender los mensajes de error es una parte fundamental de programar en Python. Si tu programa *crashea* (se rompe, da error) la última línea del mensaje de error indica el motivo. Sobre eso, vas a ver un fragmento de código, un nombre de archivo y un número de línea que identifican el problema.

- ¿En qué linea está el error?
- ¿Cuál es el error?
- Repará el error.
- Ejecutá el programa exitosamente.

Ejercicio 1.5: La pelota que rebota

Este es el primer conjunto de ejercicios en el que vas a tener que crear un archivo de Python y correrlo. A partir de aca, vamos a asumir que estás trabajando en el subdirectorio `ejercicios_python/`. Para ayudarte a ubicar el lugar correcto ya creamos un par de archivos en ese directorio. Buscá en tu terminal el archivo `ejercicios_python/rebotes.py` (cambiando de directorio como vimos recién). Lo vamos a usar en este ejercicio.

Una pelota de goma es arrojada desde una altura de 100 metros y cada vez que toca el piso salta $\frac{3}{5}$ de la altura desde la que cayó. Escribí un programa `rebotes.py` que imprima una tabla mostrando las alturas que alcanza en cada uno de sus primeros diez rebotes.

Tu programa debería hacer una tabla que se parezca a esta:

```
1 60.0
2 36.0
3 21.599999999999998
4 12.95999999999999
5 7.775999999999999
6 4.665599999999995
7 2.7993599999999996
8 1.6796159999999998
9 1.0077695999999998
10 0.6046617599999998
```

Nota: Podés limpiar un toque la salida si usás la función round() de la que miraste el help hace un rato. Tratá de usarla para redondear a cuatro dígitos.

```
1 60.0
2 36.0
3 21.6
4 12.96
5 7.776
6 4.6656
7 2.7994
8 1.6796
9 1.0078
10 0.6047
```

Ejercicio 1.6: Saludos

Escribí un programa llamado `saludo.py` que pregunte el nombre de le usuarie, imprima un saludo (por ejemplo, "Hola, Juana") y termine.

1.3 Números

Esta sección introduce los cálculos matemáticos.

Tipos de números

Python tiene 4 tipos de números:

- Booleanos
- Enteros
- Punto flotante
- Complejos (con parte real y parte imaginaria)

Booleanos (bool)

Las variables booleanas se llaman así en honor al lógico inglés George Bool. Pueden tomar dos valores: `True` y `False` (verdadero y falso).

```
a = True
b = False
```

Internamente, son evaluados como enteros con valores 1, 0.

```
c = 4 + True # 5
d = False
if d == 0:
    print('d is False')
```

No escribas código basado en esta convención. Sería bastante raro.

Enteros (int)

Representan números enteros (positivos y negativos) de cualquier magnitud:

```
a = 37
b = -299392993727716627377128481812241231
```

Incluso se pueden especificar en diferentes bases:

```
c = 0x7fa8      # Hexadecimal
d = 0o253       # Octal
e = 0b10001111 # Binario
```

Operaciones usuales:

x + y	Suma
x - y	Resta
x * y	Multiplicación
x / y	División (da un float, no un int)
x // y	División entera (da un int)
x % y	Módulo (resto)
x ** y	Potencia
abs(x)	Valor absoluto

La unidad mínima de almacenamiento de una computadora es un bit, que puede valer 0 o 1. Los números, caracteres e incluso imágenes y sonido son almacenados en la máquina usando bits. Los números enteros positivos, en particular, suelen almacenarse mediante su representación binaria (o en base dos).

Número	Representación binaria
1	1
2	10
3	11
4	100
5	101
6	110

Hay algunas operaciones primitivas que se pueden hacer con los enteros a partir de su representación como bits:

```
x << n      Desplazamiento de los bits a la izquierda
x >> n      Desplazamiento de los bits a la derecha
x & y        AND bit a bit.
x | y        OR bit a bit.
x ^ y        XOR bit a bit.
~x           NOT bit a bit.
```

Al desplazar a la izquierda, simplemente agregamos un cero en la última posición. Así, por ejemplo si corremos el 1 dos lugares a la izquierda obtenemos un 4:

```
>>> 1 << 2 # 1 << 2 -> 100
4
>>> 6 & 3 # 110 & 011 -> 010
2
```

Al desplazar los bits de un número a la derecha un lugar, el último bit "se cae".

```
>>> 1 >> 1 # 1 -> 0
0
>>> 6 >> 1 # 110 -> 11
3
```

Punto flotante (float)

Usá una notación con decimales o una notación científica para especificar un valor de tipo punto flotante:

```
a = 37.45
b = 4e5 # 4 x 10**5 o 400,000
c = -1.345e-10
```

Los números de tipo floats son representados en la máquina como números de doble precisión usando la representación nativa del microprocesador [IEEE 754](#). Es el mismo tipo que los `double` en el lenguaje C (para los que conozcan).

Almacenan hasta 17 dígitos con un exponente entre -308 to 308

Cuidado que la aritmética de los números de punto flotante no es exacta.

```
>>> a = 2.1 + 4.2
>>> a == 6.3
False
>>> a
6.300000000000001
>>>
```

Esto no es un problema de Python, si no el resultado de la forma en que el hardware de nuestras computadoras almacena los números de punto flotante.

Operaciones usuales:

<code>x + y</code>	Suma
<code>x - y</code>	Resta
<code>x * y</code>	Multiplicación
<code>x / y</code>	División (da un float, no un int)
<code>x // y</code>	División entera (da un float, pero con ceros luego del punto)
<code>x % y</code>	Módulo (resto)
<code>x ** y</code>	Potencia
<code>abs(x)</code>	Valor absoluto

Estas son las mismas operaciones que con los enteros. Otra operaciones usuales se encuentran en el módulo `math`.

```
import math
a = math.sqrt(x)
b = math.sin(x)
c = math.cos(x)
```

```
d = math.tan(x)
e = math.log(x)
```

El módulo `math` también tiene constantes (`math.e`, `math.pi`), entre otras cosas.

Comparaciones

Las siguientes comparaciones (suelen llamarse *operadores relacionales* ya que expresan una relación entre dos elementos) funcionan con números:

<code>x < y</code>	Menor que
<code>x <= y</code>	Menor o igual que
<code>x > y</code>	Mayor que
<code>x >= y</code>	Mayor o igual que
<code>x == y</code>	Igual a
<code>x != y</code>	No igual a

Observá que el `==` se usa para comparar dos elementos mientras que el `=` se usa para asignar un valor a una variable. Son símbolos distintos que cumplen funciones diferentes.

Podés formar expresiones booleanas más complejas usando

`and`, `or`, `not`

Acá mostramos algunos ejemplos:

```
if b >= a and b <= c:
    print('b está entre a y c')

if not (b < a or b > c):
    print('b sigue estando entre a y c')
```

Conversión de números

El nombre de un tipo (de datos) puede ser usado para convertir valores:

```
a = int(x)      # Convertir x a int
b = float(x)    # Convertir x a float
```

Probalo.

```
>>> a = 3.14159
>>> int(a)
3
>>> b = '3.14159' # También funciona con cadenas que representan números.
>>> float(b)
3.14159
```

>>>

Cuidado: el separador decimal en Python es el punto, como en inglés, y no la coma del castellano. Así, el comando `float(3,141592)` da un "ValueError".

Ejercicios

Recordatorio: Asumimos que estás trabajando en el subdirectorio /Ejercicios. Buscá el archivo `hipoteca.py` y hacé los ejercicios con un editor de texto en ese archivo. Ejecutalo desde la línea de comandos.

Ejercicio 1.7: La hipoteca de David

David solicitó un crédito a 30 años para comprar una vivienda, con una tasa fija nominal anual del 5%. Pidió \$500000 al banco y acordó un pago mensual fijo de \$2684,11.

El siguiente es un programa que calcula el monto total que pagará David a lo largo de los años:

```
# hipoteca.py

saldo = 500000.0
tasa = 0.05
pago_mensual = 2684.11
total_pagado = 0.0

while saldo > 0:
    saldo = saldo * (1+tasa/12) - pago_mensual
    total_pagado = total_pagado + pago_mensual

print('Total pagado', round(total_pagado, 2))
```

Copíá este código y correlo. Deberías obtener 966279.6 como respuesta.

Ejercicio 1.8: Adelantos

Supongamos que David adelanta pagos extra de \$1000/mes durante los primeros 12 meses de la hipoteca.

Modificá el programa para incorporar estos pagos extra y que imprima el monto total pagado junto con la cantidad de meses requeridos.

Cuando lo corras, este nuevo programa debería dar un pago total de 929965.62 en 342 meses.

Aclaración: aunque puede parecer sencillo, este ejercicio requiere que agregues una variable *mes* y que prestes bastante atención a cuándo la incrementás, con qué valor entra al ciclo y con qué valor sale del ciclo. Una posibilidad es inicializar *mes* en 0 y otra es inicializarla en 1. En el primer caso es probable que la variable salga del ciclo contando la cantidad de pagos que se hicieron, en el segundo, ¡es probable que salga contando la cantidad de pagos más uno!

Ejercicio 1.9: Calculadora de adelantos

¿Cuánto pagaría David si agrega \$1000 por mes durante cuatro años, comenzando en el sexto año de la hipoteca (es decir, luego de 5 años)?

Modificá tu programa de forma que la información sobre pagos extras sea incorporada de manera versatil. Agregá las siguientes variables antes del ciclo, para definir el comienzo, fin y monto de los pagos extras:

```
pago_extra_mes_comienzo = 61  
pago_extra_mes_fin = 108  
pago_extra = 1000
```

Hacé que el programa tenga en cuenta estas variables para calcular el total a pagar apropiadamente.

Ejercicio 1.10: Tablas

Modicá tu programa para que imprima una tabla mostrando el mes, el total pagado hasta el momento y el saldo restante. La salida debería verse aproximadamente así:

```
1 2684.11 499399.22  
2 5368.22 498795.94  
3 8052.33 498190.15  
4 10736.44 497581.83  
5 13420.55 496970.98  
...  
308 874705.88 3478.83  
309 877389.99 809.21  
310 880074.1 -1871.53  
Total pagado: 880074.1  
Meses: 310
```

Ejercicio 1.11: Bonus

Ya que estamos, corregí el código anterior de forma que el pago del último mes se ajuste a lo adeudado.

Asegurate de guardar el archivo `hipoteca.py` en esta última versión en tu directorio `Ejercicios`. Vamos a volver a trabajar con él.

Ejercicio 1.12: Un misterio

Las funciones `int()` y `float()` pueden usarse para convertir números. Por ejemplo,

```
>>> int("123")
123
>>> float("1.23")
1.23
>>>
```

Con esto en mente, ¿podrías explicar el siguiente comportamiento?

```
>>> bool("False")
True
>>>
```

Ejercicio 1.13: El volumen de una esfera

En tu directorio de trabajo, escribí un programa llamado `esfera.py` que le pida al usuario que ingrese por teclado el radio r de una esfera y calcule e imprima el volumen de la misma. *Sugerencia: recordar que el volumen de una esfera es $4/3 \pi r^3$.*

Finalmente, ejecutá el programa desde la línea de comandos para responder ¿cuál es el volumen de una esfera de radio 6? Debería darte 904.7786842338603.

1.4 Cadenas

En esta sección veremos cómo trabajar con textos.

Representación de textos

Las cadenas de caracteres entre comillas se usan para representar texto en Python. En este caso, fragmentos del Martín Fierro.

```
# Comillas simples
a = 'Aquí me pongo a cantar, al compás de la vigüela'

# Comillas dobles
b = "Los hermanos sean unidos porque ésa es la ley primera"

# Comillas triples
c = """
Yo no tengo en el amor
Quien me venga con querellas;
Como esas aves tan bellas
Que saltan de rama en rama
Yo hago en el trébol mi cama
Y me cubren las estrellas.
"""


```

Normalmente las cadenas de caracteres solo ocupan una linea. Las comillas triples nos permiten capturar todo el texto encerrado a lo largo de múltiples lineas.

No hay diferencia entre las comillas simples ('') y las dobles (""). *Pero el mismo tipo de comillas que se usó para abrir debe usarse para cerrar.*

Código de escape

Los códigos de escape (escape codes) son expresiones que comienzan con una barra invertida, \ y se usan para representar caracteres que no pueden ser fácilmente tipeados directamente con el teclado. Estos son algunos códigos de escape usuales:

'\n'	Avanzar una línea
'\r'	Retorno de carro
'\t'	Tabulador
'\\'	Comilla literal
'\\''	Comilla doble literal
'\\\\'	Barra invertida literal

El *retorno de carro* (código '\r') mueve el cursor al comienzo de la línea pero sin avanzar una línea. El origen de su nombre está relacionado con las máquinas de escribir.

Representación en memoria de las cadenas

Las cadenas se representan en Python asociando a cada carácter un número entero o código [Unicode](#). Es posible definir un carácter usando su código y códigos de escape como `s = '\U0001D120'` para la clave de sol.

Indexación de cadenas

Las cadenas funcionan como vectores, permitiendo el acceso a los caracteres individuales. El índice comienza a contar en cero. Los índices negativos se usan para especificar una posición respecto al final de la cadena.

```
a = 'Hello world'  
b = a[0]          # 'H'  
c = a[4]          # 'o'  
d = a[-1]         # 'd' (fin de cadena)
```

También se puede *rebanar* (slice) o seleccionar subcadenas especificando un range de índices con `:`.

```
d = a[:5]        # 'Hello'  
e = a[6:]         # 'world'  
f = a[3:8]        # 'lo wo'  
g = a[-5:]        # 'world'
```

El carácter que corresponde al último índice no se incluye. Si un extremo no se especifica, significa que es *desde el comienzo* o *hasta el final*, respectivamente.

Operaciones con cadenas

Concatenación, longitud, pertenencia y replicación.

```
# Concatenación (+)  
a = 'Hello' + 'World'    # 'HelloWorld'  
b = 'Say ' + a           # 'Say HelloWorld'  
  
# Longitud (len)  
s = 'Hello'  
len(s)                  # 5  
  
# Test de pertenencia (in, not in)  
t = 'e' in s             # True  
f = 'x' in s             # False  
g = 'hi' not in s        # True  
  
# Replicación (s * n)
```

```
rep = s * 5           # 'HelloHelloHelloHelloHello'
```

Métodos de las cadenas

Las cadenas en Python tienen *métodos* que realizan diversas operaciones con este tipo de datos.

Ejemplo: sacar (`strip`) los espacios en blanco sobrantes al inicio o al final de una cadena.

```
s = 'Hello '
t = s.strip()      # 'Hello'
```

Ejemplo: Conversión entre mayúsculas y minúsculas.

```
s = 'Hello'
l = s.lower()      # 'hello'
u = s.upper()      # 'HELLO'
```

Ejemplo: Reemplazo de texto.

```
s = 'Hello world'
t = s.replace('Hello' , 'Hallo')    # 'Hallo world'
```

Más métodos de cadenas:

Los strings (cadenas) ofrecen una amplia variedad de métodos para testear y manipular textos. Estos son algunos de los métodos:

<code>s.endswith(suffix)</code>	# Verifica si termina con el sufijo
<code>s.find(t)</code>	# Primera aparición de t en s (o -1 si no está)
<code>s.index(t)</code>	# Primera aparición de t en s (error si no está)
<code>s.isalpha()</code>	# Verifica si los caracteres son alfabéticos
<code>s.isdigit()</code>	# Verifica si los caracteres son numéricos
<code>s.islower()</code>	# Verifica si los caracteres son minúsculas
<code>s.isupper()</code>	# Verifica si los caracteres son mayúsculas
<code>s.join(slist)</code>	# Une una lista de cadenas usando s como delimitador
<code>s.lower()</code>	# Convertir a minúsculas
<code>s.replace(old,new)</code>	# Reemplaza texto
<code>s.split([delim])</code>	# Parte la cadena en subcadenas
<code>s.startswith(prefix)</code>	# Verifica si comienza con un sufijo
<code>s.strip()</code>	# Elimina espacios en blanco al inicio o al final
<code>s.upper()</code>	# Convierte a mayúsculas

Mutabilidad de cadenas

Los strings son "inmutables" o de sólo lectura. Una vez creados, su valor no puede ser cambiado.

```
>>> s = 'Hello World'  
>>> s[1] = 'a' # Intento cambiar la 'e' por una 'a'  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: 'str' object does not support item assignment  
>>>
```

Esto implica que las operaciones y métodos que manipulan cadenas deben crear nuevas cadenas para almacenar su resultado.

Conversión de cadenas

Usá `str()` para convertir cualquier valor a cadena. El resultado es una cadena con el mismo contenido que hubiera mostrado el comando `print()` sobre la expresión entre paréntesis.

```
>>> x = 42  
>>> str(x)  
'42'  
>>>
```

f-Strings

Las f-Strings son cadenas en las que ciertas expresiones son formateadas

```
>>> nombre = 'Naranja'  
>>> cajones = 100  
>>> precio = 91.1  
>>> a = f'{nombre:>10s} {cajones:10d} {precio:10.2f}'  
>>> a  
'    Naranja      100      91.10'  
>>> b = f'Costo = ${cajones*precio:0.2f}'  
>>> b  
'Costo = $9110.00'  
>>>
```

Nota: Esto requiere Python 3.6 o uno más nuevo. El significado de los códigos lo veremos más adelante.

Ejercicios

En estos ejercicios vas a experimentar con operaciones sobre el tipo de dato string de Python. Hacelo en el intérprete interactivo para ver inmediatamente los resultados.

Recordamos:

En los ejercicios donde interactuás con el intérprete, el símbolo `>>>` es el que usa Python para indicarte que espera un nuevo comando. Algunos comandos ocupan más de una línea de código --para que funcionen, vas a tener que apretar 'enter' algunas veces. Acordate de no copiar el `>>>` de los ejemplos.

Comencemos definiendo una cadena que contiene una lista de frutas así::

```
>>> frutas = 'Manzana,Naranja,Mandarina,Banana,Kiwi'  
>>>
```

Ejercicio 1.14: Extraer caracteres individuales y subcadenas

Los strings son vectores de caracteres. Tratá de extraer algunos carateres:

```
>>> frutas[0]  
?  
>>> frutas[1]  
?  
>>> frutas[2]  
?  
>>> frutas[-1]      # Último carácter  
?  
>>> frutas[-2]      # Índices negativos se cuentan desde el final  
?  
>>>
```

Como ya dijimos, en Python los strings son sólo de lectura. Verificá esto tratando de cambiar el primer carácter de `frutas` por una m minúscula 'm'.

```
>>> frutas[0] = 'm'  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'str' object does not support item assignment  
>>>
```

Ejercicio 1.15: Concatenación de cadenas

A pesar de ser sólo de lectura, siempre podés reasignar una variable a una cadena nueva. Probá el siguiente comando que concatena la palabra "Pera" al final de frutas:

```
>>> frutas = frutas + 'Pera'  
>>> frutas  
'Manzana,Naranja,Mandarina,Banana,KiwiPera'  
>>>
```

Ups! No es exactamente lo que queríamos. Reparalo para que quede

```
'Manzana,Naranja,Mandarina,Banana,Kiwi,Pera'.  
>>> frutas = ?  
>>> frutas  
'Manzana,Naranja,Mandarina,Banana,Kiwi,Pera'  
>>>
```

Agregá 'Melón' al principio de la cadena:

```
>>> frutas = ?  
>>> frutas  
'Melón,Manzana,Naranja,Mandarina,Banana,Kiwi,Pera'  
>>>
```

Podría parecer en estos ejemplos que la cadena original está siendo modificada, contradiciendo la regla de que las cadenas son de sólo lectura. No es así. Las operaciones sobre cadenas crean una nueva cadena cada vez. Cuando la variable `frutas` es reasignada, apunta a la cadena recientemente creada. Luego, la cadena vieja es destruída dado que ya no está siendo usada.

Ejercicio 1.16: Testeo de pertenencia (test de subcadena)

Experimentá con el operador `in` para buscar subcadenas. En el intérprete interactivo probá estas operaciones:

```
>>> 'Naranja' in frutas  
?  
>>> 'nana' in frutas  
True  
>>> 'Lima' in frutas  
?  
>>>
```

¿Por qué la verificación de 'nana' dió True?

Ejercicio 1.17: Iteración sobre cadenas

Usá el comando `for` para iterar sobre los caracteres de una cadena.

```
>>> cadena = "Ejemplo con for"
>>> for c in cadena:
...     print('caracter:', c)
# Mirá el output.
```

Modificá el código anterior de manera que dentro del ciclo el programa cuente cuántas letras "o" hay en la cadena.

Sugerencia: usá un contador como con los meses de la hipoteca.

Ejercicio 1.18: Geringoso rústico

Usá una iteración sobre el string `cadena` para agregar la sílaba 'pa', 'pe', 'pi', 'po', o 'pu' según corresponda luego de cada vocal.

```
>>> cadena = 'Geringoso'
>>> capadepenapa = ''
>>> for c in cadena:
...     ?
>>> capadepenapa
Geperipingoposopo
```

Podés probar tu código cambiando la cadena inicial por otra palabra, como 'apa' o 'boligoma'.

Guardá el código en un archivo `geringoso.py`.

Ejercicio 1.19: Métodos de cadenas

En el intérprete interactivo experimentá con algunos de los métodos de cadenas introducidos antes.

```
>>> frutas.lower()
?
>>> frutas
?
>>>
```

Recordá, las cadenas son siempre de sólo lectura. Si querés guardar el resultado de una operación, vas a necesitás asignárselo a una variable:

```
>>> lowersyms = frutas.lower()  
>>>
```

Probá algunas más:

```
>>> frutas.find('Mandarina')  
?  
>>> frutas[13:17]  
?  
>>> frutas = frutas.replace('Kiwi','Melón')  
>>> frutas  
?  
>>> nombre = '    Naranja    \n'  
>>> nombre = nombre.strip()      # Remove surrounding whitespace  
>>> nombre  
?  
>>>
```

Ejercicio 1.20: f-strings

A veces querés crear una cadena que incorpore los valores de otras variables en ella.

Para hacer eso, usá una f-string. Por ejemplo:

```
>>> nombre = 'Naranja'  
>>> cajones = 100  
>>> precio = 91.1  
>>> f'{cajones} cajones de {nombre} a ${precio:0.2f}'  
'100 cajones de Naranja a $91.10'  
>>>
```

Modificá el programa `hipoteca.py` del [Ejercicio 1.11](#) de la sección anterior para que escriba su salida usando f-strings. Tratá de hacer que la salida quede bien alineada.

Ejercicio 1.21: Expresiones regulares

Una limitación de las operaciones básicas de cadenas es que no ofrecen ningún tipo de transformación usando patrones más sofisticados. Para eso vas a tener que usar el módulo `re` de Python y aprender a usar expresiones regulares. El manejo de estas expresiones es un tema en sí mismo. A continuación presentamos un corto ejemplo:

```

>>> texto = 'Hoy es 6/8/2020. Mañana será 7/8/2020.'
>>> # Encontrar las apariciones de una fecha en el texto
>>> import re
>>> re.findall(r'\d+/\d+/\d+', texto)
['6/8/2020', '7/8/2020']
>>> # Reemplazá esas apariciones, cambiando el formato
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\2-\1', texto)
'Hoy es 2020-8-6. Mañana será 2020-8-7.'
>>>

```

Para más información sobre el módulo `re`, mirá la [documentación oficial en inglés](#) o algún [tutorial en castellano](#).

Commentario

A medida que empezás a usar Python es usual que quieras saber qué otras operaciones admiten los objetos con los que estás trabajando. Por ejemplo. ¿cómo podés averiguar qué operaciones se pueden hacer con una cadena?

Dependiendo de tu entorno de Python, podrás ver una lista de métodos disponibles apretando la tecla tab. Por ejemplo, intentá esto:

```

>>> s = 'hello world'
>>> s.<tecla tab>
>>>

```

Si al presionar tab no pasa nada, podés volver al viejo uso de la función `dir()`. Por ejemplo:

```

>>> s = 'hello'
>>> dir(s)
['__add__', '__class__', '__contains__', ..., 'find', 'format',
'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>>

```

`dir()` produce una lista con todas las operaciones que pueden aparecer luego del parámetro que le pasaste, en este caso `s`. También podés usar el comando `help()` para obtener más información sobre una operación específica:

```

>>> help(s.upper)
Help on built-in function upper:

upper(...)

```

```
S.upper() -> string
```

```
Return a copy of the string S converted to uppercase.
```

1.5 Listas

En esta sección estudiaremos listas que es el tipo de datos primitivo de Python para guardar colecciones ordenadas de valores.

Creación de Listas

Usá corchetes para definir una lista:

```
nombres = [ 'Rosita', 'Manuel', 'Luciana' ]  
nums = [ 39, 38, 42, 65, 111 ]
```

A veces las listas son creadas con otros métodos. Por ejemplo, los elementos de una cadena pueden ser separados en una lista usando el método `split()`:

```
>>> line = 'Pera,100,490.10'  
>>> row = line.split(',') #la coma indica el elemento que separa  
>>> row  
['Pera', '100', '490.10']  
>>>
```

Operaciones con listas

Las listas pueden almacenar elementos de cualquier tipo. Podés agregar nuevos elementos usando `append()`:

```
nombres.append('Mauro') # Lo agrega al final
```

Usá el símbolo de adición + para concatenar listas:

```
s = [1, 2, 3]  
t = ['a', 'b']  
s + t # [1, 2, 3, 'a', 'b']
```

Las listas se indexan con número enteros, comenzando en 0.

```
nombres = [ 'Rosita', 'Manuel', 'Luciana' ]  
nombres[0] # 'Rosita'
```

```
nombres[1] # 'Manuel'  
nombres[2] # 'Luciana'
```

Los índices negativos cuentan desde el final.

```
nombres[-1] # 'Luciana'
```

Podés cambiar cualquier elemento de una lista.

```
nombres[1] = 'Juan Manuel'  
nombres # [ 'Rosita', 'Juan Manuel', 'Luciana' ]
```

Y podés insertar elementos en una posición. Acordate que los índices comienzan a contar desde el 0.

```
nombres.insert(2, 'Iratxe') # Lo inserta en la posición 2.  
nombres.insert(0, 'Iratxe') # Lo inserta como primer elemento.
```

La función `len` permite obtener la longitud de una lista.

```
nombres = ['Rosita', 'Manuel', 'Luciana']  
len(nombres) # 3
```

Test de pertenencia a la lista (`in`, `not in`).

```
'Rosita' in nombres # True  
'Diego' not in nombres # True
```

Se puede replicar una lista (`s * n`).

```
s = [1, 2, 3]  
s * 3 # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Iteradores de listas y búsqueda

Usá el comando `for` para iterar sobre los elementos de una lista.

```
for nombre in nombres:  
    # usá nombre  
    # e.g. print(nombre)  
    ...
```

Para encontrar rápidamente la posición de un elemento en una lista, usá `index()`.

```
nombres = ['Rosita', 'Manuel', 'Luciana']
```

```
nombres.index('Luciana') # 2
```

Si el elemento está presente en más de una posición, `index()` te va a devolver el índice de la primera aparición. Si el elemento no está en la lista se va a generar una excepción de tipo `ValueError`.

Borrar elementos

Podés borrar elementos de una lista tanto usando el valor del elemento como su posición:

```
# Usando el valor  
nombres.remove('Luciana')  
  
# Usando la posición  
del nombres[1]
```

Al borrar un elemento no se genera un hueco. Los siguientes elementos se moverán para llenar el vacío. Si hubiera más de una aparición de un valor, `remove()` sólo sacará la primera aparición.

Ordenar una lista

Las listas pueden ser ordenadas "in-place", es decir, sin usar nuevas variables.

```
s = [10, 1, 7, 3]  
s.sort() # [1, 3, 7, 10]  
  
# Orden inverso  
s = [10, 1, 7, 3]  
s.sort(reverse=True) # [10, 7, 3, 1]  
  
# Funciona con cualquier tipo de datos que tengan orden  
s = ['foo', 'bar', 'spam']  
s.sort() # ['bar', 'foo', 'spam']
```

Usá `sorted()` si querés generar una nueva lista ordenada en lugar de ordenar la misma:

```
t = sorted(s) # s queda igual, t guarda los valores ordenados
```

Listas y matemática

Cuidado: Las listas no fueron diseñadas para realizar operaciones matemáticas.

```
>>> nums = [1, 2, 3, 4, 5]
>>> nums * 2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>>> nums + [10, 11, 12, 13, 14]
[1, 2, 3, 4, 5, 10, 11, 12, 13, 14]
```

Específicamente, las listas no representan vectores ni matrices como en MATLAB, Octave, R, etc. Sin embargo, hay paquetes de Python que hacen muy bien ese trabajo (por ejemplo [numpy](#)).

Ejercicios

En este ejercicio, vamos a experimentar con el tipo de dato *lista* de Python. En la última sección, trabajaste con cadenas que hacían referencia a cajones de frutas.

```
>>> frutas = 'Frambuesa,Manzana,Naranja,Mandarina,Banana,Sandía,Pera'
```

Armá una lista con los nombres de frutas usando el comando `split()`:

```
>>> lista_frutas = frutas.split(',')
```

Ejercicio 1.22: Extracción y reasignación de elementos.

Probá un par de estos comandos para extraer un elemento:

```
>>> lista_frutas[0]
'Frambuesa'
>>> lista_frutas[1]
'Manzana'
>>> lista_frutas[-1]
'Pera'
>>> lista_frutas[-2]
'Sandía'
>>>
```

Intentá reasignar un valor:

```
>>> lista_frutas[2] = 'Granada'
>>> lista_frutas
['Frambuesa', 'Manzana', 'Granada', 'Mandarina', 'Banana', 'Sandía',
 'Pera']
>>>
```

Hacé unas rebanadas (slices) de la lista:

```
>>> lista_frutas[0:3]
```

```
['Frambuesa', 'Manzana', 'Granada']
>>> lista_frutas[-2:]
['Sandía', 'Pera']
>>>
```

Creá una lista vacía y agregale un elemento.

```
>>> compra = []
>>> compra.append('Pera')
>>> compra
['Pera']
```

Podés incluso reasignar una lista a una porción de otra lista. Por ejemplo:

```
>>> lista_frutas[-2:] = compra
>>> lista_frutas
['Frambuesa', 'Manzana', 'Granada', 'Mandarina', 'Banana', 'Pera']
>>>
```

Cuando hacés esto, la lista del lado izquierdo (`lista_frutas`) va a cambiar su tamaño para que encaje la lista del lado derecho (`compra`). En el ejemplo de arriba los últimos dos elementos de la `lista_frutas` fueron reemplazados por un solo elemento en la lista `compra`.

Ejercicio 1.23: Ciclos sobre listas

El ciclo `for` funciona iterando sobre datos en una secuencia. Antes vimos que podíamos iterar sobre los caracteres de una cadena (las cadenas son secuencias). Ahora veremos que podemos iterar sobre listas también. Verificá esto tipeando lo que sigue y viendo qué pasa:

```
>>> for s in lista_frutas:
    print('s =', s)
```

Ejercicio 1.24: Test de pertenencia

Usá los operadores `in` o `not in` para verificar si 'Granada', 'Lima', y 'Limon' pertenecen a la lista de frutas.

```
>>> # ¿Está 'Granada' IN `lista_frutas`?
True
>>> # ¿Está 'Lima' IN `lista_frutas`?
False
>>> # ¿Está 'Limon' NOT IN `lista_frutas`?
True
>>>
```

Ejercicio 1.25: Adjuntar, insertar y borrar elementos

Usá el método `append()` para agregar 'Mango' al final de `lista_frutas`.

```
>>> # agregar 'Mango'  
>>> lista_frutas  
['Frambuesa', 'Manzana', 'Granada', 'Mandarina', 'Banana', 'Pera', 'Mango']  
>>>
```

Usá el método `insert()` para agregar 'Lima' como segundo elemento de la lista.

```
>>> # Insertar 'Lima' como segundo elemento  
>>> lista_frutas  
['Frambuesa', 'Lima', 'Manzana', 'Granada', 'Mandarina', 'Banana', 'Pera',  
'Mango']  
>>>
```

Usá el método `remove()` para borrar 'Mandarina' de la lista.

```
>>> # Borrar 'Mandarina'  
>>> lista_frutas  
['Frambuesa', 'Lima', 'Manzana', 'Granada', 'Banana', 'Pera', 'Mango']  
>>>
```

Agregá una segunda copia de 'Banana' al final de la lista.

Observación: es perfectamente válido tener valores duplicados en una lista.

```
>>> # Agregar 'Banana'  
>>> lista_frutas  
['Frambuesa', 'Lima', 'Manzana', 'Granada', 'Banana', 'Pera', 'Mango',  
'Banana']  
>>>
```

Usá el método `index()` para determinar la posición de la primera aparición de 'Banana' en la lista.

```
>>> # Encontrar la primera aparición de 'Banana'  
>>> lista_frutas  
4  
>>> lista_frutas[4]  
'Banana'  
>>>
```

Contá la cantidad de apariciones de 'Banana' en la lista:

```
>>> lista_frutas.count('Banana')
2
>>>
```

Borrá la primera aparición de 'Banana'.

```
>>> # Borrar la primer aparición de 'Banana'
>>> lista_frutas
['Frambuesa', 'Lima', 'Manzana', 'Granada', 'Pera', 'Mango', 'Banana']
>>>
```

Para que sepas, no hay un método que permita encontrar o borrar *todas* las apariciones de un elemento en una lista. Más adelante veremos una forma elegante de hacerlo.

Ejercicio 1.26: Sorting

¿Querés ordenar una lista? Usá el método `sort()`. Probalo:

```
>>> lista_frutas.sort()
>>> lista_frutas
['Banana', 'Frambuesa', 'Granada', 'Lima', 'Mango', 'Manzana', 'Pera']
>>>
```

¿Y si ordenamos al revés?

```
>>> lista_frutas.sort(reverse=True)
>>> lista_frutas
['Pera', 'Manzana', 'Mango', 'Lima', 'Granada', 'Frambuesa', 'Banana']
>>>
```

Observación: acordate de que el método `sort()` modifica el contenido de la misma lista *in-place*. Los elementos son reordenados moviéndolos de una posición a otra, pero no se crea una nueva lista.

Ejercicio 1.27: Juntar múltiples cadenas

Si querés juntar las cadenas en una lista, usá el método `join()` de los strings como sigue (ojo: parece un poco raro al principio).

```
>>> lista_frutas = ['Banana', 'Mango', 'Frambuesa', 'Pera', 'Granada',
'Manzana', 'Lima']
>>> a = ','.join(lista_frutas)
>>> a
'Banana,Mango,Frambuesa,Pera,Granada,Manzana,Lima'
>>> b = ':'.join(lista_frutas)
```

```
>>> b  
'Banana:Mango:Frambuesa:Pera:Granada:Manzana:Lima'  
>>> c = ''.join(lista_frutas)  
>>> c  
'BananaMangoFrambuesaPeraGranadaManzanaLima'  
>>>
```

Ejercicio 1.28: Listas de cualquier cosa

Las listas pueden contener cualquier tipo de objeto, incluyendo otras listas (serían 'listas anidadas').

Probá esto:

```
>>> nums = [101, 102, 103]  
>>> items = ['spam', lista_frutas, nums]  
>>> items  
['spam', ['Banana', 'Mango', 'Frambuesa', 'Pera', 'Granada', 'Manzana',  
'Lima'], [101, 102, 103]]
```

Fijate bien el output. `items` es una lista con tres elementos. El primero es un string, pero los otros dos elementos son listas.

Podés acceder a los elementos de las listas anidadas usando múltiples operaciones de acceso por índice.

```
>>> items[0]  
'spam'  
>>> items[0][0]  
's'  
>>> items[1]  
['Banana', 'Mango', 'Frambuesa', 'Pera', 'Granada', 'Manzana', 'Lima']  
>>> items[1][1]  
'Mango'  
>>> items[1][1][2]  
'n'  
>>> items[2]  
[101, 102, 103]  
>>> items[2][1]  
102  
>>>
```

A pesar de que es técnicamente posible hacer una estructura de listas muy complicada, como regla general, es mejor mantener las cosas simples. Lo más usual es guardar en las listas muchos elementos del mismo tipo. Por ejemplo, una

lista sólo de números o una lista de cadenas. Mezclar diferentes tipos de datos en una misma lista puede volverse conceptualmente difuso, así que mejor lo evitamos.

Ejercicio 1.29: Traductor (rústico) al lenguaje inclusivo

Queremos hacer un traductor que cambie las palabras masculinas de una frase por su versión neutra. Como primera aproximación, completá el siguiente código para reemplazar todas las letras 'o' que figuren en el último o anteúltimo carácter de cada palabra por una 'e'. Por ejemplo 'todos somos programadores' pasaría a ser 'todes somes programdores'. Guardá tu código en el archivo `inclusive.py`

```
>>> frase = 'todos somos programadores'
>>> palabras = frase.split()
>>> for palabra in palabras:
    if ?
        ...
    frase_t = ?
    print(frase_t)
'todes somes programdores'
>>>
```

Probá tu código con 'Los hermanos sean unidos porque ésa es la ley primera', '¿cómo transmitir a los otros el infinito Aleph?' y 'Todos, tu también'. ¿Qué falla en esta última? (¡no hace falta que lo resuelvas!)

1.6 Cierre de la primer clase

En esta clase aprendimos a correr el intérprete de Python desde la línea de comandos para usarlo como una calculadora. Aprendimos a editar programas con un editor de texto y a correrlos en la terminal. Vimos diferentes tipos de datos: números enteros, números de punto flotante, cadenas y listas.

Necesitamos que leas el [código de honor](#) del curso en el que hablamos de las reglas que rigen en este curso para evitar el plagio así como otros aspectos importantes sobre qué se puede compartir y qué no. Al enviar tus archivos entendemos que leíste y estas de acuerdo con este texto. En caso contrario no envíes tus archivos y contactate con los docentes.

Para cerrar esta clase te pedimos dos cosas:

- Que recopiles las soluciones de los siguientes ejercicios:
 - i. [Ejercicio 1.5 Pelota](#) (`rebotes.py`)
 - ii. [Ejercicio 1.11 Bonus de Hipoteca](#) (`hipoteca.py`)

- iii. [Ejercicio 1.13 Volumen de la esfera](#) (`esfera.py`)
 - iv. [Ejercicio 1.18 Geringoso](#) (`geringoso.py`)
 - v. [Ejercicio 1.29 Lenguaje inclusivo](#) (`inclusive.py`)
- Que completes [este formulario](#) usando como identificación tu dirección de mail. Al terminar vas a obtener un link para enviarnos tus ejercicios y te vamos a preguntar si querés participar de la revisión de ejercicios por pares (peer-review). En ese caso te vamos a mandar para que corrijas un par de ejercicios y vas a recibir un par de correcciones.

Por favor, usá siempre la misma dirección de mail con la que te inscribiste al curso así podemos llevar registro de tus entregas.

Observación: Si el enunciado de un ejercicio te pide que lo corras con un input particular, por favor poné la salida que obtuviste como comentario en tu código.

Por último te recordamos que si te quedaron dudas, querés discutir algún tema de interés o pedirnos a los docentes que resolvamos un ejercicio particular para la próxima clase, podés hacerlo en el [grupo de Slack](#).

2. Trabajando con datos

Para escribir programas útiles, necesitamos aprender a trabajar con datos. En esta clase vas a escribir un programa que lee un archivo de datos en formato csv y realiza un cálculo simple. Vas a aprender a estructurar tu código creando funciones. También introducimos las estructuras de datos elementales de Python que nos faltan: tuplas, conjuntos y diccionarios y profundizamos un poco más en las listas y sus usos.

Ojo, esta clase tiene la mayoría de los ejercicios para entregar en las últimas secciones. ¡Administrá tu tiempo para llegar a hacerlos adecuadamente!

Recordá que, aunque no termines todos, completar el formulario del final de cada clase y entregar los ejercicios es parte de las condiciones de aprobación de la materia (todes nos podemos atrasar alguna vez, pero en general esperamos que entregues). Esto debe hacerse ANTES de las 14hs del día miércoles 19/8 para esta clase.

- [2.1 Manejo de archivos](#)
- [2.2 Funciones](#)
- [2.3 Tipos y estructuras de datos](#)
- [2.4 Contenedores](#)
- [2.5 Secuencias](#)
- [2.6 Contadores del módulo *collections*](#)
- [2.7 Arbolado porteño](#)
- [2.8 Impresión con formato](#)
- [2.9 Cierre de la segunda clase](#)

2.1 Manejo de archivos

La mayoría de los programas usa alguna fuente de datos. En esta sección discutimos el acceso a archivos.

Archivos de entrada y salida

Con estos comandos podés abrir dos archivos, una para lectura y otro para escritura:

```
f = open('foo.txt', 'rt')      # Abrir para lectura ('r' de read, 't' de text)
```

```
g = open('bar.txt', 'wt')      # Abrir para escritura ('w' de write, 't' de text)
```

Para leer el archivo completo, o una parte:

```
data = f.read()

# Leer 'maxbytes' bytes
data = f.read([maxbytes])
```

Para escribir un texto en el archivo:

```
g.write('un texto')
```

Finalmente, hay que cerrar los archivos cuando terminamos de usarlos.

```
f.close()
g.close()
```

Es importante cerrar adecuadamente los archivos y es bastante fácil olvidarse (puede que el programa termine y no se termine de guardar bien). Por eso, preferimos abrir los archivos con el comando `with` de la siguiente forma.

```
with open(nombre_archivo, 'rt') as file:
    # Usá el archivo `file`
    ...comandos que usan el archivo
    # No hace falta cerrarlo explícitamente
    ...comandos que no usan el archivo
```

Esto cierra automáticamente el archivo cuando se termina de ejecutar el bloque indentado.

Observación: En algunos sistemas operativos es probable que le tangas que especificar el encoding agregando `encoding='utf8'` como parámetro al comando `open`.

Comandos usuales para leer un archivo

Para leer un archivo entero, todo de una, como cadena:

```
with open('foo.txt', 'rt') as file:
    data = file.read()
    # `data` es una cadena con *todo* el texto en `foo.txt`
```

Para leer línea por línea iterativamente:

```
with open(nombre_archivo, 'rt') as file:  
    for line in file:  
        # Procesar la línea
```

Comandos usuales para escribir un archivo

Para escribir cadenas:

```
with open('outfile', 'wt') as out:  
    out.write('Hello World\n')  
    ...
```

También podés simplemente redireccionar la salida del print (de la pantalla a un archivo).

```
with open('outfile', 'wt') as out:  
    print('Hello World', file=out)  
    ...
```

Ejercicios

Estos ejercicios usan el archivo `Data/camion.csv`. El archivo contiene una lista de líneas con información sobre los cajones de fruta cargados en un camión. Suponemos que estás trabajando en el directorio `ejercicios_python/` del curso. Si no estás seguro, podés pedirle al Python que te diga dónde está trabajando con este comando:

```
>>> import os  
>>> os.getcwd()  
'/Users/profe/Desktop/curso-python/Ejercicios' # La salida va a cambiar  
>>>
```

Ejercicio 2.1: Preliminares sobre lectura de archivos

Primero, tratá de leer el archivo entero de una en una larga cadena:

```
>>> with open('Data/camion.csv', 'rt') as f:  
    data = f.read()  
  
>>> data  
'nombre,cajones,precio\n"Lima",100,32.20\n"Naranja",50,91.10\n"Caqui",150,8  
3.44\n"Mandarina",200,51.23\n"Durazno",95,40.37\n"Mandarina",50,65.10\n"Na  
ranja",100,70.44\n'  
>>> print(data)  
nombre,cajones,precio  
"Lima",100,32.20
```

```
"Naranja",50,91.10
"Caqui",150,83.44
"Mandarina",200,51.23
"Durazno",95,40.37
"Mandarina",50,65.10
"Naranja",100,70.44
>>>
```

En el ejemplo de arriba podrás observar que Python tiene dos modos de salida. En el primero escribiste `data` en el intérprete y Python mostró la representación *cruda* de la cadena, incluyendo comillas y códigos de escape. Cuando escribiste `print(data)`, en cambio, obtuviste la salida formateada de la cadena.

Leer un archivo entero y cargarlo en memoria todo de una vez parece simple, pero sólo tiene ventajas si el archivo es pequeño. Si estás trabajando con archivos enormes es mejor procesar las líneas de tu archivo una a una.

Para leer una archivo línea por línea, usá un ciclo `for` como éste:

```
>>> with open('Data/camion.csv', 'rt') as f:
    for line in f:
        print(line, end='')

nombre,cajones,precio
"Lima",100,32.20
"Naranja",50,91.10
...
>>>
```

En ese código, las líneas son leídas una por una hasta el final del archivo, cuando el ciclo se termina.

En ciertas ocasiones, puede pasar que quieras leer una sola línea de texto (por ejemplo, querés saltarte la primera línea del archivo que contiene los nombres de las columnas).

```
>>> f = open('Data/camion.csv', 'rt')
>>> headers = next(f)
>>> headers
'nombre,cajones,precio\n'
>>> for line in f:
    print(line, end='')

"Lima",100,32.20
"Naranja",50,91.10
...
>>> f.close()
```

```
>>>
```

El comando `next()` devuelve la siguiente línea de texto en el archivo. Sin embargo, sólo para que sepas, los ciclos `for` usan el método `next()` para obtener sus datos. Por lo tanto, típicamente no forzás un llamado extra a `next()` salvo que explícitamente quieras saltar o leer una línea particular como en nuestro caso de acá abajo.

Una vez que estés leyendo un archivo línea a línea, podés hacer otras operaciones, como separar los datos dentro de una línea con el método `split()`. Por ejemplo, probá esto:

```
>>> f = open('Data/camion.csv', 'rt')
>>> headers = next(f).split(',')
>>> headers
['nombre', 'cajones', 'precio\n']
>>> for line in f:
...     row = line.split(',')
...     print(row)

['"Lima"', '100', '32.20\n']
['"Naranja"', '50', '91.10\n']
...
>>> f.close()
```

Observación: En estos ejemplos tuvimos que llamar a `f.close()` explícitamente porque no estamos trabajando con el comando `with`.

Ejercicio 2.2: Lectura de un archivo de datos

Ahora que sabés leer un archivo, escribamos un programa que haga un cálculo simple con los datos leídos.

Las columnas en `camion.csv` corresponden a un nombre de fruta, una cantidad de cajones cargados en el camión, y un precio de compra por cada cajón de ese grupo. Escribí un programa llamado `costo_camion.py` que abra el archivo, lea las líneas, y calcule el precio pagado por los cajones cargados en el camión.

Ayuda: para interpretar un string `s` como un número entero, usá `int(s)`. Para leerlo como punto flotante, usá `float(s)`.

Tu programa debería imprimir una salida como la siguiente:

```
Costo total 47671.15
```

Acordate de guardar tu archivo. Vamos a volver a trabajar sobre él.

Ejercicio 2.3: Archivos comprimidos

¿Que pasaría si quisiéramos leer un archivo comprimido con gzip, por ejemplo? La función primitiva de Python `open()` no hace esa tarea. Pero hay un módulo de la biblioteca de Python llamado `gzip` que lee archivos comprimidos.

Probalo:

```
>>> import gzip
>>> with gzip.open('Data/camion.csv.gz', 'rt') as f:
    for line in f:
        print(line, end='')

... mirá la salida ...
>>>
```

Observación: La inclusión del modo 'rt' es crítica acá. Si te lo olvidás, vas a estar leyendo cadenas de bytes en lugar de cadenas de caracteres.

Comentario: ¿No deberíamos estar usando Pandas para esto?

Es frecuente que los estudiantes que conocen un poco más de Python rápidamente señalen que hay módulos como `Pandas` que tienen, entre muchas otras funcionalidades, la posibilidad de leer archivos CSV en una sola instrucción. Es verdad, y funcionan muy bien. Sin embargo, este no es un curso sobre Pandas. Si bien más adelante veremos algo de esta biblioteca, lo que nos interesa en este momento es aprender a manejar archivos directamente. Estamos trabajando con archivos CSV porque es un formato sencillo que es muy útil conocer, pero es principalmente una excusa para mostrar cómo Python maneja archivos de texto. En resumen, cuando tengas que trabajar con datos, definitivamente usá Pandas. Pero para aprender a manejar archivos vamos a seguir usando las funciones básicas de Python.

2.2 Funciones

A medida que tus programas se vuelven más largos y complejos, vas a necesitar organizarte. En esta sección vamos a introducir brevemente funciones y módulos de la biblioteca así como también la administración de errores y excepciones.

Funciones a medida

Usá funciones para encapsular código que quieras reutilizar. El siguiente ejemplo muestra una definición de una función:

```
def sumcount(n):
    """
    Devuelve la suma de los primeros n enteros
    """
    total = 0
    while n > 0:
        total += n
        n -= 1
    return total
```

Para llamar a una función:

```
a = sumcount(100)
```

Una función es una serie de instrucciones que realiza una tarea y devuelve un resultado. La palabra `return` es necesaria para explicitar el valor de retorno de la función.

Funciones de la biblioteca

Python trae una gran biblioteca estándar. Los módulos de esta biblioteca se cargan usando `import`. Por ejemplo:

```
import math
x = math.sqrt(10)

import urllib.request
u = urllib.request.urlopen('http://www.python.org/')
data = u.read()
```

Vamos a estudiar bibliotecas y módulos en detalle más adelante.

Errores y excepciones

Las funciones informan los errores como excepciones. Dado que una excepción interrumpe la ejecución de una función, la misma puede generar que todo el programa se detenga si no es administrada adecuadamente.

Probá por ejemplo lo siguiente en tu intérprete:

```
>>> int('N/A')
Traceback (most recent call last):
File "<stdin>", line 1, in <módulo>
ValueError: invalid literal for int() with base 10: 'N/A'
>>>
```

Para poder entender qué pasó (debuguear), el mensaje describe cuál fue el problema, dónde ocurrió y un poco de la historia (traceback) de los llamados que terminaron en este error.

Atrapar y administrar excepciones

Las excepciones pueden ser atrapadas y administradas. Para atrapar una excepción, se usan los comandos `try - except`.

```
numero_valido=False
while not numero_valido:
    try:
        a = input('Ingresá un número entero: ')
        n = int(a)
        numero_valido = True
    except ValueError:
        print('No es válido. Intentá de nuevo.')
print(f'Ingresaste {n}.')
```

Si en este ejemplo el usuario ingresa por ejemplo una letra, el comando `n = int(a)` genera una excepción de tipo `ValueError`: el comando `numero_valido = True` no se ejecuta, la excepción es atrapada por el `except ValueError` y el ciclo se repite. Probalo ingresando letras, números con decimales y números enteros. Probá también qué ocurre si querés salir sin ingresar nada generando una excepción presionando las teclas `Ctrl+C`. Leé el mensaje que describe lo ocurrido: `Ctrl+C` genera una excepción de tipo `KeyboardInterrupt` que no es atrapada.

Si no especificamos el tipo de excepción que queremos atrapar, vamos a terminar atrapando todas las excepciones. Probá lo mismo que antes pero con este código.

```
numero_valido=False
```

```
while not numero_valido:  
    try:  
        a = input('Ingresá un número entero: ')  
        n = int(a)  
        numero_valido = True  
    except:  
        print('No es válido. Intentá de nuevo.')  
print(f'Ingresaste {n}.')
```

Deberías observar una diferencia: al presionar las teclas `Ctrl+C` la excepción `KeyboardInterrupt` sí es atrapada y no se termina el ciclo hasta no ingresar un número entero.

Suele ser difícil saber exactamente qué tipo de errores pueden ocurrir por adelantado. Para bien o para mal, la administración de excepciones suele ir creciendo a medida que un programa va generando errores inesperados (al mejor estilo: "Uh! Me olvidé de que podía pasar esto. Deberíamos preverlo y administrarlo adecuadamente para la próxima").

Generar excepciones

Para generar una excepción (también diremos *levantar* una excepción, porque más cercano al término inglés "raise"), se usa el comando `raise`. Por ejemplo, si tenemos el siguiente código en el archivo `foo.py`:

```
raise RuntimeError(';Qué moco!')
```

Al correrlo va a detener la ejecución y permite rastrear la excepción leyendo el mensaje de error que imprime.

```
bash $ python3 foo.py  
Traceback (most recent call last):  
  File "foo.py", line 21, in <módulo>  
    raise RuntimeError(";Qué moco!")  
RuntimeError: ;Qué moco!
```

Alternativamente, esa excepción puede ser atrapada por un bloque `try-except`, pudiendo de esta forma evitar que el programa termine.

Ejercicios

Ejercicio 2.4: Definir una función

Probá primero definir una función simple:

```
>>> def saludar(nombre):
    'Saluda a alguien'
    print('Hola', nombre)

>>> saludar('Guido')
Hola Guido
>>> saludar('Paula')
Hola Paula
>>>
```

Si la primera instrucción de una función es una cadena, sirve como documentación de la función. Probalo escribiendo `help(saludar)` para ver cómo la muestra.

Ejercicio 2.5: Transformar un script en una función

Transformá el programa `costo_camion.py` (que escribiste en el [Ejercicio 2.2](#) de la sección anterior) en una función `costo_camion(nombre_archivo)`. Esta función recibe un nombre de archivo como entrada, lee la información sobre los cajones que cargó el camión y devuelve el costo de la carga de frutas como una variable de punto flotante.

Para usar tu función, cambiá el programa de forma que se parezca a esto:

```
def costo_camion(nombre_archivo):
    ...
    # Tu código
    ...

costo = costo_camion('Data/camion.csv')
print('Costo total:', costo)
```

Cuando ejecutás tu programa, deberías ver la misma salida impresa que antes. Una vez que lo hayas corrido, podés llamar interactivamente a la función escribiendo esto:

```
bash $ python3 -i costo_camion.py
```

Esto va a ejecutar el código en el programa y dejar abierto el intérprete interactivo.

```
>>> costo_camion('Data/camion.csv')
47671.15
>>>
```

Es útil para testear y debuguear poder interactuar interactivamente con tu código.

Ejercicio 2.6: Administración de errores

Probá correr la siguiente función ingresando tu edad real, una edad escrita con letras (como "ocho") y una edad negativa (-3):

```
def preguntar_edad(nombre):
    edad = int(input(f'ingresá tu edad {nombre}: '))
    if edad<0:
        raise ValueError('La edad no puede ser negativa.')
    return edad
```

Ahora probá este ejemplo que atrapa la excepción generada con `raise` y continúa la ejecución con la siguiente persona.

```
for nombre in ['Pedro', 'Juan', 'Caballero']:
    try:
        edad = preguntar_edad(nombre)
        print(f'{nombre} tiene {edad} años.')
    except ValueError:
        print(f'{nombre} no ingresó una edad válida.')
```

Vamos a usar estas ideas aplicadas al procesamiento de un archivo CSV. ¿Qué pasa si intentás usar la función `costo_camion()` con un archivo que tiene datos faltantes?

```
>>> costo_camion('Data/missing.csv')
Traceback (most recent call last):
  File "<stdin>", line 1, in <módulo>
  File "costo_camion.py", line 11, in costo_camion
    ncajones = int(fields[1])
ValueError: invalid literal for int() with base 10: ''
>>>
```

El programa termina con un error. A esta altura tenés que tomar una decisión. Para que el programa funcione podés editar el archivo CSV de entrada de manera de corregirlo (borrando líneas o adecuando la información) o podés modificar el código para que maneje las líneas *incorrectas* de alguna manera.

Modificá el programa `costo_camion.py` para que atrape la excepción, imprima un mensaje de aviso (warning) y continúe procesando el resto del archivo.

Vamos a profundizar en la administración de errores en las próximas clases.

Ejercicio 2.7: Funciones de la biblioteca

Python viene con una gran biblioteca estándar de funciones útiles. En este caso el módulo `csv` podría venirnos muy bien. Podés usarlo cada vez que tengas que leer archivos CSV. Acá va un ejemplo de cómo funciona.

```
>>> import csv
>>> f = open('Data/camion.csv')
>>> rows = csv.reader(f)
>>> headers = next(rows)
>>> headers
['nombre', 'cajones', 'precio']
>>> for row in rows:
    print(row)

['Lima', '100', '32.20']
['Naranja', '50', '91.10']
['Caqui', '150', '83.44']
['Mandarina', '200', '51.23']
['Durazno', '95', '40.37']
['Mandarina', '50', '65.10']
['Naranja', '100', '70.44']
>>> f.close()
>>>
```

Una cosa buena que tiene el módulo `csv` es que maneja solo una gran variedad de detalles de bajo nivel como el problema de las comillas, o la separación con comas de los datos. En la salida del último ejemplo podés ver que el lector ya sacó las comillas dobles de los nombres de las frutas de la primera columna.

Modificá tu programa `costo_camion.py` para que use el módulo `csv` para leer los archivos CSV y probalo en un par de los ejemplos anteriores.

Ejercicio 2.8: Ejecución desde la línea de comandos con parámetros

En el programa `costo_camion.py`, el nombre del archivo de entrada '`Data/camion.csv`' fue escrito en el código.

```
# costo_camion.py
import csv
```

```
def costo_camion(nombre_archivo):
    ...
    # Tu código
    ...

cost = costo_camion('Data/camion.csv')
print('Total cost:', cost)
```

Esto está bien para ejercitarte, pero en un programa real probablemente no harías eso ya que querrías una mayor flexibilidad. Una posibilidad es pasarle al programa el nombre del archivo que querés procesar como un parámetro cuando lo llamas desde la línea de comandos.

Copíá el contenido de `costo_camion.py` a un nuevo archivo llamado `camion_commandline.py` que incorpore la lectura de parámetros por línea de comando según la sugerencia del siguiente ejemplo:

```
# camion_commandline.py
import csv
import sys

def costo_camion(nombre_archivo):
    ...
    # Tu código
    ...

if len(sys.argv) == 2:
    nombre_archivo = sys.argv[1]
else:
    nombre_archivo = 'Data/camion.csv'

costo = costo_camion(nombre_archivo)
print('Costo total:', costo)
```

`sys.argv` es una lista que contiene los argumentos que le pasamos al script al momento de llamarlo desde la línea de comandos (si es que le pasamos alguno). Por ejemplo, desde una terminal de Unix (en Windows es similar), para correr nuestro programa y que procese el mismo archivo podríamos escribir:

```
bash $ python3 camion_commandline.py Data/camion.csv
Costo total: 47671.15
bash $
```

O con el archivo `missing.csv`:

```
bash $ python3 camion_commandline.py Data/missing.csv
```

```
...
Costo total: 30381.15
bash $
```

Si no le pasamos ningún archivo, va a mostrar el resultado para `camion.csv` porque lo indicamos con la línea `nombre_archivo = 'Data/camion.csv'`.

Guardá el archivo `camion_commandline.py` para entregar al final de la clase.

2.3 Tipos y estructuras de datos

Esta sección introduce dos estructuras de datos elementales: las tuplas y los diccionarios.

Tipos de datos primitivos

Python tiene pocos tipos primitivos de datos.

- Números enteros
- Números de punto flotante
- Cadenas de texto

Algo ya sabemos sobre estos tipos de datos por el capítulo anterior.

Tipo None

```
email_address = None
```

`None` suele utilizarse como un comodín para reservar el lugar para un valor opcional o faltante. En los condicionales, evalúa como `False`.

```
if email_address:
    send_email(email_address, msg)
```

Estructuras de datos

Los programas reales tienen datos más complejos que los que podemos almacenar en los tipos primitivos. Por ejemplo, información sobre un pedido de frutas:

```
100 cajones de Manzanas a $490.10 cada uno
```

Podemos ver esto como un "objeto" con tres partes:

- Nombre del símbolo ("Manzanas", una cadena)
- Número o cantidad (100, un entero)
- Precio (490.10 un flotante)

Tuplas

Una tupla es una colección con valores agrupados juntos.

Ejemplo:

```
s = ('Manzanas', 100, 490.1)
```

Las tuplas suelen usarse para representar registros o estructuras *simples*. Típicamente, una tupla representa un solo *objeto* con múltiples partes. Una analogía posible es la siguiente: *Una tupla es como una fila de una base de datos*.

Los contenidos de una tupla están ordenados (como en una lista).

```
s = ('Manzana', 100, 490.1)
nombre = s[0]                      # 'Manzana'
cantidad = s[1]                     # 100
precio = s[2]                       # 490.1
```

El contenido de las tuplas no puede ser modificado.

```
>>> s[1] = 75
TypeError: object does not support item assignment
```

Podés, sin embargo, hacer una nueva tupla basada en el contenido de otra, que no es lo mismo que modificar el contenido.

```
s = (s[0], 75, s[2])
```

Empaquetar tuplas

Las tuplas suelen usarse para empaquetar información relacionada en una sola *entidad*.

```
s = ('Manzanas', 100, 490.1)
```

Una tupla puede ser pasada de un lugar a otro de un programa como un solo objeto.

Desempaquetar tuplas

Para usar una tupla en otro lado, debemos desempaquetar su contenido en diferentes variables.

```
fruta, cajones, precio = s
print('Costo:', cajones * precio)
```

El número de variables a la izquierda debe ser consistente con la estructura de la tupla.

```
nombre, cajones = s      # ERROR
Traceback (most recent call last):
...
ValueError: too many values to unpack
```

Tuplas vs. Listas

Las tuplas parecieran ser listas de solo-lectura. Sin embargo, las tuplas suelen usarse para un solo ítem que consiste de múltiples partes mientras que las listas suelen usarse para una colección de diferentes elementos, típicamente del mismo tipo.

```
record = ('Manzanas', 100, 490.1)           # Una tupla representando
un registro dentro de un pedido de frutas

symbols = [ 'Manzanas', 'Peras', 'Mandarinas' ]  # Una lista representando
tres frutas diferentes.
```

Diccionarios

Un diccionario es una función que manda *claves* en *valores*. Las claves sirven como índices para acceder a los valores.

```
s = {
    'fruta': 'Manzana',
    'cajones': 100,
    'precio': 490.1
}
```

Operaciones usuales

Para obtener el valor almacenado en un diccionario usamos las claves.

```
>>> print(s['fruta'], s['cajones'])
Manzanas 100
```

```
>>> s['precio']
490.10
>>>
```

Para agregar o modificar valores, simplemente asignamos usando la clave.

```
>>> s['cajones'] = 75
>>> s['fecha'] = '6/8/2020'
>>>
```

para borrar un valor, usamos el comando `del`.

```
>>> del s['fecha']
>>>
```

¿Por qué diccionarios?

Los diccionarios son útiles cuando hay *muchos* valores diferentes y esos valores pueden ser modificados o manipulados. Dado que el acceso a los elementos se hace por *clave*, no es necesario recordar una posición para cierto dato, lo que muchas veces cumple un objetivo fundamental: hacer que el código sea más legible (y con esto menos propenso a errores).

```
s['precio'] # diccionario
# vs
s[2] # lista
```

Ejercicios

Anteriormente escribiste un programa que leía el archivo `Data/camion.csv` usando el módulo `csv` para leer el archivo fila por fila.

```
>>> import csv
>>> f = open('Data/camion.csv')
>>> filas = csv.reader(f)
>>> next(filas)
['nombre', 'cajones', 'precio']
>>> fila = next(filas)
>>> fila
['Lima', '100', '32.20']
>>>
```

A veces, además de leerlo, queremos hacer otras cosas con el archivo CSV, como por ejemplo usar los datos que contiene para hacer un cálculo. Lamentablemente una fila de datos en crudo no es suficiente para operar aritméticamente. Vamos a

querer interpretar los elementos de la fila de datos de alguna manera particular, convirtiéndolos a otro tipo de datos que resulte más adecuado para trabajar. Es frecuente además de convertir los elementos de las filas, transformar las filas enteras en tuplas o diccionarios.

Ejercicio 2.9: Tuplas

En el intérprete interactivo, creá la siguiente tupla que representa la fila de antes, pero con las columnas numéricas pasadas a formatos adecuados:

```
>>> t = (fila[0], int(fila[1]), float(fila[2]))  
>>> t  
('Lima', 100, 32.2)  
>>>
```

A partir de esta tupla, ahora podés calcular el costo total multiplicando cajones por precio:

```
>>> cost = t[1] * t[2]  
>>> cost  
3220.000000000005  
>>>
```

¿Qué pasó? ¿Qué hace ese 5 al final?

Este error no es un problema de Python, sino de la forma en la que la máquina representa los números de punto flotante. Así como en base 10 no podemos escribir un tercio de manera exacta, en base 2 escribir un quinto requiere infinitos dígitos. Al usar una representación finita (una cantidad acotada de dígitos) la máquina redondea los números. La aritmética de punto flotante no es exacta.

Esto pasa en todos los lenguajes de programación que usan punto flotante, pero en muchos casos estos pequeños errores quedan ocultos al imprimir. Por ejemplo:

```
>>> print(f'{cost:0.2f}')  
3220.00  
>>>
```

Las tuplas son de sólo lectura. Verificalo tratando de cambiar el número de cajones a 75.

```
>>> t[1] = 75  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

```
>>>
```

Aunque no podés cambiar al tupla, sí podés reemplazar la tupla por una nueva.

```
>>> t = (t[0], 75, t[2])
>>> t
('Lima', 75, 32.2)
>>>
```

Siempre que reasignes una variable como recién lo hiciste con `t`, el valor anterior de la variable se pierde. Aunque la asignación de arriba pueda parecer como que estás modificando la tupla, en realidad estás creando una nueva tupla y tirando la vieja.

Las tuplas muchas veces se usan para empaquetar y desempaquetar valores dentro de variables. Probá esto:

```
>>> nombre, cajones, precio = t
>>> nombre
'Lima'
>>> cajones
75
>>> precio
32.2
>>>
```

Tomá las variables de arriba y empaquetalas en una tupla.

```
>>> t = (nombre, 2*cajones, precio)
>>> t
('Lima', 150, 32.2)
>>>
```

Ejercicio 2.10: Diccionarios como estructuras de datos

Una alternativa a la tupla es un diccionario.

```
>>> d = {
    'nombre' : fila[0],
    'cajones' : int(fila[1]),
    'precio' : float(fila[2])
}
>>> d
{'nombre': 'Lima', 'cajones': 100, 'precio': 32.2 }
>>>
```

Calculá el costo total de este lote:

```
>>> cost = d['cajones'] * d['precio']
>>> cost
3220.000000000005
>>>
```

Compará este ejemplo con el mismo cálculo hecho con tuplas más arriba. Cambiá el número de cajones a 75.

```
>>> d['cajones'] = 75
>>> d
{'nombre': 'Lima', 'cajones': 75, 'precio': 32.2 }
>>>
```

A diferencia de las tuplas, los diccionarios se pueden modificar libremente. Agregá algunos atributos:

```
>>> d['fecha'] = (14, 8, 2020)
>>> d['cuenta'] = 12345
>>> d
{'nombre': 'Lima', 'cajones': 75, 'precio': 32.2, 'fecha': (14, 8, 2020),
 'cuenta': 12345}
>>>
```

Ejercicio 2.11: Más operaciones con diccionarios

Si usás el comando `for` para iterar sobre el diccionario, obtenés las claves:

```
>>> for k in d:
    print('k =', k)

k = nombre
k = cajones
k = precio
k = fecha
k = cuenta
>>>
```

Probá esta variante:

```
>>> for k in d:
    print(k, '=', d[k])

nombre = 'Lima'
cajones = 75
precio = 32.2
fecha = (14, 8, 2020)
cuenta = 12345
>>>
```

Una manera más elegante de trabajar con claves y valores simultáneamente es usar el método `items()`. Esto te devuelve una lista de tuplas de la forma `(clave, valor)` sobre la que podés iterar.

```
>>> items = d.items()
>>> items
dict_items([('nombre', 'Lima'), ('cajones', 75), ('precio', 32.2),
('fecha', (14, 8, 2020))])
>>> for k, v in d.items():
    print(k, '=', v)

nombre = Lima
cajones = 75
precio = 32.2
fecha = (14, 8, 2020)
>>>
```

Si pasás un diccionario a una lista, obtenés sus claves.

```
>>> list(d)
['nombre', 'cajones', 'precio', 'fecha', 'cuenta']
>>>
```

También podés obtener todas las claves del diccionario usando el método `keys()`:

```
>>> claves = d.keys()
>>> claves
dict_keys(['nombre', 'cajones', 'precio', 'fecha', 'cuenta'])
>>>
```

Si tenés tuplas como en `items` podés crear un diccionario usando la función `dict()`. Probá esto:

```
>>> nuevos_items = [('nombre', 'Manzanas'), ('cajones', 100), ('precio',
490.1), ('fecha', (13, 8, 2020))]
>>> nuevos_items
[('nombre', 'Manzanas'), ('cajones', 100), ('precio', 490.1), ('fecha',
(13, 8, 2020))]
>>> d = dict(nuevos_items)
>>> d
{'nombre': 'Manzanas', 'cajones': 100, 'precio': 490.1, 'fecha': (13, 8,
2020)}
```

2.4 Contenedores

En esta sección trataremos listas, diccionarios y conjuntos.

Panorama

Los programas suelen trabajar con muchos objetos.

- Un camión con cajones de fruta
- Una tabla de precios de cajones de fruta

En Python hay tres opciones principales para elegir.

- Listas. Datos ordenados.
- Diccionarios. Datos desordenados.
- Conjuntos. Colección desordenada de elementos únicos.

Listas como contenedores

Usá listas cuando el orden de los datos importe. Acordate de que las listas pueden contener cualquier tipo de objeto. Por ejemplo, una lista de tuplas.

```
camion = [
    ('Pera', 100, 490.1),
    ('Naranja', 50, 91.3),
    ('Limon', 150, 83.44)
]

camion[0]           # ('Pera', 100, 490.1)
camion[2]           # ('Limon', 150, 83.44)
```

Construcción de una lista

Cómo armar una lista desde cero.

```
registros = [] # Empezamos con una lista vacía

# Usamos el .append() para agregar elementos
registros.append(('Pera', 100, 490.10))
registros.append(('Naranja', 50, 91.3))
...

```

Un ejemplo de cómo cargar registros desde un archivo.

```

registros = [] # Empezamos con una lista vacía

with open('Data/camion.csv', 'rt') as f:
    next(f) # Saltar el encabezado
    for line in f:
        row = line.split(',')
        registros.append((row[0], int(row[1]), float(row[2])))

```

Diccionarios como contenedores

Los diccionarios son útiles si vamos a querer buscar rápidamente (por claves). Por ejemplo, un diccionario de precios de cajones.

```

precios = {
    'Pera': 513.25,
    'Limon': 87.22,
    'Naranja': 93.37,
    'Mandarina': 44.12
}

```

Así podemos buscar datos:

```

>>> precios['Naranja']
93.37
>>> precios['Pera']
513.25
>>>

```

Construcción de diccionarios

Ejemplo de armado de un diccionario desde cero.

```

precios = {} # Empezamos con un diccionario vacío

# Agregamos elementos
precios['Pera'] = 513.25
precios['Limon'] = 87.22
precios['Naranja'] = 93.37

```

Un ejemplo de cómo armar un diccionario a partir del contenido de un archivo.

```
precios = {} # Empezamos con un diccionario vacío
```

```

with open('Data/precios.csv', 'rt') as f:
    for line in f:
        row = line.split(',')

```

```
precios[row[0]] = float(row[1])
```

Nota: Si probás estos comandos en el archivo `Data/precios.csv`, vas a ver que casi anda. Pero hay una línea en blanco al final que genera un error. Usando lo que ya vimos, en el [Ejercicio 2.14](#) vas a tener que modificar el código para resolver el problema.

Búsquedas en un diccionario

Podés verificar si una clave existe:

```
if key in d:  
    # YES  
else:  
    # NO
```

Claves compuestas

Casi cualquier valor puede usarse como clave en un diccionario de Python. La principal restricción es que una clave debe ser de tipo inmutable. Por ejemplo, tuplas:

```
feriados = {  
    (1, 1) : 'Año nuevo',  
    (1, 5) : 'Día del trabajador',  
    (13, 9) : "Día del programador",  
}
```

Luego, podemos acceder al diccionario así:

```
>>> feriados[(1, 5)]  
'Día del trabajador'  
>>>
```

Las listas, los conjuntos y los diccionarios no pueden ser usados como claves de diccionarios, porque son mutables.

Conjuntos

Un conjunto es una colección de elementos únicos sin orden y sin repetición.

```
citricos = { 'Naranja', 'Limon', 'Mandarina' }  
# Alternativamente podemos escribirlo así:  
citricos = set(['Naranja', 'Limon', 'Mandarina'])
```

Los conjuntos son útiles para evaluar pertenencia.

```
>>> citricos
set(['Naranja', 'Limon', 'Mandarina'])
>>> 'Naranja' in citricos
True
>>> 'Manzana' in citricos
False
>>>
```

Los conjuntos también son útiles para eliminar duplicados.

```
nombres = ['Naranja', 'Manzana', 'Pera', 'Naranja', 'Pera', 'Banana']

unicos = set(nombres)
# unicos = set(['Naranja', 'Manzana', 'Pera', 'Naranja', 'Pera', 'Banana'])
```

Más operaciones en conjuntos:

```
citricos.add('Banana')           # Agregar un elemento
citricos.remove('Limon')         # Eliminar un elemento

s1 | s2                         # Unión de conjuntos s1 y s2
s1 & s2                         # Intersección de conjuntos
s1 - s2                         # Diferencia de conjuntos
```

Ejercicios

En estos ejercicios, vas a empezar a construir un programa más largo. Trabajá en el archivo `ejercicios_python/informe.py`.

Ejercicio 2.12: Lista de tuplas

El archivo `Data/camion.csv` contiene la lista de cajones cargados en un camión. En el [Ejercicio 2.5](#) de la sección anterior escribiste una función `costo_camion(nombre_archivo)` que leía el archivo y realizaba un cálculo.

La función debería verse parecida a ésta:

```
# fragmento de costo_camion.py
import csv
...

def costo_camion(nombre_archivo):
    '''Computa el precio total del camion (cajones*precio) de un archivo'''
```

```

total = 0.0

with open(nombre_archivo, 'rt') as f:
    rows = csv.reader(f)
    headers = next(rows)
    for row in rows:
        ncajones = int(row[1])
        precio = float(row[2])
        total += ncajones * precio
return total

...

```

Usando este código como guía, creá un nuevo archivo `informe.py`. En este archivo, definí una función `leer_camion(nombre_archivo)` que abre un archivo con el contenido de un camión, lo lee y devuelve la información como una lista de tuplas. Para hacerlo vas a tener que hacer algunas modificaciones menores al código de arriba.

Primero, en vez de definir `total = 0`, tenés que empezar con una variable que empieza siendo una lista vacía Por ejemplo:

```
camion = []
```

Después, en vez de sumar el costo, tenés que pasar cada fila a una tupla igual a como lo hiciste en el último ejercicio, y agregarla a la lista. Por ejemplo:

```

for row in rows:
    lote = (row[0], int(row[1]), float(row[2]))
    camion.append(lote)

```

Por último, la función debe devolver la lista `camion`.

Experimentá con tu función interactivamente (acordate de que primero tenés que correr el programa `informe.py` en el intérprete):

Ayuda: Usá `-i` para ejecutar un archivo en la terminal y quedar en el intérprete

```

>>> camion = leer_camion('Data/camion.csv')
>>> camion
[('Lima', 100, 32.2), ('Naranja', 50, 91.1), ('Limon', 150, 83.44),
('Mandarina', 200, 51.23), ('Durazno', 95, 40.37), ('Mandarina', 50, 65.1),
('Naranja', 100, 70.44)]
>>>
>>> camion[0]
('Lima', 100, 32.2)
>>> camion[1]
('Naranja', 50, 91.1)
>>> camion[1][1]

```

```

50
>>> total = 0.0
>>> for s in camion:
    total += s[1] * s[2]

>>> print(total)
47671.15
>>>

```

Esta lista de tuplas que creaste es muy similar a un array o matriz bidimensional. Por ejemplo, podés acceder a una fila específica y columna específica usando una búsqueda como `camion[fila][columna]` donde fila y columna son números enteros.

También podés reescribir el último ciclo for usando un comando como éste:

```

>>> total = 0.0
>>> for nombre, cajones, precio in camion:
    total += cajones*precio

>>> print(total)
47671.15
>>>

```

Observación: la instrucción `+=` es una abreviación. Poner `a += b` es equivalente a poner `a = a + b`

Ejercicio 2.13: Lista de diccionarios

Tomá la función que escribiste en el ejercicio anterior y modificala para representar cada cajón del camión con un diccionario en vez de una tupla. En este diccionario usá los campos "nombre", "cajones" y "precio" para representar las diferentes columnas del archivo de entrada.

Experimentá con esta función nueva igual que en el ejercicio anterior.

```

>>> camion = leer_camion('Data/camion.csv')
>>> camion
[{'nombre': 'Lima', 'cajones': 100, 'precio': 32.2}, {'nombre': 'Naranja',
'cajones': 50, 'precio': 91.1}, {'nombre': 'Limon', 'cajones': 150,
'precio': 83.44}, {'nombre': 'Mandarina', 'cajones': 200, 'precio': 51.23},
{'nombre': 'Durazno', 'cajones': 95, 'precio': 40.37}, {'nombre':
'Mandarina', 'cajones': 50, 'precio': 65.1}, {'nombre': 'Naranja',
'cajones': 100, 'precio': 70.44}]
>>> camion[0]
{'nombre': 'Lima', 'cajones': 100, 'precio': 32.2}
>>> camion[1]

```

```

{'nombre': 'Naranja', 'cajones': 50, 'precio': 91.1}
>>> camion[1]['cajones']
50
>>> total = 0.0
>>> for s in camion:
    total += s['cajones']*s['precio']

>>> print(total)
47671.15
>>>

```

Fijate que acá los distintos campos para cada entrada se acceden a través de claves en vez de la posición en la lista. Muchas veces preferimos esto porque el código resulta más fácil de leer. Tanto para otros como para nosotros en el futuro.

Mirar diccionarios y listas muy grandes puede ser un lío. Para limpiar el output para debuguear, probá la función `pprint` (Pretty-print) que le da un formato más sencillo de interpretar.

```

>>> from pprint import pprint
>>> pprint(camion)
[{'nombre': 'Lima', 'precio': 32.2, 'cajones': 100},
 {'nombre': 'Naranja', 'precio': 91.1, 'cajones': 50},
 {'nombre': 'Limon', 'precio': 83.44, 'cajones': 150},
 {'nombre': 'Mandarina', 'precio': 51.23, 'cajones': 200},
 {'nombre': 'Durazno', 'precio': 40.37, 'cajones': 95},
 {'nombre': 'Mandarina', 'precio': 65.1, 'cajones': 50},
 {'nombre': 'Naranja', 'precio': 70.44, 'cajones': 100}]
>>>

```

Ejercicio 2.14: Diccionarios como contenedores

Los diccionarios son útiles si querés buscar elementos usando índices que no sean números enteros. En la terminal de Python, jugá con un diccionario:

```

>>> precios = {}
>>> precios['Naranja'] = 92.45
>>> precios['Mandarina'] = 45.12
>>> precios
... mirá el resultado ...
>>> precios['Naranja']
92.45
>>> precios['Manzana']
... mirá el resultado ...
>>> 'Manzana' in precios
False
>>>

```

El archivo `Data/precios.csv` contiene una serie de líneas con precios de venta de cajones en el mercado al que va el camión. El archivo se ve así:

```
"Lima",9.22
"Uva",24.85
"Ciruela",44.85
"Cereza",11.27
"Frutilla",3.72
...
```

Escribí una función `leer_precios(nombre_archivo)` que a partir de un conjunto de precios como éste arme un diccionario donde las claves sean los nombres de frutas y verduras, y los valores sean los precios por cajón.

Para hacerlo, empezá con un diccionario vacío y andá agregándole valores igual que como hiciste antes, pero ahora esos valores los vas leyendo del archivo.

Vamos a usar esta estructura de datos para buscar rápidamente los precios de las frutas y verduras.

Un par de consejos: Usá el módulo `csv` igual que antes.

```
>>> import csv
>>> f = open('Data/precios.csv', 'r')
>>> rows = csv.reader(f)
>>> for row in rows:
    print(row)
```

```
['Lima', '9.22']
['Uva', '24.85']
...
[]
>>>
```

El archivo `Data/precios.csv` puede tener líneas en blanco, esto te puede traer complicaciones. Observá que arriba figura una lista vacía (la última), porque la última línea del archivo no tenía datos.

Puede suceder que esto haga que tu programa termine con una excepción. Usá los comandos `try` y `except` para evitar el problema. Para pensar: ¿Sería mejor prevenir estos problemas con el comando `if` en vez de `try` y `except`?

Una vez que hayas escrito tu función `leer_precios()`, testeala interactivamente para asegurarte de que funciona bien:

```
>>> precios = leer_precios('Data/precios.csv')
>>> precios['Naranja']
106.28
>>> precios['Mandarina']
80.89
>>>
```

Ejercicio 2.15: Balances

Supongamos que los precios en `camion.csv` son los precios pagados al productor de frutas mientras que los precios en `precios.csv` son los precios de venta en el lugar de descarga del camión.

Ahora vamos calcular el balance del negocio: juntá todo el trabajo que hiciste recién en tu programa `informe.py` (usando las funciones `leer_camion()` y `leer_precios()`) y completa el programa para que con los precios del camión ([Ejercicio 2.13](#)) y los de venta en el negocio ([Ejercicio 2.14](#)) calcule lo que costó el camión, lo que se recaudo con la venta, y la diferencia. ¿Hubo ganancia o pérdida? El programa debe imprimir por pantalla un balance con estos datos.

2.5 Secuencias

Tipo de secuencias

Python tiene tres tipos de datos que son *secuencias*.

- **String:** `'Hello'`. Una cadena es una secuencia de caracteres.
- **Lista:** `[1, 4, 5]`.
- **Tupla:** `('Pera', 100, 490.1)`.

Todas las secuencias tienen un orden, indexado por enteros, y tienen una longitud.

```
a = 'Hello'                      # String o cadena
b = [1, 4, 5]                      # Lista
c = ('Pera', 100, 490.1)          # Tupla

# Orden indexado
a[0]                                # 'H'
b[-1]                               # 5
c[1]                                # 100

# Longitud de secuencias
```

```
len(a)          # 5
len(b)          # 3
len(c)          # 3
```

Las secuencias pueden ser replicadas: `s * n`.

```
>>> a = 'Hello'
>>> a * 3
'HelloHelloHello'
>>> b = [1, 2, 3]
>>> b * 2
[1, 2, 3, 1, 2, 3]
>>>
```

Las secuencias del mismo tipo también pueden ser concatenadas: `s + t`.

```
>>> a = (1, 2, 3)
>>> b = (4, 5)
>>> a + b
(1, 2, 3, 4, 5)
>>>
>>> c = [1, 5]
>>> a + c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "list") to tuple
```

Rebanadas (slicing)

Sacar una rebanada es tomar una subsecuencia de una secuencia. La sintaxis es `s[comienzo:fin]`, donde `comienzo` y `fin` son los índices de la subsecuencia que querés.

```
a = [0,1,2,3,4,5,6,7,8]

a[2:5]    # [2,3,4]
a[-5:]    # [4,5,6,7,8]
a[:3]     # [0,1,2]
```

- Los índices `comienzo` y `fin` deben ser enteros.
- Las rebanadas *no* incluyen el valor final. Es como los intervalos semi-abiertos en matemática.
- Si los índices son omitidos toman sus valores por defecto: el principio o el final de la lista.

Reasigación de rebanadas

En listas, una rebanada puede ser reasignada o eliminada.

```
# Reasignación
a = [0,1,2,3,4,5,6,7,8]
a[2:4] = [10,11,12]      # [0,1,10,11,12,4,5,6,7,8]
```

Observación: La rebanada reasignada no tiene que tener necesariamente la misma longitud.

```
# Eliminación
a = [0,1,2,3,4,5,6,7,8]
del a[2:4]           # [0,1,4,5,6,7,8]
```

Reducciones de secuencias

Hay algunas operaciones usuales que reducen una secuencia a un solo valor.

```
>>> s = [1, 2, 3, 4]
>>> sum(s)
10
>>> min(s)
1
>>> max(s)
4
>>> t = ['Hello', 'World']
>>> max(t)
'World'
>>>
```

Iterar sobre una secuencia

Los ciclos `for` iteran sobre los elementos de una secuencia.

```
>>> s = [1, 4, 9, 16]
>>> for i in s:
...     print(i)
...
1
4
9
16
>>>
```

En cada iteración del ciclo obtenés un nuevo elemento para trabajar. La variable iteradora va a tomar este nuevo valor. En el siguiente ejemplo la variable iteradora es `x`:

```
for x in s:      # `x` es una variable iteradora
    ...instrucciones
```

En cada iteración, el valor previo de la variable (si hubo alguno) es sobreescrito. Luego de terminar el ciclo, la variable retiene su último valor.

El comando break

Podés usar el comando `break` para romper un ciclo antes de tiempo.

```
for name in namelist:
    if name == 'Juana':
        break
    ...
    ...
instrucciones
```

Cuando el comando `break` se ejecuta, sale del ciclo y se mueve a las siguientes instrucciones. El comando `break` sólo se aplica al ciclo más interno. Si un ciclo está anidado en otro ciclo, el comando no va a romper el ciclo externo.

El comando continue

Para saltar un elemento y moverse al siguiente, usá el comando `continue`.

```
for line in lines:
    if line == '\n':    # Salteo las líneas en blanco
        continue
    # Más instrucciones
    ...
```

Ésta es útil cuando el elemento actual no es de interés o es necesario ignorarlo en el procesamiento.

Ciclos sobre enteros

Para iterar sobre un rango de números enteros, usá `range()`.

```
for i in range(100):
    # i = 0,1,...,99
```

La sintaxis es `range([comienzo,] fin [,paso])` (lo que figura entre corchetes es opcional).

```
for i in range(100):
    # i = 0,1,...,99
    ...codigo
for j in range(10,20):
    # j = 10,11,..., 19
    ...codigo
for k in range(10,50,2):
    # k = 10,12,...,48
    # Observá que va de a dos.
    ...codigo
```

- El valor final nunca es incluido. Es como con las rebanadas.
- `comienzo` es opcional. Por defecto es `0`.
- `paso` es opcional. Por defecto es `1`.
- `range()` calcula los valores a medida que los necesita. No guarda realmente en memoria el rango completo de números.

La función enumerate()

La función `enumerate` agrega un contador extra a una iteración.

```
nombres = ['Edmundo', 'Juana', 'Rosita']
for i, nombre in enumerate(nombres):
    # i = 0, nombre = 'Edmundo'
    # i = 1, nombre = 'Juana'
    # i = 2, nombre = 'Rosita'
```

La forma general es `enumerate(secuencia [, start = 0])`. `start` es opcional. Un buen ejemplo de cuándo usar `enumerate()` es para llevar la cuenta del número de línea mientras estás leyendo un archivo:

```
with open(nombre_archivo) as f:
    for nlinea, linea in enumerate(f, start=1):
        ...
```

Al fin de cuentas, `enumerate` es sólo una forma abreviada y simpática de escribir:

```
i = 0
for x in s:
    instrucciones
    i += 1
```

Al usar `enumerate` tenemos que tipar menos y el programa funciona un toque más rápido.

Tuplas y ciclos for

Podés iterar con múltiples variables de iteración.

```
points = [
    (1, 4), (10, 40), (23, 14), (5, 6), (7, 8)
]
for x, y in points:
    #     x = 1, y = 4
    #     x = 10, y = 40
    #     x = 23, y = 14
    #     ...
```

Cuando usás múltiples variables, cada tupla es *desempaquetada* en un conjunto de variables de iteración. El número de variables debe coincidir con la cantidad de elementos de cada tupla.

La función zip()

La función `zip` toma múltiples secuencias y las combina en un iterador.

```
columnas = ['nombre', 'cajones', 'precio']
valores = ['Pera', 100, 490.1]
pares = zip(columnas, valores)
# ('nombre', 'Pera'), ('cajones', 100), ('precio', 490.1)
```

Para obtener el resultado debés iterar. Podés usar múltiples variables para desempaquetar las tuplas como mostramos antes.

```
for columna, valor in pares:
    ...
```

Un uso frecuente de `zip` es para crear pares clave/valor y construir diccionarios.

```
d = dict(zip(columnas, valores))
```

Ejercicios

Ejercicio 2.16: Contar

Probá algunos ejemplos elementales de conteo:

```
>>> for n in range(10):           # Contar 0 ... 9
    print(n, end=' ')
0 1 2 3 4 5 6 7 8 9
>>> for n in range(10,0,-1):      # Contar 10 ... 1
    print(n, end=' ')
10 9 8 7 6 5 4 3 2 1
>>> for n in range(0,10,2):       # Contar 0, 2, ... 8
    print(n, end=' ')
0 2 4 6 8
>>>
```

Ejercicio 2.17: Más operaciones con secuencias

Interactivamente experimentá con algunas operaciones de reducción de secuencias.

```
>>> data = [4, 9, 1, 25, 16, 100, 49]
>>> min(data)
1
>>> max(data)
100
>>> sum(data)
204
>>>
```

Probá iterar sobre los datos.

```
>>> for x in data:
    print(x)

4
9
...
>>> for n, x in enumerate(data):
    print(n, x)

0 4
1 9
2 1
```

```
...  
>>>
```

A veces los comandos `for`, `len()`, y `range()` son combinados para recorrer listas:

```
>>> for n in range(len(data)):  
    print(data[n])
```

```
4  
9  
1  
...  
>>>
```

Sin embargo, Python tiene mejores alternativas para esto. Te recomendamos familiarizarte con ellas y usarlas: por su simpleza producen código más legible y reducen la posibilidad de un bug en el código. Simplemente usa un ciclo `for` normal si querés iterar sobre los elementos de la variable `data`. Y usá `enumerate()` si necesitás tener el índice por algún motivo.

Ejercicio 2.18: Un ejemplo práctico de `enumerate()`

Recordá que el archivo `Data/missing.csv` contiene datos sobre los cajones de un camión, pero tiene algunas filas que faltan. Usando `enumerate()`, modifícá tu programa `costo_camion.py` de forma que imprima un aviso (warning) cada vez que encuentre una fila incorrecta.

```
>>> cost = costo_camion('Data/missing.csv')  
Fila 4: No pude interpretar: ['Mandarina', '', '51.23']  
Fila 7: No pude interpretar: ['Naranja', '', '70.44']  
>>>
```

Para hacer esto, vas a tener que cambiar algunas partes de tu código.

```
...  
for n_fila, fila in enumerate(filas, start=1):  
    try:  
        ...  
    except ValueError:  
        print(f'Fila {n_fila}: No pude interpretar: {fila}')
```

Ejercicio 2.19: La función zip()

En el archivo `Data/camion.csv`, la primera línea tiene los encabezados de las columnas. En los códigos anteriores la descartamos.

```
>>> f = open('Data/camion.csv')
>>> filas = csv.reader(f)
>>> encabezados = next(filas)
>>> encabezados
['nombre', 'cajones', 'precio']
>>>
```

Pero, ¿no puede ser útil conocer los encabezados? Es acá donde la función `zip()` entra en acción. Primero tratá de aparear los encabezados con una fila de datos:

```
>>> fila = next(filas)
>>> fila
['Lima', '100', '32.20']
>>> list(zip(encabezados, fila))
[('nombre', 'Lima'), ('cajones', '100'), ('precio', '32.20')]
>>>
```

Fijate cómo `zip()` apareó los encabezados de las columnas con los valores de la columna. Usamos `list()` arriba para devolver el resultado en una lista de forma que lo puedas ver. Normalmente, `zip()` crea un iterador que debe ser consumido en un ciclo `for`.

Este apareamiento es un paso intermedio para construir un diccionario. Probá lo siguiente:

```
>>> record = dict(zip(encabezados, fila))
>>> record
{'precio': '32.20', 'nombre': 'Lima', 'cajones': '100'}
>>>
```

Esta transformación es un truco sumamente útil cuando tenés que procesar muchos archivos de datos. Por ejemplo, suponé que querés hacer que el programa `costo_camion.py` trabaje con diferentes archivos de entrada, pero que no le importe la posición exacta de la columna que tiene la cantidad de cajones o el precio. Es decir, que entienda que la columna tiene el precio por su encabezado y no por su posición dentro del archivo.

Modificá la función `costo_camion()` en el archivo `costo_camion.py` para que se vea así:

```
# costo_camion.py

def costo_camion(nombre_archivo):
    ...
    for n_fila, fila in enumerate(filas, start=1):
        record = dict(zip(encabezados, fila))
        try:
            ncajones = int(record['cajones'])
            precio = float(record['precio'])
            costo_total += ncajones * precio
        # Esto atrapa errores en los int() y float() de arriba.
        except ValueError:
            print(f'Fila {n_fila}: No pude interpretar: {fila}')
    ...


```

Ahora, probá tu función con un archivo completamente diferente Data/fecha_camion.csv que se ve así:

```
nombre,fecha,hora,cajones,precio
"Lima","6/11/2007","9:50am",100,32.20
"Naranja","5/13/2007","4:20pm",50,91.10
"Caqui","9/23/2006","1:30pm",150,83.44
"Mandarina","5/17/2007","10:30am",200,51.23
"Durazno","2/1/2006","10:45am",95,40.37
"Mandarina","10/31/2006","12:05pm",50,65.10
"Naranja","7/9/2006","3:15pm",100,70.44
```

```
>>> costo_camion('Data/fecha_camion.csv')
47671.15
>>>
```

Si lo hiciste bien, vas a descubrir que tu programa aún funciona a pesar de que le pasaste un archivo con un formato de columnas completamente diferente al de antes. ¡Y eso está muy bueno!

El cambio que hicimos acá es sutil, pero importante. En lugar de tener *hardcodeado* un formato fijo, la nueva versión de la función `costo_camion()` puede sacar la información de interés de cualquier archivo CSV. En la medida en que el archivo tenga las columnas requeridas, el código va a funcionar.

Modificá el programa `informe.py` que escribiste antes (ver [Ejercicio 2.15](#)) para que use esta técnica para elegir las columnas a partir de sus encabezados.

Probá correr el programa `informe.py` sobre el archivo Data/fecha_camion.csv y fijate si da la misma salida que antes.

Ejercicio 2.20: Invertir un diccionario

Un diccionario es una función que mapea claves en valores. Por ejemplo, un diccionario de precios de cajones de frutas.

```
>>> precios = {
    'Pera' : 490.1,
    'Lima' : 23.45,
    'Naranja' : 91.1,
    'Mandarina' : 34.23
}
>>>
```

Si usás el método `items()`, obtenés pares `(clave, valor)`:

```
>>> precios.items()
dict_items([('Pera', 490.1), ('Lima', 23.45), ('Naranja', 91.1),
('Mandarina', 34.23)])
>>>
```

Sin embargo, si lo que querés son pares `(valor, clave)`, ¿cómo lo hacés? Ayuda: usá `zip()`.

```
>>> lista_precios = list(zip(precios.values(), precios.keys()))
>>> lista_precios
[(490.1, 'Pera'), (23.45, 'Lima'), (91.1, 'Naranja'), (34.23, 'Mandarina')]
>>>
```

¿Por qué haría algo así? Por un lado porque te permite realizar cierto tipo de procesamiento de datos sobre la información del diccionario.

```
>>> min(lista_precios)
(23.45, 'Lima')
>>> max(lista_precios)
(490.1, 'Pera')
>>> sorted(lista_precios)
[(23.45, 'Lima'), (34.23, 'Mandarina'), (91.1, 'Naranja'), (490.1, 'Pera')]
>>>
```

Esto también ilustra un atributo importante de las tuplas. Cuando son usadas en una comparación, las tuplas son comparadas elemento-a-elemento comenzando con el primero. Es similar a la lógica subyacente al orden lexicográfico o alfabético en las cadenas.

La función `zip()` se usa frecuentemente en este tipo de situaciones donde necesitás aparear datos provenientes de diferentes lugares. Por ejemplo, para aparear los

nombres de las columnas con los valores para hacer un diccionario de valores con nombres.

Observá que `zip()` no está limitada a pares. Podés usarla con cualquier número de listas de entrada:

```
>>> a = [1, 2, 3, 4]
>>> b = ['w', 'x', 'y', 'z']
>>> c = [0.2, 0.4, 0.6, 0.8]
>>> list(zip(a, b, c))
[(1, 'w', 0.2), (2, 'x', 0.4), (3, 'y', 0.6), (4, 'z', 0.8))]
```

También, tené en cuenta que `zip()` se detiene cuando la más corta de las entradas se agota.

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> b = ['x', 'y', 'z']
>>> list(zip(a,b))
[(1, 'x'), (2, 'y'), (3, 'z'))]
```

2.6 Contadores del módulo *collections*

El módulo `collections` ofrece objetos útiles para manejar datos. En esta sección introducimos brevemente los contadores, que son solo una de las clases incluidas en este módulo.

Ejemplo: Contar cosas

Digamos que querés hacer una tabla con el total de cajones de cada fruta.

```
camion = [
    ('Pera', 100, 490.1),
    ('Naranja', 50, 91.1),
    ('Caqui', 150, 83.44),
    ('Naranja', 100, 45.23),
    ('Pera', 75, 572.45),
    ('Lima', 50, 23.15)
]
```

Hay dos entradas de Naranja y dos de Pera en esta lista. Estos cajones deben ser combinados juntos de alguna forma.

Contadores

Solución: Usá un Counter (contador).

```
from collections import Counter
total_cajones = Counter()
for nombre, n_cajones, precio in camion:
    total_cajones[nombre] += n_cajones

total_cajones['Naranja'] # 150
```

Ejercicios

En este ejercicio vas a probar contadores en un par de ejemplos simples. Cargá tu programa informe.py y ejecutalo en el interprete de forma de tener los datos del camión con cajones cargado en modo interactivo.

Podés usar el interprete desde la línea de comandos ejecutando:

```
bash % python3 -i informe.py
```

O podés cargarlo en el Spyder y correrlo.

Ejercicio 2.21: Contadores

Vamos a usar un contador (objeto Counter) para contar cajones de frutas. Probalo:

```
>>> camion = leer_camion('Data/camion.csv')
>>> from collections import Counter
>>> tenencias = Counter()
>>> for s in camion:
    tenencias[s['nombre']] += s['cajones']

>>> tenencias
Counter({'Caqui': 150, 'Durazno': 95, 'Lima': 100, 'Mandarina': 250,
'Naranja': 150})
>>>
```

Observá que las entradas múltiples como Mandarina y Naranja en camion se combinan en una sola entrada.

Podés usar el contador como un diccionario para recuperar valores individuales:

```
>>> tenencias['Naranja']
150
>>> tenencias['Mandarina']
250
>>>
```

Podés listar las tres frutas con mayores tenencias:

```
>>> # Las 3 frutas con más cajones
>>> tenencias.most_common(3)
[('Mandarina', 250), ('Naranja', 150), ('Caqui', 150)]
>>>
```

Carguemos los datos de otro camión con cajones de fruta en un nuevo contador:

```
>>> camion2 = leer_camion('Data/camion2.csv')
>>> tenencias2 = Counter()
>>> for s in camion2:
...     tenencias2[s['nombre']] += s['cajones']

>>> tenencias2
Counter({'Durazno': 125, 'Frambuesa': 250, 'Lima': 50, 'Mandarina': 25})
>>>
```

Y finalmente combinemos las tenencias de ambos camiones con una operación simple:

```
>>> tenencias
Counter({'Caqui': 150, 'Durazno': 95, 'Lima': 100, 'Mandarina': 250,
'Naranja': 150})
>>> tenencias2
Counter({'Frambuesa': 250, 'Durazno': 125, 'Lima': 50, 'Mandarina': 25})
>>> combinada = tenencias + tenencias2
>>> combinada
Counter({'Caqui': 150, 'Durazno': 220, 'Frambuesa': 250, 'Lima': 150,
'Mandarina': 275, 'Naranja': 150})
>>>
```

Esto es solo una pequeña muestra de lo que se puede hacer con contadores. El módulo `collections` es muy poderoso pero meterse a ver sus detalles sería una distracción ahora. Sigamos con nuestro curso...

2.7 Arbolado porteño

En esta sección haremos algunos ejercicios que integran los conceptos aprendidos en las clases anteriores. Vamos a manejar archivos, diccionarios, listas, contadores y el comando `zip`, entre otras cosas. Entregá lo que puedas hacer.

Ejercicios

Vamos a repasar las herramientas que vimos en esta clase aplicándolas a una base de datos sobre árboles en parques de la Ciudad de Buenos Aires. Para empezar, descargá el archivo CSV de "[Arbolado en espacios verdes](#)" en tu carpeta `Data`. Vamos a estudiar esta base de datos y responder algunas preguntas. Guardá los ejercicios de esta sección en un archivo `árboles.py`.



Descripción de la base

Título de la columna	Tipo de dato	Descripción
long	Número flotante (float)	Coordenadas para geolocalización
lat	Número flotante (float)	Coordenadas para geolocalización
id_arbol	Número entero (integer)	Identificador único del árbol
altura_tot	Número entero (integer)	Altura del árbol (m)
diametro	Número entero (integer)	Diámetro del árbol (cm)
inclinacio	Número entero (integer)	Inclinación del árbol (grados)
id_especie	Número entero (integer)	Identificador de la especie
nombre_com	Texto (string)	Nombre común del árbol
nombre_cie	Texto (string)	Nombre científico del árbol
tipo_folla	Texto (string)	Tipo de follaje del árbol
espacio_ve	Texto (string)	Nombre del espacio verde
ubicacion	Texto (string)	Dirección del espacio verde

nombre_fam	Texto (string)	Nombre de la familia del árbol
nombre_gen	Texto (string)	Nombre del género del árbol
origen	Texto (string)	Origen del árbol
coord_x	Número flotante (float)	Coordenadas para localización
coord_y	Número flotante (float)	Coordenadas para localización

Ejercicio 2.22: Lectura de los árboles de un parque

Definí una función `leer_parque(nombre_archivo, parque)` que abra el archivo indicado y devuelva una lista de diccionarios con la información del parque especificado. La función debe devolver, en una lista un diccionario con todos los datos por cada árbol del parque elegido (recordá que cada fila del csv es un árbol).

Sugerencia: basate en la función `leer_camion()` y usá también el comando `zip` como hiciste en el Ejercicio 2.19 para combinar el encabezado del archivo con los datos de cada fila. Inicialmente no te preocupes por los tipos de datos de cada columna, pero cuando empieces a operar con una columna modifica esta función para que ese dato sea del tipo adecuado para operar.

Observación: La columna que indica el nombre del parque en el que se encuentra el árbol se llama '`espacio_ve`' en el archivo CSV.

Probá con el parque "GENERAL PAZ" para tener un ejemplo de trabajo, debería darte una lista con 690 árboles.

Ejercicio 2.23: Determinar las especies en un parque

Escribí una función `especies(lista_arboles)` que tome una lista de árboles como la generada en el ejercicio anterior y devuelva el conjunto de especies (la columna '`nombre_com`' del archivo) que figuran en la lista.

Sugerencia: Usá el comando `set` como en la Sección 2.4.

Ejercicio 2.24: Contar ejemplares por especie

Usando contadores como en el [Ejercicio 2.21](#), escribí una función `contar_ejemplares(lista_arboles)` que, dada una lista como la que generada con `leer_parque()`, devuelva un diccionario en el que las especies (recordá, es la columna `'nombre_com'` del archivo) sean las claves y tengan como valores asociados la cantidad de ejemplares en esa especie en la lista dada.

Luego, combiná esta función con `leer_parque()` y con el método `most_common()` para informar las cinco especies más frecuentes en cada uno de los siguientes parques:

- 'GENERAL PAZ'
- 'ANDES, LOS'
- 'CENTENARIO'

Resultados de cantidad por especie en tres parques:

General Paz	Los Andes	Centenario
Casuarina: 97	Jacarandá: 117	Plátano: 137
Tipa blanca: 54	Tipa blanca: 28	Jacarandá: 45
Eucalipto: 49	Ciprés: 21	Tipa blanca: 42
Palo borracho rosado: 44	Palo borracho rosado: 18	Palo borracho rosado: 41
Fenix: 40	Lapacho: 12	Fresno americano: 38

Ejercicio 2.25: Alturas de una especie en una lista

Escribí una función `obtener_alturas(lista_arboles, especie)` que, dada una lista de árboles como la anterior y una especie de árbol (un valor de la columna `'nombre_com'` del archivo), devuelva una lista con las alturas (columna `'altura_tot'`) de los ejemplares de esa especie en la lista.

Observación: Acá sí, fijate de devolver las alturas como números (de punto flotante) y no como cadenas de caracteres. Podés hacer esto modificando `leer_parque`.

Usala para calcular la altura promedio y altura máxima de los 'Jacarandá' en los tres parques mencionados.

Resultados de alturas de Jacarandás en tres parques:

Medida	General Paz	Los Andes	Centenario
max	16.0	25.0	18.0
prom	10.2	10.54	8.96

Ejercicio 2.26: Inclinación promedio por especie de una lista

Escribí una función `obtener_inclinaciones(lista_arboles, especie)` que, dada una especie de árbol y una lista de árboles como la anterior, devuelva una lista con las inclinaciones (columna '`inclinacion`') de los ejemplares de esa especie.

Ejercicio 2.27: Especie con el ejemplar más inclinado

Combinando la función `especies()` con `obtener_inclinaciones()` escribí una función `especimen_mas_inclinado(lista_arboles)` que, dada una lista de árboles devuelva la especie que tiene el ejemplar más inclinado y su inclinación.

Correlo para los tres parques mencionados anteriormente.

Resultados. Deberías obtener, por ejemplo, que en el Parque Centenario hay un *Falso Guayabo* inclinado 80 grados.

Ejercicio 2.28: Especie con más inclinada en promedio

Volvé a combinar las funciones anteriores para escribir la función `especie_promedio_mas_inclinada(lista_arboles)` que, dada una lista de árboles devuelva la especie que en promedio tiene la mayor inclinación y el promedio calculado..

Resultados. Deberías obtener, por ejemplo, que los *Álamos Plateados* del Parque Los Andes tiene un promedio de inclinación de 25 grados.

Preguntas extras: ¿Qué habría que cambiar para obtener la especie con un ejemplar más inclinado de toda la ciudad y no solo de un parque? ¿Podrías dar la latitud y longitud de ese ejemplar? ¿Y dónde se encuentra (lat,lon) el ejemplar más alto? ¿De qué especie es?

2.8 Impresión con formato

En esta sección se ven detalles técnicos sobre cómo hacer que la salida por pantalla sea más amena para el usuario. No es indispensable para el curso. Si te alcanza el tiempo está semana leela, sino no te preocupes podés volver a mirar acá en el futuro, cuando lo necesites.

Cuando trabajás con datos es usual que quieras imprimir salidas estructuradas (tablas, etc.). Por ejemplo:

Nombre	Cajones	Precio
Lima	100	32.20
Naranja	50	91.10
Caqui	150	103.44
Mandarina	200	51.23
Durazno	95	40.37
Mandarina	50	65.10
Naranja	100	70.44

Formato de cadenas

Una excelente manera de darle formato a una cadena en Python (a partir de la versión 3.6) es usando *f-strings*.

```
>>> nombre = 'Naranja'
>>> cajones = 100
>>> precio = 91.1
>>> f'{nombre:>10s} {cajones:>10d} {precio:>10.2f}'
'      Naranja      100      91.10'
>>>
```

La parte `{expresion:formato}` va a ser reemplazada. Usualmente los `f-strings` se usan con `print`.

```
print(f'{nombre:>10s} {cajones:>10d} {precio:>10.2f}')
```

Códigos de formato

Los códigos de formato (lo que va luego de `:` dentro de `{}`) son similares a los que se usan en el `printf()` del lenguaje C. Los más comunes son:

d	Entero decimal
b	Entero binario
x	Entero hexadecimal
f	Flotante como <code>[-]m.######</code>
e	Flotante como <code>[-]m.#####e+/-xx</code>
g	Flotante, pero con uso selectivo de la notación exponencial E.
s	Cadenas
c	Carácter (a partir de un entero, su código)

Los modificadores permiten ajustar el ancho a imprimir o la precisión decimal (cantidad de dígitos luego del punto). Ésta es una lista parcial:

<code>:>10d</code>	Entero alineado a la derecha en un campo de 10 caracteres
<code>:<10d</code>	Entero alineado a la izquierda en un campo de 10 caracteres
<code>:^10d</code>	Entero centrado en un campo de 10 caracteres
<code>:0.2f</code>	Flotante con dos dígitos de precisión

Formato a diccionarios

Pores usar el método `format_map()` para aplicarle un formato a los valores de un diccionario:

```
>>> s = {
    'nombre': 'Naranja',
    'cajones': 100,
    'precio': 91.1
}
>>> '{nombre:>10s} {cajones:>10d} {precio:>10.2f}'.format_map(s)
'      Naranja      100      91.10'
>>>
```

Usa los mismos códigos que los `f-strings` pero toma los valores que provee el diccionario.

El método format()

Existe un método `format()` que permite aplicar formato a argumentos.

```
>>> '{nombre:>10s} {cajones:10d} {precio:10.2f}'.format(nombre='Naranja',  
cajones=100, precio=91.1)  
'      Naranja      100      91.10'  
>>> '{:10s} {:10d} {:10.2f}'.format('Naranja', 100, 91.1)  
'      Naranja      100      91.10'  
>>>
```

La verdad es que `format()` nos resulta un poco extenso y preferimos usar f-strings.

Formato estilo C

También podés usar el operador `%`.

```
>>> 'The value is %d' % 3  
'The value is 3'  
>>> '%5d %-5d %10d' % (3, 4, 5)  
'      3 4          5 '  
>>> '%0.2f' % (3.1415926,)  
'3.14'
```

Esto requiere un solo ítem, o una tupla a la derecha. Los códigos están tambien inspirados en el `printf()` de C. Tiene la dificultad de que hay que contar posiciones y todas las variables van juntas al final.

Ejercicios

Ejercicio 2.29: Formato de números

Un problema usual cuando queremos imprimir números es especificar el número de dígitos decimales. Los f-strings nos permiten hacerlo. Probá los siguientes ejemplos:

```
>>> value = 42863.1  
>>> print(value)  
42863.1  
>>> print(f'{value:0.4f}')  
42863.1000  
>>> print(f'{value:>16.2f}')  
        42863.10  
>>> print(f'{value:<16.2f}')  
42863.10
```

```
>>> print(f'{value:>16,.2f}')
*****
42,863.10
>>>
```

La documentación completa sobre los códigos de formato usados en f-strings puede consultarse [acá](#). El formato puede aplicarse también usando el operador `%` de cadenas.

```
>>> print('%0.4f' % value)
42863.1000
>>> print('%16.2f' % value)
        42863.10
>>>
```

La documentación sobre códigos usados con `%` puede encontrarse [acá](#).

A pesar de que suelen usarse dentro de un `print`, el formato de cadenas no está necesariamente ligado a la impresión. Por ejemplo, podés simplemente asignarlo a una variable.

```
>>> f = '%0.4f' % value
>>> f
'42863.1000'
>>>
```

Ejercicio 2.30: Recolectar datos

En el [Ejercicio 2.15](#), escribiste un programa llamado `informe.py` que calculaba las ganancias o pérdidas de un camión que compra a productores y vende en el mercado. Dejá ese archivo para entregar al final de la clase y copiá su contenido en un archivo `tabla_informe.py`. En este ejercicio, vas a comenzar a modificarlo para producir una tabla como ésta:

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

En este informe, el "Precio" es el precio en el mercado y el "Cambio" es la variación respecto al precio cobrado por el productor.

Para generar un informe como el de arriba, primero tenés que recolectar todos los datos de la tabla. Escribí una función `hacer_informe()` que recibe una lista de cajones y un diccionario con precios como `input` y devuelve una lista de tuplas conteniendo la información mostrada en la tabla anterior.

Agregá esta función a tu archivo `tabla_informe.py`. Debería funcionar como se muestra en el siguiente ejemplo:

```
>>> camion = leer_camion('Data/camion.csv')
>>> precios = leer_precios('Data/precios.csv')
>>> informe = hacer_informe(camion, precios)
>>> for r in informe:
    print(r)

('Lima', 100, 32.2, 8.019999999999996)
('Naranja', 50, 91.1, 15.180000000000007)
('Caqui', 150, 103.44, 2.019999999999996)
('Mandarina', 200, 51.23, 29.660000000000004)
('Durazno', 95, 40.37, 33.110000000000001)
('Mandarina', 50, 65.1, 15.790000000000006)
('Naranja', 100, 70.44, 35.84)
...
>>>
```

Ejercicio 2.31: Imprimir una tabla con formato

Volvé a hacer el ciclo `for` del ejercicio anterior pero cambiando la forma de imprimir como sigue:

```
>>> for r in informe:
    print('%10s %10d %10.2f %10.2f' % r)

    Lima      100     32.20     8.02
    Naranja     50     91.10    15.18
    Caqui     150    103.44     2.02
    Mandarina   200     51.23    29.66
    Durazno      95     40.37    33.11
    Mandarina    50     65.10    15.79
    Naranja     100    70.44    35.84
...
>>>
```

O directamente usando f-strings. Por ejemplo:

```
>>> for nombre, cajones, precio, cambio in informe:
```

```

        print(f'{nombre:>10s} {cajones:>10d} {precio:>10.2f}
{cambio:>10.2f} ')

```

	Lima	100	32.20	8.02
Naranja	50	91.10	15.18	
Caqui	150	103.44	2.02	
Mandarina	200	51.23	29.66	
Durazno	95	40.37	33.11	
Mandarina	50	65.10	15.79	
Naranja	100	70.44	35.84	
...				

>>>

Agregá estos últimos comandos a tu programa `tabla_informe.py`. Hacé que el programa tome la salida de la función `hacer_informe()` e imprima una tabla bien formateada.

Ejercicio 2.32: Agregar encabezados

Suponete que tenés una tupla con nombres de encabezado como ésta:

```
headers = ('Nombre', 'Cajones', 'Precio', 'Cambio')
```

Agregá el código necesario a tu programa para que tome una tupla de encabezados como la de arriba y cree una cadena donde cada nombre de encabezado esté alineado a la derecha en un campo de 10 caracteres de ancho y separados por un solo espacio.

```
'    Nombre    Cajones    Precio    Cambio'
```

Escribí el código que recibe los encabezados y crea una cadena de separación entre los encabezados y los datos que siguen. Esta cadena es simplemente una tira de caracteres "-" bajo cada nombre de campo. Por ejemplo:

```
'----- ----- ----- -----'
```

Cuando esté listo, tu programa debería producir una tabla como esta:

Nombre	Cajones	Precio	Cambio
Lima	100	32.20	8.02
Naranja	50	91.10	15.18
Caqui	150	103.44	2.02
Mandarina	200	51.23	29.66
Durazno	95	40.37	33.11
Mandarina	50	65.10	15.79
Naranja	100	70.44	35.84

Ejercicio 2.33: Un desafío de formato

Por último, modifícá tu código para que el precio mostrado incluya un símbolo de pesos (\$) y la salida se vea como esta tabla:

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

Guardá estos cambios en el archivo `tabla_informe.py` que más adelante los vas a necesitar.

Ejercicio 2.34: Tablas de multiplicar

Escribí un programa `tablamult.py` que imprima de forma prolífica las tablas de multiplicar del 1 al 9 usando f-strings. Si podés, evitá usar la multiplicación, usando sólo sumas alcanza.

	0	1	2	3	4	5	6	7	8	9
0:	0	0	0	0	0	0	0	0	0	0
1:	0	1	2	3	4	5	6	7	8	9
2:	0	2	4	6	8	10	12	14	16	18
3:	0	3	6	9	12	15	18	21	24	27
4:	0	4	8	12	16	20	24	28	32	36
5:	0	5	10	15	20	25	30	35	40	45
6:	0	6	12	18	24	30	36	42	48	54
7:	0	7	14	21	28	35	42	49	56	63
8:	0	8	16	24	32	40	48	56	64	72
9:	0	9	18	27	36	45	54	63	72	81

2.9 Cierre de la segunda clase

En esta segunda clase aprendimos a trabajar con datos. Manejamos archivos CSV y estructuras un poco más complejas como tuplas, conjuntos y diccionarios y profundizamos un poco más en las listas.

Te recordamos que leas el [código de honor](#) del curso en el que hablamos de las reglas que rigen en este curso para evitar el plagio así como otros aspectos importantes sobre qué se puede compartir y qué no. Al enviar tus archivos entendemos que leíste y estas de acuerdo con este texto. En caso contrario no envíes tus archivos y contactate con los docentes.

Para cerrar esta clase te pedimos dos cosas:

- Que recopiles las soluciones de los siguientes ejercicios:
 - i. El archivo `camion_commandline.py` del [Ejercicio 2.8](#).
 - ii. El archivo `informe.py` del [Ejercicio 2.19](#).
 - iii. El archivo `arboles.py` sobre arbolado porteño incluyendo todos los ejercicios que hayas hecho (esperamos al menos el [Ejercicio 2.22](#) y uno más).
 - iv. El archivo `tabla_informe.py` del [Ejercicio 2.33](#) (o del [Ejercicio 2.32](#) si no salió lo del signo \$).
 - v. El archivo `tablamult.py` del [Ejercicio 2.34](#).
- Que completes [este formulario](#) usando como identificación tu dirección de mail. Al terminar vas a obtener un link para enviarnos tus ejercicios y tendrás la opción de participar en la revisión de pares.

Esperamos que entregues como mínimo tres archivos de los cinco o más que te pedimos.

Acordate, usá siempre la misma dirección de mail con la que te inscribiste al curso así podemos llevar registro de tus entregas.

Observación: Si el enunciado de un ejercicio te pide que lo corras con un input particular, por favor poné la salida que obtuviste como comentario en tu código.

Por último te recordamos que si te quedaron dudas, querés discutir algún tema de interés o pedirnos a los docentes que resolvamos un ejercicio particular para la próxima clase, podés hacerlo en el [grupo de Slack](#).

3. Algoritmos sobre listas y comprensión de listas

En esta clase empezamos a usar un IDE (Spyder), discutimos los tipos de errores que pueden aparecer en un programa. Introducimos algunas técnicas primitivas para buscar bugs y les proponemos una serie de ejercicios donde tienen que escribir algoritmos que operen sobre listas. Luego introducimos la *comprensión de listas*, un concepto muy hermoso y pythonesco. Cerramos la clase con una discusión sobre el concepto de objeto que subyace al lenguaje y algunos ejercicios para repasar los conceptos con el dataset de la clase pasada.

Recordá: aunque no termines todos los ejercicios, completá el formulario del final de la clase y subí los que hayas podido hacer ANTES del día miércoles 26/8 a las 14hs. Esto es parte de las condiciones de aprobación de la materia: todos podemos tener una semana complicada, pero en general esperamos que entregues la mayoría de los ejercicios.

- [3.1 Entorno de desarrollo integrado](#)
- [3.2 Errores](#)
- [3.3 Listas y búsqueda lineal](#)
- [3.4 Comprensión de listas](#)
- [3.5 Objetos](#)
- [3.6 Arbolado porteño y comprensión de listas](#)
- [3.7 Cierre de la tercera clase](#)

3.1 Entorno de desarrollo integrado

A partir de aquí te vamos a proponer trabajar principalmente dentro de un entorno de desarrollo integrado (IDE, por sus siglas en inglés).

En particular, sugerimos trabajar con el [Spyder](#) que es un entorno de desarrollo de Python diseñado para científicos, ingenieros y analistas de datos. El Spyder puede descargarse solo o como parte de la distribución de [Anaconda](#) que trae, además de Python y del Spyder, una serie de bibliotecas con módulos muy útiles para desarrollos relacionados a la ciencia de datos.

Esta es una imagen de una captura de pantalla del Spyder:

The screenshot shows the Spyder Python IDE interface. On the left, the code editor displays several Python scripts: generic.py, plot_inscriptos.py*, CART.py, compilar.py, and traducir.py. The plot_inscriptos.py* script is open and contains code related to calculating the number of bills needed to reach a certain height for an obelisk. The middle section features the Variable explorer, which lists variables like altura_obelisco (float, 67.5), baño (int, 3), día (int, 21), grosor_billete (float, 0.00011), and num_billetes (int, 1048576). Below the variable explorer is the Console 1/A tab, which shows a series of numerical values from 6 to 18 followed by their corresponding floating-point equivalents. The bottom status bar indicates the file is ready, the Kite plugin is active, and the current line and column are Line 31, Col 1.

En la captura se puede ver que por defecto el Spyder viene estructurado con tres ventanas. Un editor de código ocupa la mitad izquierda de la ventana, mientras que la mitad derecha se divide en una terminal (o consola) interactiva de Python en la mitad inferior y un inspector de variables en la mitad superior. El Spyder nos permite correr línea por línea el código del editor (tecla F9) y ver el estado de las variables en el inspector de variables, o ejecutarlo completo (tecla F5). También nos permite debuguear el código con facilidad (botones azules de la barra superior).

Les recomendamos que le dediquen un tiempo a probar sus últimos ejercicios en este entorno. Verán que es muy cómodo. Pueden mirar un [breve tutorial](#) donde no sólo les enseñan el uso de la tecla F5, sino también una introducción al uso del debugger (le dice depurador) que veremos en la próxima clase.

3.2 Errores

Tres tipos de errores:

Programando nos podemos encontrar con tres tipos de errores.

Los errores *sintácticos* son los que se dan cuando escribimos incorrectamente. Por ejemplo si queremos escribir `x = (a + b) * c` pero en vez de eso escribimos `x = (a + b] * c`, el programa no va a correr.

Un segundo tipo de error lo forman los errores *en tiempo de ejecución*, que se dan cuando el programa empieza a ejecutarse pero se produce un error durante su ejecución. Por ejemplo si le pedimos al usuario que ingrese su edad esperando un número entero e ingresa "veintiséis años", es probable que el programa dé un error. Si leemos un archivo CSV y una fila tiene datos faltantes, el programa puede dar un error. Este tipo de errores en Python generan *excepciones* que, como veremos más adelante, pueden administrarse adecuadamente.

El tercer tipo de error es el más difícil de encontrar y de entender. Son los *errores semánticos*, que se dan cuando el programa no hace lo que está diseñado para hacer. Tienen que ver con el sentido de las instrucciones. En estos casos el programa se ejecuta pero da un resultado incorrecto o inesperado. En general, la mejor forma de encontrar estos errores es correr paso a paso el código que genera un resultado inesperado, tratando de entender dónde está la falla, usando el debugger. Veremos cómo usar el debugger la clase que viene, por ahora trabajaremos de forma un poco más primitiva.

Debuggear a mano

Los errores (o bugs) son difíciles de rastrear y resolver. Especialmente errores que sólo aparecen bajo cierta combinación particular de condiciones que resulta en que el programa no pueda continuar o dé un resultado inesperado. Si tu programa corre, pero no da el resultado que esperás, o *se cuelga* y no entendés porqué, tenés algunas herramientas concretas que te ayudan a buscar el origen del problema. A continuación veremos algunas metodologías específicas (aunque un poco primitivas) que permiten rastrear el origen del problema.

¿Qué dice un traceback?

Si te da un error, lo primero que podés hacer es intentar entender la causa del error usando como punto de partida el "traceback":

```
python3 blah.py
Traceback (most recent call last):
  File "blah.py", line 13, in ?
    foo()
  File "blah.py", line 10, in foo
    bar()
```

```
File "blah.py", line 7, in bar
    spam()
File "blah.py", line 4, in spam
    x.append(3)
AttributeError: 'int' object has no attribute 'append'
```

La última línea dice algo como que "el objeto `int` no tiene un atributo `append`" - lo cual es obvio, pero ¿cómo llegamos ahí?

La última línea es el motivo concreto del error.

Las líneas anteriores te dicen el camino que siguió el programa hasta llegar al error. En este caso: el error ocurrió en `x.append(3)` en la línea 4, dentro de la función `spam` del módulo `"blah.py"`, que fue llamado por la función `bar` en la línea 7 del mismo archivo, que fue llamada por... y así siguiendo.

Sin embargo a veces esto no proporciona suficiente información (por ejemplo, no sabemos el valor de cada parámetro usado en las llamadas).

Una posibilidad que a veces da resultado es copiar el traceback en Google. Si estás usando una biblioteca de funciones que mucha gente usa (como `numpy` ó `math`) es muy probable que alguien se haya encontrado antes con el mismo problema que vos, y alguien más le haya explicado qué lo causa, o cómo evitarlo.

Usá el modo **REPL** de Python

Si usás Python desde la línea de comandos, podés usarlo pasándole un `-i` como parámetro antes del script a ejecutar. Cuando el intérprete de Python termine de ejecutar el script se va a quedar en modo interactivo (en lugar de volver al sistema operativo). Podés averiguar en qué estado quedó el sistema.

```
python3 -i blah.py
Traceback (most recent call last):
  File "blah.py", line 13, in ?
    foo()
  File "blah.py", line 10, in foo
    bar()
  File "blah.py", line 7, in bar
    spam()
  File "blah.py", line 4, in spam
    x.append(3)
AttributeError: 'int' object has no attribute 'append'
>>>     print( repr(x) )
```

Este `parámetro` (el `-i`, que ya usamos antes) preserva el estado del intérprete al finalizar el script y te permite interrogarlo sobre el estado de las variables y obtener

información que de otro modo perderías. En el ejemplo de recién interesa saber qué es `x` y cómo llegó a ese estado. Si estás usando un IDE esta posibilidad de interacción suele ocurrir naturalmente.

Debuggear con `print`

`print()` es una forma rápida y sencilla de permitir que el programa se ejecute (casi) normalmente mientras te da información del estado de las variables. Si elegís bien las variables que mostrar, es probable que digas "¡¡Ajá!!".

Sugerencia: es conveniente usar `repr()` para imprimir las variables

```
def spam(x):
    print('DEBUG:', repr(x))
    ...
```

`repr()` te muestra una representación técnicamente más precisa del valor de una variable, y no la representación *bonita* que solemos ver.

```
>>> from decimal import Decimal
>>> x = Decimal('3.4')
# SIN `repr`
>>> print(x)
3.4
# CON `repr`
>>> print(repr(x))
Decimal('3.4')
>>>
```

Debuggear con lápiz y papel

Muchas veces uno *asume* que el intérprete está haciendo algo. Si agarrás un lápiz y un papel y *hacés de intérprete* anotando el estado de cada variable y siguiendo las instrucciones del programa paso a paso, es posible que entiendas que las cosas no son como creías.

Estas alternativas son útiles pero un poco primitivas. La mejor forma de debuggear un programa en Python es usar el debugger.

Ejercicios:

En los siguientes ejercicios te proponemos que uses las técnicas que mencionamos arriba para resolver los problemas que aparecen a continuación. Determiná los

errores de los siguientes códigos y corregilos en un archivo `solucion_de_errores.py` comentando brevemente los errores. ¿Qué tipo de errores tiene cada uno?

En el archivo `solucion_de_errores.py` separá las correcciones de los distintos códigos con una línea que contenga solamente los símbolos `%%` seguido de una o varias líneas comentadas indicando el ejercicio y el problema que tenía. Al terminar, debería verse así tu archivo:

```
#solucion_de_errores.py
#Ejercicios de errores en el código
%%%
#Ejercicio 3.1. Función tiene_a()
#Comentario: El error era de TAL tipo y estaba ubicado en TAL lugar.
#    Lo corregí cambiando esto por aquello.
#    A continuación va el código corregido
...
...

%%%
#Ejercicio 3.2. Función tiene_a(), nuevamente
#Comentario: El error era de TAL tipo y estaba ubicado en TAL lugar.
...
...

%%%
#Ejercicio 3.3. Función tiene_uno()
#Comentario: El error era de TAL tipo y estaba ubicado en TAL lugar.
...
...
```

Ejercicio 3.1: Semántica

¿Anda bien en todos los casos de prueba?

```
def tiene_a(expresion):
    n = len(expresion)
    i = 0
    while i < n:
        if expresion[i] == 'a':
            return True
        else:
            return False
        i += 1

tiene_a('UNSAM 2020')
tiene_a('abracadabra')
tiene_a('La novela 1984 de George Orwell')
```

Ejercicio 3.2: Sintaxis

¿Anda bien en todos los casos de prueba?

```
def tiene_a(expresion)
    n = len(expresion)
    i = 0
    while i < n:
        if expresion[i] = 'a':
            return True
        i += 1
    return False

tiene_a('UNSAM 2020')
tiene_a('La novela 1984 de George Orwell')
```

Ejercicio 3.3: Tipos

¿Anda bien en todos los casos de prueba?

```
def tiene_uno(expresion):
    n = len(expresion)
    i = 0
    tiene = False
    while (i < n) and not tiene:
        if expresion[i] == '1':
            tiene = True
        i += 1
    return tiene

tiene_uno('UNSAM 2020')
tiene_uno('La novela 1984 de George Orwell')
tiene_uno(1984)
```

Ejercicio 3.4: Alcances

La siguiente suma no da lo que debería:

```
def suma(a,b):
    c = a + b

a = 2
b = 3
c = suma(a,b)
print(f"La suma da {a} + {b} = {c}")
```

Ejercicio 3.5: Pisando memoria

El siguiente ejemplo usa el dataset de la clase anterior, pero no lo imprime como corresponde, ¿podés determinar por qué y explicarlo brevemente en la versión corregida?

```
import csv
from pprint import pprint

def leer_camion(nombre_archivo):
    camion = []
    registro = {}
    with open(nombre_archivo, "rt") as f:
        filas = csv.reader(f)
        encabezado = next(filas)
        for fila in filas:
            registro[encabezado[0]] = fila[0]
            registro[encabezado[1]] = int(fila[1])
            registro[encabezado[2]] = float(fila[2])
        camion.append(registro)
    return camion

camion = leer_camion("Data/camion.csv")
pprint(camion)
```

Ayuda: Primero tratá de pensarlo, pero si este último se te hace muy difícil, podés mirar un poco de la teoría relacionada con esto un par de secciones más adelante ([Sección 3.5](#)).

3.3 Listas y búsqueda lineal

En esta sección seguiremos usando Python, pero nos concentraremos en la parte algorítmica. Vas a escribir funciones sencillas (y no tanto) que realicen operaciones de bajo nivel sobre listas.

Este es un curso de Python y de algoritmos. Python es un lenguaje de alto nivel. Esto significa que con pocas instrucciones permite realizar operaciones muy complejas. Los lenguajes de bajo nivel están más cerca del lenguaje del procesador y programar en ellos por ejemplo, un análisis de datos, es mucho más tedioso.

Sin embargo, entre las cosas que trae resueltas Python hay algunos algoritmos que nos interesa que vuelvas a escribir vos, por motivos didácticos. En lo que sigue te vamos a pedir en algunas ocasiones que no uses toda la potencia y simpleza de

Python sino que te arremangues y escribas algunas funciones desde los primeros rudimentos.

Queremos mostrarte en ejemplos concretos cómo distintas maneras de resolver un mismo problema pueden dar lugar a algoritmos con eficiencias muy diferentes. A veces una es mejor para un uso y la otra para otro uso. En concreto, vamos a profundizar en el problema de la búsqueda y en el problema del ordenamiento, que son dos problemas elementales que ilustran conceptos centrales del desarrollo de algoritmos.

El uso adecuado de estos conceptos puede hacer la diferencia entre un algoritmo que termina el procesamiento en unos pocos minutos o uno que hay que dejar corriendo dos días (y rezar para que no se corte la electricidad mientras corre).

Búsqueda lineal

El problema de la búsqueda

Presentamos ahora uno de los problemas más clásicos de la computación: el problema de la búsqueda. El mismo se puede enunciar de la siguiente manera:

Problema: Dada una lista `lista` y un elemento `e` devolver el índice de `e` en `lista` si `e` está en `lista`, y devolver `-1` si `e` no está en `lista`.

Este problema tiene una solución muy sencilla en Python: se puede usar el método `index()` de las listas.

Probá esta solución:

```
>>> [1, 3, 5, 7].index(5)
2
>>> [1, 3, 5, 7].index(20)
Traceback (most recent call last):

  File "<ipython-input-177-1bcce50c5c91>", line 1, in <module>
    [1, 3, 5, 7].index(20)

ValueError: 20 is not in list
```

Vemos que usar la función `index()` resuelve nuestro problema si el valor buscado está en la lista, pero si el valor no está no sólo no devuelve un `-1`, sino que se produce un error.

El problema es que para poder aplicar la función `index()` debemos estar seguros de que el valor está en la lista, y para averiguar eso Python nos provee del operador `in`:

```
>>> 5 in [1, 3, 5, 7]
True
>>> 20 in [1, 3, 5, 7]
False
```

Si llamamos a la función `index()` sólo cuando el resultado de `in` es verdadero, y devolvemos `-1` cuando el resultado de `in` es falso, estaremos resolviendo el problema planteado usando sólo funciones provistas por Python:

```
def busqueda_con_index(lista, e):
    '''Busca un elemento e en la lista.

    Si e está en lista devuelve el índice,
    de lo contrario devuelve -1.
    '''
    if e in lista:
        pos = lista.index(e)
    else:
        pos = -1
    return pos
```

Probemos la función `busqueda_con_index()`:

```
>>> busqueda_con_index([1, 4, 54, 3, 0, -1], 1)
0
>>> busqueda_con_index([1, 4, 54, 3, 0, -1], -1)
5
>>> busqueda_con_index([1, 4, 54, 3, 0, -1], 3)
3
>>> busqueda_con_index([1, 4, 54, 3, 0, -1], 44)
-1
>>> busqueda_con_index([], 0)
-1
```

¿Cuántas comparaciones hace este programa?

Es decir, ¿cuánto esfuerzo computacional requiere este programa? ¿Cuántas veces compara el valor que buscamos con los datos de la lista? No lo sabemos porque no sabemos cómo están implementadas las operaciones `in` e `index()`. La pregunta queda planteada por ahora pero daremos un método para averiguarlo más adelante.

Búsqueda lineal

Nos interesa estudiar formas alternativas de programar la búsqueda usando operaciones más elementales, y no las primitivas `in` e `index()` de nuestro lenguaje de alto nivel. Aceptemos entonces que no vamos a usar ni `in` ni `index()`. En cambio, podemos iterar sobre los índices y elementos de una lista para hacer comparaciones elementales.

Consideremos la siguiente solución: iterar sobre los índices y elementos de una lista de manera de comparar el elemento `e` buscado con cada uno de los elementos de la lista y devolver la posición donde lo encontramos, en caso de encontrarlo. Si llegamos al final de la lista sin haber salido antes de la función es porque el valor de `e` no está en la lista, y en ese caso devolvemos -1.

En esta solución lo ideal es usar `enumerate` (ver la [Sección 2.5](#)) ya que dentro de la iteración necesitamos tener acceso tanto al valor del elemento (para ver si es igual al buscado) como a su índice (es el valor que tenemos que devolver).

Primero hagámoslo sin usarlo y luego lo agregamos para entender su ventaja. En ambos casos necesitamos una variable `i` que cuente en cada momento en qué posición de la lista estamos. Si no usamos `enumerate`, debemos inicializar `i` en 0 antes de entrar en el ciclo e incrementarla en 1 en cada paso.

El programa nos queda así:

```
def busqueda_lineal(lista, e):
    '''Si e está en la lista devuelve su posición, de lo contrario devuelve -1.'''
    pos = -1 # comenzamos suponiendo que e no está
    i = 0
    for z in lista: # recorremos los elementos de la lista
        if z == e: # si encontramos a e
            pos = i # guardamos su posición
            break # y salimos del ciclo
        i += 1
    return pos
```

La versión con `enumerate` es mucho más elegante:

```
def busqueda_lineal(lista, e):
    '''Si e está en la lista devuelve su posición, de lo contrario devuelve -1.'''
    pos = -1 # comenzamos suponiendo que e no está
    for i, z in enumerate(lista): # recorremos la lista
        if z == e: # si encontramos a e
            pos = i # guardamos su posición
            break # y salimos del ciclo
```

```
    return pos
```

Y ahora lo probamos:

```
>>> busqueda_lineal([1, 4, 54, 3, 0, -1], 44)
-1
>>> busqueda_lineal([1, 4, 54, 3, 0, -1], 3)
3
>>> busqueda_lineal([1, 4, 54, 3, 0, -1], 0)
4
>>> busqueda_lineal([], 42)
-1
```

¿Cuántas comparaciones hace este programa?

Volvemos a preguntarnos lo mismo que en la sección anterior pero con el nuevo programa: ¿cuánto esfuerzo computacional requiere este programa?, ¿cuántas veces compara el valor que buscamos con los datos de la lista? Ahora podemos analizar el código de `busqueda_lineal`:

El ciclo recorre uno a uno los elementos de la lista, y en el cuerpo de ese ciclo, se compara cada elemento con el valor buscado. En el caso de encontrarlo se devuelve la posición. Si el valor no está en la lista, se recorrerá la lista entera, haciendo una comparación por cada elemento.

O sea que si el valor está en la posición p de la lista se hacen p comparaciones. En el *peor caso*, si el valor no está, se hacen tantas comparaciones como elementos tenga la lista.

En resumen: Si la lista crece, la cantidad de comparaciones para encontrar un valor arbitrario crecerá en forma proporcional al tamaño de la lista. Es decir que:

El algoritmo de búsqueda lineal tiene un comportamiento *proporcional a la longitud de la lista involucrada*, o que es un algoritmo *lineal*.

Ejercicios

Ejercicio 3.6: Búsquedas de un elemento

Creá el archivo `busqueda_en_listas.py` para guardar tu código de este ejercicio y el siguiente.

En este primer ejercicio tenés que escribir una función `buscar_u_elemento()` que reciba una lista y un elemento y devuelva la posición de la última aparición de ese elemento en la lista (o -1 si el elemento no pertenece a la lista).

Probá tu función con algunos ejemplos:

```
>>> buscar_u_elemento([1,2,3,2,3,4],1)
0
>>> buscar_u_elemento([1,2,3,2,3,4],2)
3
>>> buscar_u_elemento([1,2,3,2,3,4],3)
4
>>> buscar_u_elemento([1,2,3,2,3,4],5)
-1
```

Agregale a tu programa `busqueda_en_listas.py` una función `buscar_n_elemento()` que reciba una lista y un elemento y devuelva la cantidad de veces que aparece el elemento en la lista. Probá también esta función con algunos ejemplos.

Ejercicio 3.7: Búsqueda de máximo y mínimo

Agregale a tu archivo `busqueda_en_listas.py` una función `maximo()` que busque el valor máximo de una lista de números positivos. Python tiene el comando `max` que ya hace esto, pero como práctica te proponemos que completes el siguiente código:

```
def maximo(lista):
    '''Devuelve el máximo de una lista,
    la lista debe ser no vacía y de números positivos.
    '''
    # m guarda el máximo de los elementos a medida que corro la lista.
    m = 0 # Lo inicializo en 0
    for e in lista: # Recorro la lista y voy guardando el mayor
        ...
    return m
```

Probá tu función con estos ejemplos:

```
>>> maximo([1,2,7,2,3,4])
7
>>> maximo([1,2,3,4])
4
>>> maximo([-5,4])
4
>>> maximo([-5,-4])
0
```

¿Por qué falla en el último caso? ¿Por qué anda en el caso anterior? ¿Cómo se puede inicializar `m` para que la función ande también con números negativos? Corregilo y guarda la versión mejorada en el archivo `busqueda_en_listas.py`.

Si te dan ganas, agregá una función `minimo()` al archivo.

Ejercitación con iteradores y listas

Ejercicio 3.8: Invertir una lista

Escribí una función `invertir_lista(lista)` que dada una lista devuelva otra que tenga los mismos elementos pero en el orden inverso. Es decir, el que era el primer elemento de la lista de entrada deberá ser el último de la lista de salida y análogamente con los demás elementos.

```
def invertir_lista(lista):
    invertida = []
    for e in lista: # Recorro la lista
        ... #agrego el elemento e al principio de la lista invertida
    return invertida
```

Guardá la función en el archivo `invlista.py` y probala con las siguientes listas:

1. `[1, 2, 3, 4, 5]`
2. `['Bogotá', 'Rosario', 'Santiago', 'San Fernando', 'San Miguel']`

Ejercicio 3.9: Propagación

Imagina una fila con varios fósforos uno al lado del otro. Los fósforos pueden estar en tres estados: nuevos, prendidos fuego o ya gastados (carbonizados). Representaremos esta situación con una lista L con un elemento por fósforo, que en cada posición tiene un 0 (nuevo), un 1 (encendido) o un -1 (carbonizado). El fuego se propaga inmediatamente de un fósforo encendido a cualquier fósforo nuevo que tenga a su lado. Los fósforos carbonizados no se encienden nuevamente.

Escribí una función llamada `propagar` que reciba un vector con 0's, 1's y -1's y devuelva un vector en el que los 1's se propagaron a sus vecinos con 0. Guardalo en un archivo `propaga.py`.

Por ejemplo:

```
>>> propagar([ 0, 0, 0,-1, 1, 0, 0, 0,-1, 0, 1, 0, 0])
```

```
[ 0, 0, 0,-1, 1, 1, 1,-1, 1, 1, 1]
>>> propagar([ 0, 0, 0, 1, 0, 0])
[ 1, 1, 1, 1, 1, 1]
```



Propagación análoga a la del Ejercicio

3.4 Comprensión de listas

Una tarea que realizamos una y otra vez es procesar los elementos de una lista. En esta sección introducimos la definición de listas por compresión que es una herramienta potente para hacer exactamente eso.

Crear listas nuevas

La comprensión de listas crea un una nueva lista aplicando una operación a cada elemento de una secuencia.

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [2*x for x in a]
>>> b
[2, 4, 6, 8, 10]
>>>
```

Otro ejemplo:

```
>>> nombres = ['Edmundo', 'Juana']
>>> a = [nombre.lower() for nombre in nombres]
>>> a
['edmundo', 'juana']
>>>
```

La sintaxis general es : [<expresión> for <variable> in <secuencia>].

Filtros

La comprensión de listas se puede usar para filtrar.

```
>>> a = [1, -5, 4, 2, -2, 10]
>>> b = [2*x for x in a if x > 0]
>>> b
[2, 8, 4, 20]
>>>
```

Casos de uso

La comprensión de listas es enormemente útil. Por ejemplo, podés recolectar los valores de un campo específico de un diccionario:

```
frutas = [s['nombre'] for s in camion]
```

O podés hacer consultas (*queries*) como si las secuencias fueran bases de datos.

```
a = [s for s in camion if s['precio'] > 100 and s['cajones'] > 50 ]
```

También podés combinar la comprensión de listas con reducciones de secuencias:

```
costo = sum([s['cajones']*s['precio'] for s in camion])
```

Sintaxis general

```
[<expresión> for <variable> in <secuencia> if <condición>]
```

Lo que significa

```
resultado = []
for variable in secuencia:
    if condición:
        resultado.append(expresión)
```

Digresión histórica

La comprensión de listas viene de la matemática (definición de conjuntos por comprensión).

```
a = [ x * x for x in s if x > 0 ] # Python
a = { x^2 | x ∈ s, x > 0 }           # Matemática
```

La mayoría de los programadores no suelen pensar en el costado matemático de esta herramienta. Podemos verla simplemente como una abreviación copada para definir listas.

Ejercicios

Corré tu programa `informe.py` de forma de tener los datos sobre cajones cargados en tu intérprete en modo interactivo.

Luego, tratá de escribir los comandos adecuados para realizar las operaciones descriptas abajo. Estas operaciones son reducciones, transformaciones y consultas sobre la carga del camión.

Ejercicio 3.10: Comprensión de listas

Probá un par de comprensión de listas para familiarizarte con la sintaxis.

```
>>> nums = [1, 2, 3, 4]
>>> cuadrados = [x * x for x in nums]
>>> cuadrados
[1, 4, 9, 16]
>>> dobles = [2 * x for x in nums if x > 2]
>>> dobles
[6, 8]
>>>
```

Observá que estás creando nuevas listas con los datos adecuadamente transformados o filtrados.

Ejercicio 3.11: Reducción de secuencias

Calculá el costo total de la carga del camión en un solo comando.

```
>>> camion = leer_camion('Data/camion.csv')
>>> costo = sum([s['cajones'] * s['precio'] for s in camion])
>>> costo
47671.15
>>>
```

Luego, leyendo la variable `precios`, calculá también el valor en el mercado de la carga del camión usando una sola línea de código.

```
>>> precios = leer_precios('Data/precios.csv')
>>> valor = sum([s['cajones'] * precios[s['nombre']] for s in camion])
>>> valor
```

```
62986.1
```

```
>>>
```

Ambos son ejemplos de aplicación-reducción. La comprensión de listas está aplicando una operación a lo largo de la lista.

```
>>> [ s['cajones'] * s['precio'] for s in camion ]
[3220.000000000005, 4555.0, 15516.0, 10246.0, 3835.1499999999996,
3254.999999999995, 7044.0]
>>>
```

La función `sum()` luego realiza una reducción del resultado

```
>>> sum(_)
47671.15
>>>
```

Con este conocimiento algunos ya empiezan su startup de big-data.

Ejercicio 3.12: Consultas de datos

Probá los siguientes ejemplos de consultas (queries) de datos.

Primero, generá una lista con la info de todas las frutas que tienen más de 100 cajones en el camión.

```
>>> mas100 = [ s for s in camion if s['cajones'] > 100 ]
>>> mas100
[{'cajones': 150, 'nombre': 'Caqui', 'precio': 103.44},
 {'cajones': 200, 'nombre': 'Mandarina', 'precio': 51.23}]
>>>
```

Ahora, una con la info sobre cajones de Mandarina y Naranja.

```
>>> myn = [ s for s in camion if s['nombre'] in {'Mandarina', 'Naranja'} ]
>>> myn
[{'cajones': 50, 'nombre': 'Naranja', 'precio': 91.1},
 {'cajones': 200, 'nombre': 'Mandarina', 'precio': 51.23},
 {'cajones': 50, 'nombre': 'Mandarina', 'precio': 65.1},
 {'cajones': 100, 'nombre': 'Naranja', 'precio': 70.44}]
>>>
```

O una con la info de las frutas que costaron más de \$10000.

```
>>> costo10k = [ s for s in camion if s['cajones'] * s['precio'] > 10000 ]
>>> costo10k
```

```
[{'cajones': 150, 'nombre': 'Caqui', 'precio': 103.44},  
 {'cajones': 200, 'nombre': 'Mandarina', 'precio': 51.23}]  
>>>
```

Esta forma de escribir resulta análoga a las consultas a una base de datos con SQL.

Ejercicio 3.13: Extracción de datos

Usando un comprensión de listas, construí una lista de tuplas (`nombre`, `cajones`) que indiquen la cantidad de cajones de cada fruta tomando los datos de `camion`.

```
>>> nombre_cajones = [ (s['nombre'], s['cajones']) for s in camion ]  
>>> nombre_cajones  
[('Lima', 100), ('Naranja', 50), ('Caqui', 150), ('Mandarina', 200),  
 ('Durazno', 95), ('Mandarina', 50), ('Naranja', 100)]  
>>>
```

Si cambiás los corchetes (`[,]`) por llaves (`{, }`), obtenés algo que se conoce como comprensión de conjuntos. Vas a obtener valores únicos.

Por ejemplo, si quisieras un listado de las frutas en el camión pordías usar:

```
>>> nombres = { s['nombre'] for s in camion }  
>>> nombres  
{'Caqui', 'Durazno', 'Lima', 'Mandarina', 'Naranja'}  
>>>
```

Si especificás pares `clave:valor`, podés construir un diccionario. Por ejemplo, si queremos un diccionario con el total de cada fruta en el camión podemos comenzar con

```
>>> stock = { nombre: 0 for nombre in nombres }  
>>> stock  
{'Caqui': 0, 'Durazno': 0, 'Lima': 0, 'Mandarina': 0, 'Naranja': 0}  
>>>
```

que es una comprensión de diccionario. Y seguir sumando los cajones:

```
>>> for s in camion:  
     stock[s['nombre']] += s['cajones']  
  
>>> stock  
{'Caqui': 150, 'Durazno': 95, 'Lima': 100, 'Mandarina': 250, 'Naranja':  
 150}  
>>>
```

Otro ejemplo útil podría ser generar un diccionario de precios de venta de aquellos productos que están efectivamente cargados en el camión:

```
>>> camion_precios = { nombre: precios[nombre] for nombre in nombres }
>>> camion_precios
{'Caqui': 105.46, 'Durazno': 73.48, 'Lima': 40.22, 'Mandarina': 80.89,
'Naranja': 106.28}
>>>
```

Ejercicio 3.14: Extraer datos de una arhcivo CSV

Saber usar combinaciones de comprensión de listas, diccionarios y conjuntos resulta útil para procesar datos en diferentes contextos. Aunque puede volverse medio críptico si no estás habituado. Acá te mostramos un ejemplo de cómo extraer columnas seleccionadas de un archivo CSV que tiene esas características. No es difícil cuando lo entendés, pero está muy concentrado todo.

Primero, leamos el encabezado (header) del archivo CSV:

```
>>> import csv
>>> f = open('Data/fecha_camion.csv')
>>> rows = csv.reader(f)
>>> headers = next(rows)
>>> headers
['nombre', 'fecha', 'hora', 'cajones', 'precio']
>>>
```

Luego, definamos una lista que tenga las columnas que nos importan:

```
>>> select = ['nombre', 'cajones', 'precio']
>>>
```

Ubiquemos los índices de esas columnas en el CSV:

```
>>> indices = [headers.index(ncolumna) for ncolumna in select]
>>> indices
[0, 3, 4]
>>>
```

Y finalmente leamos los datos y armemos un diccionario usando comprensión de diccionarios:

```
>>> row = next(rows)
>>> record = {ncolumna: row[index] for ncolumna, index in zip(select, indices)} # comprensión de diccionario
>>> record
{'precio': '32.20', 'nombre': 'Lima', 'cajones': '100'}
```

```
>>>
```

No es trivial este comando. El comando es sintácticamente muy compacto, pero es conceptualmente (un poco) complejo. Cuando te sientas cómodo con esta lectura de una línea del archivo (si no pasa, tranca, podemos seguir sin esto), leé el resto:

```
>>> camion = [ { ncolumna: row[index] for ncolumna, index in zip(select, indices) } for row in rows ]
>>> camion
[{'cajones': '50', 'nombre': 'Naranja', 'precio': '91.1'},
 {'cajones': '150', 'nombre': 'Caqui', 'precio': '103.44'},
 {'cajones': '200', 'nombre': 'Mandarina', 'precio': '51.23'},
 {'cajones': '95', 'nombre': 'Durazno', 'precio': '40.37'},
 {'cajones': '50', 'nombre': 'Mandarina', 'precio': '65.1'},
 {'cajones': '100', 'nombre': 'Naranja', 'precio': '70.44'}]
```

```
>>>
¡Por las barbas de mi abuelo! Acabamos de reducir casi toda la función leer_camion() a un solo comando.
```

Comentario

La comprensión de listas se usa frecuentemente Python. Es una forma eficiente de transformar, filtrar o juntar datos. Tiene una sintaxis potente pero tratá de no pasarte con su uso: mantené cada comando tan simple como sea posible. Está perfecto descomponer un solo comando complejo en muchos pasos. Concretamente: compartir el último ejemplo con personas desprevenidas puede no ser lo ideal.

Dicho esto, saber manipular datos rápidamente es una habilidad increíblemente útil. Hay numerosas situaciones donde puede que tengas que resolver algún tipo de problema excepcional (en el sentido de raro o único) para importar, extraer o exportar datos. La comprensión de listas te puede ahorrar muchísimo tiempo en esas tareas.

3.5 Objetos

En esta sección introducimos algunos conceptos sobre el modelo de objeto interno de Python y discutimos algunos temas relacionados con el manejo de memoria, copias de variable y verificación de tipos.

Asignaciones

Muchas operaciones en Python están relacionadas con *asignar* o *guardar* valores.

```
a = valor      # Asignación a una variable
s[n] = valor    # Asignación a una lista
s.append(valor) # Agregar a una lista
d['key'] = valor # Agregar a una diccionario
```

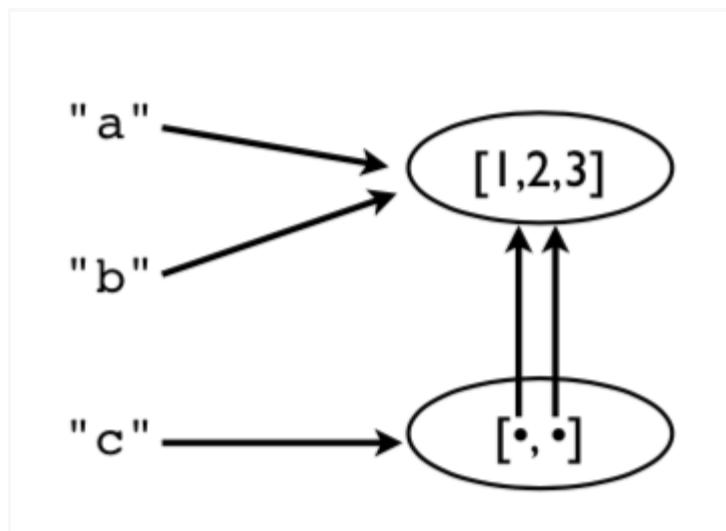
Ojo: las operaciones de asignación nunca hacen una copia del valor asignado. Las asignaciones son simplemente copias de las referencias (o copias del puntero, si preferís).

Ejemplo de asignación

Considerá este fragmento de código.

```
a = [1,2,3]
b = a
c = [a,b]
```

A continuación te mostramos en un gráfico las operaciones de memoria sujetas. En este ejemplo, hay solo un objeto lista [1,2,3], pero hay cuatro referencias a él.



Esto significa que al modificar un valor modificamos *todas* las referencias.

```
>>> a.append(999)
>>> a
[1,2,3,999]
>>> b
[1,2,3,999]
>>> c
[[1,2,3,999], [1,2,3,999]]
>>>
```

Observá cómo un cambio en la lista original desencadena cambios en todas las demás variables (ouch!). Esto es porque no se hizo ninguna copia. Todos son punteros a la misma cosa.

Esto es lo mismo que pasaba en el [Ejercicio 3.5](#).

Reasignar valores

La reasignación de valores *nunca* sobreescribe la memoria ocupada por un valor anterior.

```
a = [1, 2, 3]
b = a
a = [4, 5, 6]

print(a)      # [4, 5, 6]
print(b)      # [1, 2, 3]      Mantiene el valor original
```

Acordate: Las variables son nombres, no ubicaciones en la memoria.

Peligros

Si no te explican esto, tarde o temprano te trae problemas. Un típico ejemplo es cuando cambiás un dato pensando que es una copia privada y, sin querer, esto corrompe los datos en otra parte del programa.

Comentario: Esta es una de las razones por las que los tipos de datos primitivos (`int`, `float`, `string`) son *immutables* (de sólo lectura).

Identidad y referencias

Podés usar el operador `is` (es) para verificar si dos valores corresponden al mismo objeto.

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
>>>
```

`is` compara la identidad del objeto (que está representada por un número entero). Esta identidad también la podés obtener usando `id()`.

```
>>> id(a)
```

```
3588944
>>> id(b)
3588944
>>>
```

Observación: Para ver si dos valores son iguales, es mejor usar el `==`. El comportamiento de `is` puede dar resultados inesperados:

```
>>> a = [1, 2, 3]
>>> b = a
>>> c = [1, 2, 3]
>>> a is b
True
>>> a is c
False
>>> a == c
True
>>>
```

Copias superficiales

Las listas y diccionarios tienen métodos para hacer copias (no meras referencias, sino duplicados):

```
>>> a = [2, 3, [100, 101], 4]
>>> b = list(a) # Hacer una copia
>>> a is b
False
```

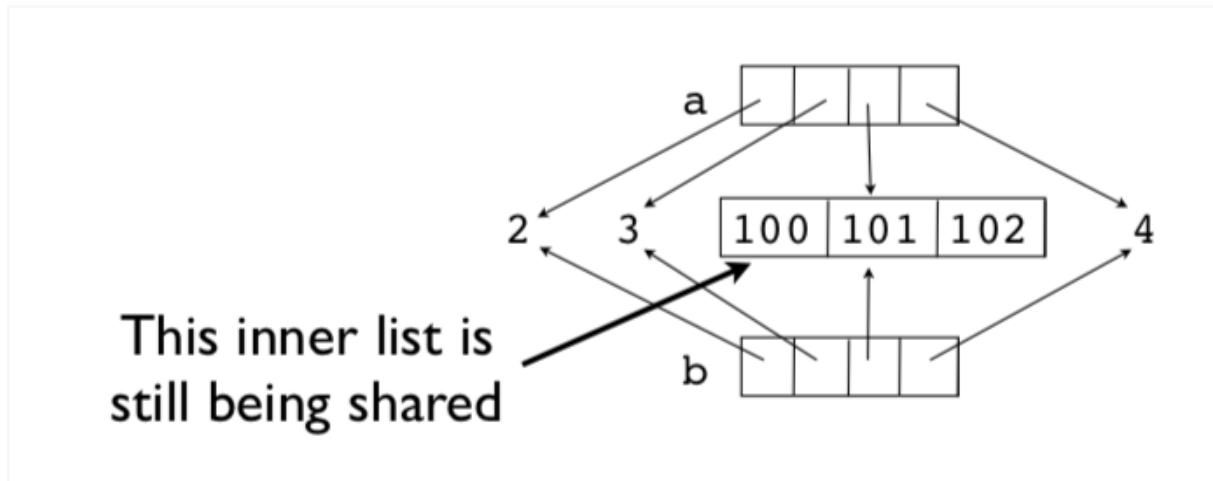
Ahora `b` es una nueva lista.

```
>>> a.append(5)
>>> a
[2, 3, [100, 101], 4, 5]
>>> b
[2, 3, [100, 101], 4]
```

A pesar de esto, los elementos de `a` y de `b` siguen siendo compartidos.

```
>>> a[2].append(102)
>>> b[2]
[100, 101, 102]
>>>
>>> a[2] is b[2]
True
>>>
```

En este ejemplo, la lista interna [100, 101, 102] es compartida por ambas variables. La copia que hicimos con el comando `b = list(a)` es un *copia superficial* (superficial en el sentido de *poco profunda*, en inglés se dice *shallow copy*). Mirá este gráfico.



La lista interna sigue siendo compartida.

Copias profundas

A veces vas a necesitar hacer una copia de un objeto así como de todos los objetos que contenga. Llamamos a esto una *copia profunda* (*deep copy*). Podés usar la función `deepcopy` del módulo `copy` para esto:

```
>>> a = [2, 3, [100, 101], 4]
>>> import copy
>>> b = copy.deepcopy(a)
>>> a[2].append(102)
>>> b[2]
[100, 101]
>>> a[2] is b[2]
False
>>>
```

Nombre, valores y tipos

Los nombres de variables no tienen un tipo asociado. Sólo son nombres. Pero los valores sí tienen un tipo subyacente.

```
>>> a = 42
>>> b = 'Hello World'
>>> type(a)
<type 'int'>
>>> type(b)
```

```
<type 'str'>
type() te dice el tipo del valor.
```

Verificación de tipos

Podés verificar si un objeto es una instancia de cierto tipo.

```
if isinstance(a, list):
    print('a es una lista')
```

O incluso si su tipo está entre varios tipos.

```
if isinstance(a, (list,tuple)):
    print('a una lista o una tupla')
```

Cuidado: Demasiadas verificaciones de tipos pueden resultar en un código excesivamente complejo. Típicamente lo usás para evitar errores comunes cometidos por otros usuarios de tu código.

Todo es un objeto

Números, cadenas, listas, funciones, excepciones, clases, instancias, etc. son todos objetos. Esto significa que pueden ser nombrados, pueden ser pasados como datos, ubicados en contenedores, etc. sin restricciones. No hay objetos especiales en Python. Todos los objetos viajan en primera clase.

Un ejemplo simple:

```
>>> import math
>>> items = [abs, math, ValueError]
>>> items
[<built-in function abs>,
 <module 'math' (builtin)>,
 <type 'exceptions.ValueError'>]
>>> items[0](-45)
45
>>> items[1].sqrt(2)
1.4142135623730951
>>> try:
    x = int('not a number')
except items[2]:
    print('Failed!')
Failed!
>>>
```

Acá, `items` es una lista que tiene una función, un módulo y una excepción. Sí, éste es un ejemplo raro. Pero es un ejemplo al fin. Podés usar los elementos de la lista en lugar de los nombres originales:

```
items[0] (-45)      # abs
items[1].sqrt(2)    # math
except items[2]:    # ValueError
```

Con un gran poder viene siempre una gran responsabilidad. Que puedas no significa que debas hacer este tipo de cosas.

Ejercicios

En estos ejercicios mostramos algo de la potencia que tiene el hecho de que todos los objetos sean de la misma jerarquía.

Ejercicio 3.15: Datos de primera clase

En el archivo `Data/camion.csv`, leímos datos organizados en columnas que se ven así:

```
nombre,cajones,precio
"Lima",100,32.20
"Naranja",50,91.10
...
```

En las clases anteriores, usamos el módulo `csv` para leer el archivo, pero tuvimos que hacer conversiones de tipo. Por ejemplo:

```
for row in rows:
    nombre = row[0]
    cajones = int(row[1])
    precio = float(row[2])
```

Este tipo de conversiones puede hacerse de una manera más inteligente usando algunas operaciones de listas.

Hagamos una lista de Python con los nombres de las funciones de conversión que necesitamos para convertir cada columna al tipo apropiado:

```
>>> types = [str, int, float]
>>>
```

Podés crear esta lista porque en Python todos los objetos son de la misma clase (de primera clase, digamos). Por lo tanto, si querés tener funciones en una lista, no pasa

nada. Los elementos de la lista que creaste son funciones que convierten un valor x a un tipo dado (`str(x)`, `int(x)`, `float(x)`).

Ahora, leé una fila de datos del archivo anterior:

```
>>> import csv
>>> f = open('Data/camion.csv')
>>> rows = csv.reader(f)
>>> headers = next(rows)
>>> row = next(rows)
>>> row
['Lima', '100', '32.20']
>>>
```

Como ya dijimos, con esta fila no podemos hacer operaciones porque los tipos son incorrectos. Por ejemplo:

```
>>> row[1] * row[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
>>>
```

Sin embargo, los datos pueden aparecerse con los tipos especificados en `types`. Por ejemplo:

```
>>> types[1]
<type 'int'>
>>> row[1]
'100'
>>>
```

Probá convertir uno de los valores:

```
>>> types[1](row[1])      # Es equivalente a int(row[1])
100
>>>
```

Probá con otro:

```
>>> types[2](row[2])      # Equivalente a float(row[2])
32.2
>>>
```

Probá calcular usando los tipos convertidos:

```
>>> types[1](row[1])*types[2](row[2])
```

```
3220.0000000000005
>>>
```

Hagamos un Zip de la lista de tipos con la de datos y veamos el resultado:

```
>>> r = list(zip(types, row))
>>> r
[(<type 'str'>, 'Lima'), (<type 'int'>, 100), (<type 'float'>, 32.2)]
>>>
```

Se puede ver que esto aparea una función de conversión de tipos con un valor. Por ejemplo, `int` está en un par con el valor '100'.

Esta lista zipeada es útil si querés realizar conversiones de todos los valores. Por ejemplo:

```
>>> converted = []
>>> for func, val in zip(types, row):
...     converted.append(func(val))
...
>>> converted
['Lima', 100, 32.2]
>>> converted[1] * converted[2]
3220.0000000000005
>>>
```

Asegurate de entender lo que está pasando en el código de arriba. En el ciclo la variable `func` va tomando los valores de las funciones de conversión de tipos (`str`, `int`, `float`) y la variable `val` va tomando los valores de los datos en la fila: 'Lima', '100', '32.2'. La expresión `func(val)` convierte los tipos de cada dato.

El código de arriba puede comprimirse en una sola instrucción usando comprensión de listas.

```
>>> converted = [func(val) for func, val in zip(types, row)]
>>> converted
['Lima', 100, 32.2]
>>>
```

Ejercicio 3.16: Diccionarios

¿Te acordás que la función `dict()` te permite hacer fácilmente un diccionario si tenés una secuencia de tuplas con claves y valores? Hagamos un diccionario usando el encabezado de las columnas:

```
>>> headers
```

```

['nombre', 'cajones', 'precio']
>>> converted
['Lima', 100, 32.2]
>>> dict(zip(headers, converted))
{'precio': 32.2, 'nombre': 'Lima', 'cajones': 100}
>>>

```

Si estás en sintonía con la comprensión de listas podés escribir una sola línea usando comprensión de diccionarios:

```

>>> { name: func(val) for name, func, val in zip(headers, types, row) }
{'precio': 32.2, 'name': 'Lima', 'cajones': 100}
>>>

```

Ejercicio 3.17: Fijando ideas

Usando las técnicas de este ejercicio, vas a poder escribir instrucciones que conviertan fácilmente campos como los de nuestro archivo en un diccionario de Python.

Para ilustrar esto, supongamos que leés un archivo de datos de la siguiente forma:

```

>>> f = open('Data/dowstocks.csv')
>>> rows = csv.reader(f)
>>> headers = next(rows)
>>> row = next(rows)
>>> headers
['name', 'price', 'date', 'time', 'change', 'open', 'high', 'low',
'volume']
>>> row
['AA', '39.48', '6/11/2007', '9:36am', '-0.18', '39.67', '39.69', '39.45',
'181800']
>>>

```

Convertamos estos datos usando un truco similar:

```

>>> types = [str, float, str, str, float, float, float, float, int]
>>> converted = [func(val) for func, val in zip(types, row)]
>>> record = dict(zip(headers, converted))
>>> record
{'volume': 181800, 'name': 'AA', 'price': 39.48, 'high': 39.69,
'low': 39.45, 'time': '9:36am', 'date': '6/11/2007', 'open': 39.67,
'change': -0.18}
>>> record['name']
'AA'
>>> record['price']
39.48
>>>

```

Bonus: ¿Cómo modificarías este ejemplo para transformar la fecha (`date`) en una tupla como `(6, 11, 2007)`?

Es importante que entiendas lo que hicimos en este ejercicio. Volveremos sobre esto más adelante.

3.6 Arbolado porteño y comprensión de listas

Seguimos aquí con el arbolado porteño. Vamos a plantear algunos ejercicios para hacer con la técnica de comprensión de listas introducida recientemente.

Ejercicios

Seguimos trabajando con el archivo CSV de "Arbolado en espacios verdes" que ya está en tu carpeta `Data`. Vamos a estudiar esta base de datos y responder algunas preguntas. Guardá los ejercicios de esta sección en el archivo `arboles.py`.

Ejercicio 3.18: Lectura de todos los árboles

Basándote en la función `leer_parque(nombre_archivo, parque)` del [Ejercicio 2.22](#), escribí otra `leer_arboles(nombre_archivo)` que lea el archivo indicado y devuelva una lista de diccionarios con la información de todos los árboles en el archivo. La función debe devolver una lista conteniendo un diccionario por cada árbol con todos los datos.

Vamos a llamar `arboleda` a esta lista.

Ejercicio 3.19: Lista de altos de Jacarandá

Usando comprensión de listas y la variable `arboleda` podés por ejemplo armar la lista de la altura de los árboles.

```
H=[float(arbol['altura_tot']) for arbol in arboleda]
```

Usá los filtros (recordá la [Sección 3.4](#)) para armar la lista de alturas de los Jacarandás solamente.

Ejercicio 3.20: Lista de altos y diámetros de Jacarandá

En el ejercicio anterior usaste una sola linea para seleccionar las alturas de los Jacarandás en parques porteños. Ahora te proponemos que armes una nueva lista que tenga pares (tuplas de longitud 2) conteniendo no solo el alto sino también el diámetro de cada Jacarandá en la lista.

Esperamos que obtengas una lista similar a esta:

```
[ (5.0, 10.0),  
  (5.0, 10.0),  
  ...  
  (12.0, 25.0),  
  ...  
  (7.0, 97.0),  
  (8.0, 28.0),  
  (2.0, 30.0),  
  (3.0, 10.0),  
  (17.0, 40.0) ]
```

Ejercicio 3.21: Diccionario con medidas

En este ejercicio vamos a considerar algunas especies de árboles. Por ejemplo:

```
especies = ['Eucalipto', 'Palo borracho rosado', 'Jacarandá']
```

Te pedimos que armes un diccionario en el que estas `especies` sean las claves y los valores asociados sean los datos que generaste en el ejercicio anterior. Más aún, organizá tu código dentro de una función `medidas_de_especies(especies, arboleada)` que recibe una lista de nombres de especies y una lista como la del [Ejercicio 3.18](#) y devuelve un diccionario cuyas claves son estas `especies` y sus valores asociados sean las medidas generadas en el ejercicio anterior.

Vamos a usar esta función la semana próxima. A modo de control, si llamás a la función con las tres especies del ejemplo como parámetro (`['Eucalipto', 'Palo borracho rosado', 'Jacarandá']`) y la `arboleada` entera, deberías recibir un diccionario con tres entradas (una por especie), cada una con una lista asociada conteniendo 4112, 3150 y 3255 pares de números (altos y diámetros), respectivamente.

Acordate de guardar los ejercicios de esta sección en el archivo `árboles.py`.

Extra: casi todos usan un `for` para crear este diccionario. ¿Lo podés hacer usando una comprensión de diccionarios? Te recordamos la sintaxis: `diccionario = { clave: valor for clave in claves }`

3.7 Cierre de la tercera clase

En esta tercera clase vimos algo más sobre errores, el manejo de lista y la creación de listas por comprensión. Para cerrar esta clase te pedimos dos cosas:

- Que recopiles las soluciones de los siguientes ejercicios:
 1. El archivo `solucion_de_errores.py` del [Ejercicio 3.1](#) y siguientes.
 2. El archivo `busqueda_en_listas.py` del [Ejercicio 3.6](#) y el [Ejercicio 3.7](#).
 3. El archivo `invlista.py` del [Ejercicio 3.8](#).
 4. El archivo `propaga.py` del [Ejercicio 3.9](#).
 5. El archivo `arboles.py` sobre arbolado porteño y comprensión de listas incluyendo el [Ejercicio 3.19](#) y el [Ejercicio 3.20](#).
- Que completes [este formulario](#) 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! Nos vemos en la próxima clase.

4. Aleatoriedad

En esta clase introducimos conceptos y técnicas relacionadas al azar. La aleatoriedad puede ser un gran aliado para realizar cálculos de fenómenos estocásticos pero también de fenómenos deterministas. Introducimos los números pseudoaleatorios, así como el módulo `numpy` y sus métodos más sencillos.

Presentamos una versión libre de un ejercicio lindísimo del curso [Exactas Programa](#) que te guía para que resuelvas la pregunta ¿cuántos paquetes de figuritas tengo que comprar para llenar un álbum? con un enfoque estadístico usando el [método de Montecarlo](#).

Introducimos los `arrays` de la biblioteca `numpy` en una sección un poco técnica pero importante para el futuro.

A lo largo de la clase iremos haciendo nuestros primeros gráficos en Python. Cerramos introduciendo los scatterplots que permiten visualizar dos variables en simultaneo y facilitan el análisis exploratorio de datos.

- [4.1 Debuggear programas](#)
- [4.2 Random](#)
- [4.3 NumPy](#)
- [4.4 El album de Figuritas](#)
- [4.5 Gráficos del Arbolado porteño](#)
- [4.6 Cierre de la cuarta clase](#)

4.1 Debuggear programas

Python tiene un debugger poderoso que te permite probar porciones de código. Esto es sencillo y está integrado en IDEs como Spyder.

Vimos en la [Sección 3.2](#) diferentes ejemplos de problemas que pueden aparecer y tuviste que arremangarte e ingenártelas para resovlerlos a mano. En esta sección vamos a introducir la herramientas *pdb* (Python debugger) que ofrece el lenguaje para resolver este tipo de problemas.

Testear es genial, debuggear es horrible.

Se dice que hay un *bug* (un error) cuando un programa no se comporta como el programador espera o hace algo inesperado. Es muy frecuente que los programas tengan bugs. Después de escribir un fragmento de código por primera vez, es conveniente correrlo algunas veces usando tests que permitan poner en evidencia esos bugs.

Diseñar un conjunto de *tests* adecuado no es una tarea sencilla y es frecuente que queden casos especiales que causen errores inesperados.

Python es un lenguaje interpretado, con tipos de datos dinámicos (una misma variable puede cambiar de tipo, de `int` a `float`, por ejemplo). No tiene un compilador que te alerte sobre inconsistencias de tipos antes de ejecutar el programa. Es bueno usar *buenas prácticas* que minimicen estos potenciales errores pero igual es posible que algunos errores se filtren.

Testear consiste en ejecutar un programa o porción de código en condiciones controladas, con entradas conocidas y salidas predichas de forma de poder verificar si lo que da el algoritmos es lo que esperabas.

La ejecución de un algoritmo puede pensarse como un árbol (el árbol de ejecución del algoritmo, cada condición booleana da lugar a una ramificación del árbol). Según la entrada que le des, el programa se va a ejecutar siguiendo una rama u otra. Lo ideal es testear todas las ramas posibles de ejecución y que los casos de prueba (*test cases*) incluyan todos los casos *especiales* (casos como listas vacías, índices apuntando al primer o al último elemento, claves ausentes, etc.) comprobando en cada caso que el programa se comporte según lo esperado.

The screenshot shows the Spyder Python 3.6 IDE interface. The code editor window displays a Python script named 'temp.py' with the following content:

```

1
2 def invertir_lista(lista):
3     invertida = []
4     i=len(lista)
5     while i > 0:
6         invertida.append (lista[i])
7         i=i-1
8     return invertida
9
10 # ESTE código puede ser
11 # ejecutado, guardado en
12 # un archivo y vuelto a leer
13 #
14 #
15 #
16 #
17 #
18 # Usando la misma instancia
19 # del intérprete de la consola
20 # de scripting. ----->
21 #

```

The IPython console window below shows the following interaction:

```

In [1]: # Aquí podemos probar comandos ....
In [2]: invertir_lista([1,2,3,4,5])

```

Los entornos de desarrollo integrado (como el Spyder) dan la posibilidad de combinar el uso de un intérprete de Python con un editor de código y suelen integrar también el uso del debugger. Aún con herramientas como el Spyder, hacer debugging es lento y tedioso. Antes de entrar en los detalles de cómo hacerlo, comentaremos algunos métodos que tratan de reducir su necesidad. Profundizaremos sobre estos métodos más adelante.

Aseveraciones (assert)

El comando `assert` se usa para un control interno del programa. Si la expresión que queremos verificar es `False`, se levanta una excepción de tipo `AssertionError`. La sintaxis de `assert` es la siguiente.

```
assert <expresion> [, 'Mensaje']
```

Por ejemplo

```
assert isinstance(10, int), 'Necesito un entero (int)'
```

La idea *no es* usarlo para comprobar la validez de lo ingresado por el usuario. El propósito de usar `assert` es verificar que ciertas condiciones se cumplan. En general se lo usa mientras el programa está en desarrollo, y luego se los quita o desactiva cuando el programa funciona.

Programación por contratos

Se llama programación por contratos a una forma de programar en la que le programadore define, para cada parte del programa, el tipo y formato de datos con que llamarla y el tipo de datos que devolverá.

Para asegurarse que los tipos de datos sean los esperados, el uso irrestricto de verificaciones puede ayudar en el diseño de software, y detecta tempranamente un error en los datos pasados a una función evitando que se propague.

Por ejemplo: podrías poner verificaciones para cada parámetro de una función.

```
def add(x, y):
    assert isinstance(x, int), 'Necesito un entero (int)'
    assert isinstance(y, int), 'Necesito un entero (int)'
    return x + y
```

De este modo, una función puede verificar que todos sus argumentos sean válidos.

```
>>> add(2, 3)
5
>>> add('2', '3')
Traceback (most recent call last):
...
AssertionError: Necesito un entero (int)
>>>
```

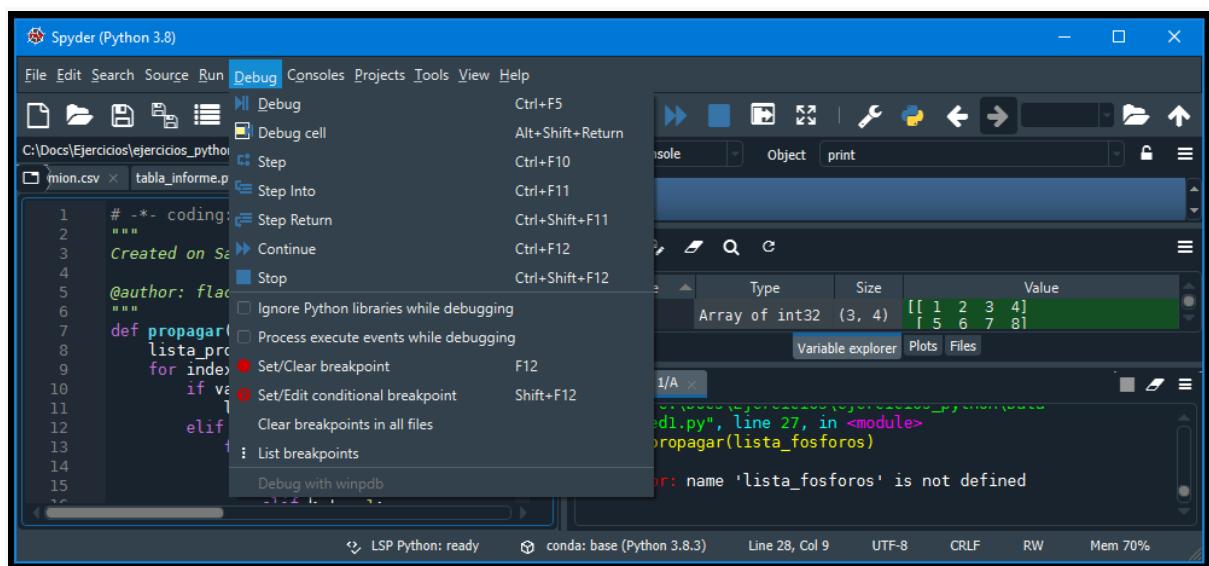
El debugger de Python (pdb)

Es posible usar el debugger de Python directamente en el intérprete (sin interfaz gráfica) para seguir el funcionamiento de un programa. No vamos a entrar en esos detalles acá. Solo mencionamos que la función `breakpoint()` inicia el debugger:

```
def mi_funcion():
    ...
    breakpoint()      # Iniciar el debugger (Python 3.7+)
    ...
```

Podés encontrar instrucciones detalladas [acá](#) sobre como usarlo.

Nos resulta más cómodo usar un IDE como Spyder para hacer debugging y ése es el método que describiremos aquí. Este es el menú desplegable del debugger:



Fijate los nombres de cada ícono:

Nombre	Acción
Debug	inicia el modo debug
Step	da un paso en el programa
Step Into	entra en la función referida
Step Return	ejecuta hasta salir de la función

Continue	retoma la ejecución normal
Stop	detiene el programa

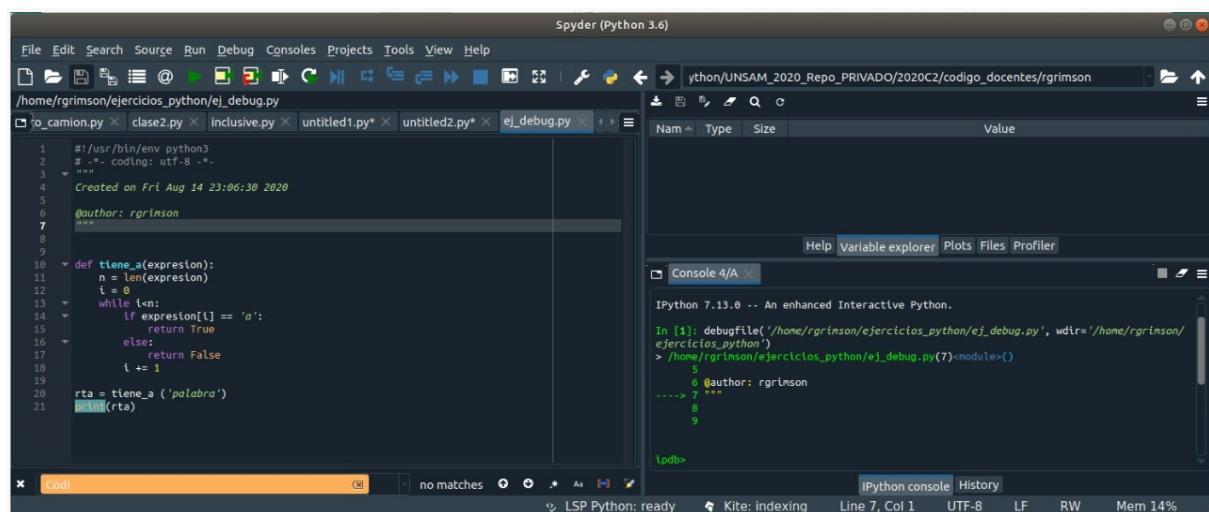
Vamos a volver a analizar el siguiente código, similar al del [Ejercicio 3.1](#) para que veas la utilidad del debugger:

```
def tiene_a(expresion):
    n = len(expresion)
    i = 0
    while i < n:
        if expresion[i] == 'a':
            return True
        else:
            return False
        i += 1

rta = tiene_a ('palabra')
print(rta)
```

Una vez que tengas el código copiado en el Spyder, vamos a ejecutarlo en *modo debug*:

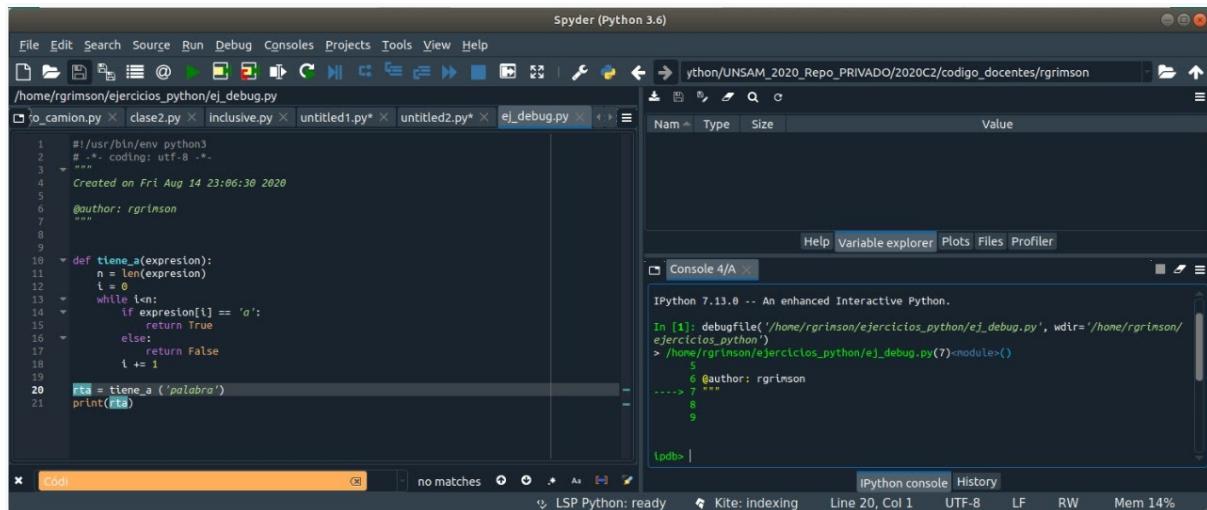
Primero entramos al *modo debug* (Ctrl+F5): El programa queda pausado antes de comenzar. Notá los cambios en la ventana interactiva.



Si damos un paso en el programa: ¿qué va a ocurrir? Debemos tratar de responder esta pregunta antes de avanzar cada paso. *Es nuestra predicción, contrastada con lo que realmente sucede, lo que delata el error.*

Queremos ver la evolución de las variables en la solapa *Variable Explorer* (solapa del centro en el panel superior de la derecha). El programa está en ejecución pero pausado. Sabemos que estamos en *modo debug* por el prompt `(pdb)>` abajo.

Damos algunos pasos (con `Step`, `Ctrl + F10`) hasta llegar a la llamada a la función `tiene_a()` que queremos analizar.



The screenshot shows the Spyder Python 3.6 IDE interface. The code editor on the left displays the file `ej_debug.py` containing the following code:

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4      Created on Fri Aug 14 23:06:38 2020
5
6      @author: rgrimson
7
8
9      def tiene_a(expresion):
10         n = len(expresion)
11         i = 0
12         while i <n:
13             if expresion[i] == 'a':
14                 return True
15             else:
16                 return False
17             i += 1
18
19
20 rta = tiene_a ('palabra')
21 print(rta)
```

The IPython console on the right shows the debugger session:

```
In [1]: debugfile('/home/rgrimson/ejercicios_python/ej_debug.py', wdir='/home/rgrimson/ejercicios_python')
> /home/rgrimson/ejercicios_python/ej_debug.py(7)<module>()
      5
      6 @author: rgrimson
      7
      8
      9
--> 7
(pdb)>
```

The status bar at the bottom indicates "LSP Python: ready", "Kite: indexing", and memory usage "Mem 14%".

Fíjate que el debugger pasó por la línea de definición de la función (y ahora sabe dónde ir a buscarla) pero nunca entró al cuerpo de la función aún. Eso va a ocurrir recién al llamarla.

A esta altura, no queremos simplemente dar un paso (eso ejecutaría la función entera, de una) sino entrar en los detalles de esta función. Para eso usamos `Step Into` (`Ctrl + F11`) de forma de entrar en la ejecución de la función `tiene_a()`. Una vez dentro, seguimos dando pasos (con `Step`, `Ctrl + F10`), siempre pensando qué esperamos que haga la función y observando la evolución de las variables en el explorador de variables. Sigamos así hasta llegar al condicional `if`. Vemos en el *Variable Explorer* que todas las variables internas de la función están definidas y con sus valores asignados.

```

Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
/home/grimson/ejercicios_python/ej_debug.py
/o_camion.py x clase2.py x inclusive.py x untitled1.py* x untitled2.py* x ej_debug.py x
Name Type Size Value
expression str 1 palabra
i int 1 0
n int 1 7
Help Variable explorer Plots Files Profiler
Console 4/A
ipython 7.13.0 -- An enhanced interactive python.
In [1]: debugfile('/home/grimson/ejercicios_python/ej_debug.py', wdir='/home/grimson/ejercicios_python')
> /home/grimson/ejercicios_python/ej_debug.py(7)<module>()
    5
    6 #author: rgrimson
----> 7 """
    8
    9
--Call--
ipdb>

```

Como `i = 0` sabemos que es la primera iteración. Corroboramos que `n=7` ("palabra" tiene 7 letras). En este punto se evalúa `if palabra[i] == 'a':`, y saltaremos a alguna de las dos ramas de ejecución según la evaluación resulte `True` o `False`.

La expresión resulta `False` ya que la primera letra de 'palabra' es la 'p' y no una 'a'. Pero entonces, la siguiente instrucción será el `return False` con lo que saldremos de la función habiendo sólo evaluado la primera letra de la palabra pasada como parámetro. ¿Esto es lo que queríamos?

```

Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
/home/grimson/ejercicios_python/ej_debug.py
/o_camion.py x clase2.py x inclusive.py x untitled1.py* x untitled2.py* x ej_debug.py x
Name Type Size Value
rta bool 1 False
Help Variable explorer Plots Files Profiler
Console 4/A
ipython 7.13.0 -- An enhanced interactive python.
In [1]: debugfile('/home/grimson/ejercicios_python/ej_debug.py', wdir='/home/grimson/ejercicios_python')
> /home/grimson/ejercicios_python/ej_debug.py(7)<module>()
    5
    6 #author: rgrimson
----> 7 """
    8
    9
--Call--
--Return...
ipdb>

```

Acabamos de volver de la función. Las variables internas a la función ya no están visibles (salimos de su alcance o *scope*). El programa sigue en ejecución, en *modo debug*.

Si seguimos dando pasos con `Step` (`Ctrl + F10`) vamos a pasar por el `print()` y terminar la ejecución del programa, saliendo del *modo debug*.

Si, en cambio, al llegar a la línea del `print()` en lugar de `Step` (`Ctrl + F10`) avanzáramos con un `Step Into` (`Ctrl + F11`), entraríamos en los detalles de la

definición de esta función y la cosa se pondría un toque técnica. Cuando esto ocurre es útil usar el Step Return (Ctrl + Shift + F11) para salir de tanto nivel de detalle.

En todo caso, lo que observamos en esta ejecución de `tiene_a()` es que salimos de la función después de haber analizado sólo la primera letra de la palabra. ¿Es correcto esto? ¿Dónde está el error? ¿Cómo lo podemos resolver?

Comentario. Recorrer la ejecución de un programa como un simple expectador no nos muestra claramente un error en el código. Es la incongruencia entre lo esperado y lo que realmente sucede lo que lo marca. Esto exige mucha atención para, antes de ejecutar cada paso, preguntarse: ¿qué espero que ocurra? Luego, al avanzar un paso en la ejecución, puede ocurrir que lo que esperamos que pase no sea lo que realmente pasa. Entonces estamos en un paso clave de la ejecución, que nos marca que estamos frente a una de dos: ó frente a un error en el código ó frente a la oportunidad de mejorar nuestra comprensión del mismo.

Ejercicios

Ejercicio 4.1: Debugger

Ingresá y corré el siguiente código en tu IDE:

```
def invertir_lista(lista):
    '''Recibe una lista L y la devuelve invertida.'''
    invertida = []
    i=len(lista)
    while i > 0:    # tomo el último elemento
        i=i-1
        invertida.append (lista.pop(i))  #
    return invertida

l = [1, 2, 3, 4, 5]
m = invertir_lista(l)
print(f'Entrada {l}, Salida: {m}')
```

Deberías observar que la función modifica el valor de la lista de entrada. Eso no debería ocurrir: una función nunca debería modificar los parámetros salvo que sea lo esperado. Usá el debugger y el explorador de variables para determinar cuál es el primer paso clave en el que se modifica el valor de esta variable.

Ejercicio 4.2: Más debugger

Siguiendo con los ejemplos del [Ejercicio 3.1](#), usá el debugger para analizar el siguiente código:

```
import csv
from pprint import pprint

def leer_camion(nombre_archivo):
    camion = []
    registro = {}
    with open(nombre_archivo, "rt") as f:
        filas = csv.reader(f)
        encabezado = next(filas)
        for fila in filas:
            registro[encabezado[0]] = fila[0]
            registro[encabezado[1]] = int(fila[1])
            registro[encabezado[2]] = float(fila[2])
        camion.append(registro)
    return camion

camion = leer_camion("Data/camion.csv")
pprint(camion)
```

Observá en particular lo que ocurre al leer la segunda fila de datos del archivo y guardarlos en la variable `registro` con los datos ya guardados en la lista `camion`.

Análisis de alternativas para propagar

Los siguientes tres ejercicios proponen diferentes soluciones al [Ejercicio 3.9](#) de propagación del fuego. Vamos a analizar sus diferencias y comenzar a pensar en su eficiencia. Algunas soluciones tienen errores que deberás corregir oportunamente. ¡Usá el debugger de Python!

Observación: Cuando te pidamos que cuentes cuántas operaciones hace una función, no nos va a importar el detalle de las constantes. Por ejemplo: si una función para una entrada de largo n hace $n+2$ operaciones y otra hace $3n+5$ nos va a importar que ambas hacen una cantidad lineal de operaciones en el tamaño de la entrada, pero no las constantes 2, 3 y 5 que figuran en cada caso. Diremos que la cantidad de operaciones es $O(n)$ (se lee 'o de n '). En cambio, sí vamos a hacer una diferencia si una función hace n y otra hace n^2 operaciones (una va a tener complejidad $O(n)$ y la otra $O(n^2)$). Volveremos sobre estos temas más adelante.

Ejercicio 4.3: Propagar por vecinos

El siguiente código propaga el fuego de cada fósforo encendido a sus vecinos inmediatos (si son fósforos nuevos) a lo largo de toda la lista. Y repite esta

operación mientras sea necesario. ¿Te animás a estimar cuántas operaciones puede tener que hacer, en el peor caso?

```
def propagar_al_vecino(l):
    modif = False
    n = len(l)
    for i,e in enumerate(l):
        if e==1 and i<n-1 and l[i+1]==0:
            l[i+1] = 1
            modif = True
        if e==1 and i>0 and l[i-1]==0:
            l[i-1] = 1
            modif = True
    return modif

def propagar(l):
    m = l.copy()
    veces=0
    while propagar_al_vecino(l):
        veces += 1

    print(f"Repetí {veces} veces la función propagar_al_vecino.")
    print(f"Con input {m}")
    print(f"Y obtuve {l}")
    return m
#%%
propagar([0,0,0,0,1])
propagar([0,0,1,0,0])
propagar([1,0,0,0,0])
```

Preguntas:

1. ¿Por qué los tests `l[i+1]==0` y `l[i-1]==0` de la función `propagar_al_vecino` no causan un `IndexError` en los bordes de la lista?
2. ¿Por qué `propagar([0,0,0,0,1])` y `propagar([1,0,0,0,0])`, siendo entradas perfectamente simétricas, no generan la misma cantidad de repeticiones de llamadas a la función `propagar_al_vecino`?
3. Sobre la complejidad. Si te sale, calculá:
 - ¿Cuántas veces como máximo se puede repetir el ciclo while en una lista de largo n ?
 - ¿Cuántas operaciones hace "propagar_al_vecino" en una lista de largo n ?
 - Entonces, ¿cuántas operaciones hace como máximo esta versión de `propagar` en una lista de largo n ? ¿Es un algoritmo de complejidad lineal o cuadrática?

Ejercicio 4.4: Propagar por como el auto fantástico

El siguiente código propaga el fuego inspirado en las luces del **auto fantástico**.

```
def propagar_a_derecha(l):
    n = len(l)
    for i,e in enumerate(l):
        if e==1 and i<n-1:
            if l[i+1]==0:
                l[i+1] = 1
    return l

#%
def propagar_a_izquierda(l):
    return propagar_a_derecha(l[::-1])[::-1]

#%
def propagar(l):
    ld=propagar_a_derecha(l)
    lp = propagar_a_izquierda(ld)
    return lp

#%%
l = [0,0,0,-1,1,0,0,0,-1,0,1,0,0]
print("Estado original: ",l)
print("Propagando...")
lp=propagar(l)
print("Estado original: ",l)
print("Estado propagado: ",lp)
```

Preguntas:

1. ¿Por qué se modificó la lista original?
2. ¿Por qué no quedó igual al `estado propagado`?
3. Corregí el código para que no cambie la lista de entrada.
4. ¿Cuántas operaciones hace como máximo `propagar_a_derecha` en una lista de largo n ?
5. Sabiendo que invertir una lista (`[::-1]`) requiere una cantidad lineal de operaciones en la longitud de la lista ¿Cuántas operaciones hace como máximo `propagar` en una lista de largo n ?

Ejercicio 4.5: Propagar con cadenas

Esta versión usa métodos de *cadenas* para resolver el problema separando los fósforos en *grupos sin fósforos quemados* y analizando cada grupo. Sin embargo algo falla...

```
def trad2s(l):
    '''traduce una lista con 1,0 y -1
    a una cadena con 'f', 'o' y 'x' '''
    d={1:'f', 0 :'o', -1:'x'}
```

```

        s=''.join([d[c] for c in l])
        return s

def trad2l(ps):
    '''traduce cadena con 'f', 'o' y 'x'
    a una lista con 1,0 y -1'''
    inv_d={'f':1, 'o':0, 'x':-1}
    l = [inv_d[c] for c in ps]
    return l

def propagar(l, debug = True):
    s = trad2s(l)
    if debug:
        print(s#, end = ' -> ')
    W=s.split('x')
    PW=[w if ('f' not in w) else 'f'*len(w) for w in W]
    ps=''.join(PW)
    if debug:
        print(ps)
    return trad2l(ps)

#%%
l = [0,0,0,-1,1,0,0,0,-1,0,1,0,0]
lp = propagar(l)
print("Estado original: ",l)
print("Estado propagado: ",lp)

```

Preguntas:

1. ¿Porqué se acorta la lista?
2. ¿Podés corregir el error agregando un solo carácter al código?
3. ¿Te parece que este algoritmo es cuadrático como el [Ejercicio 4.3](#) o lineal como el [Ejercicio 4.4](#)?

4.2 Random

En esta sección veremos algunas de las funciones del módulo `random`. Este módulo se usa para generar valores pseudo-aleatorios. Desde el punto de vista práctico, usaremos estos valores como perfectamente aleatorios --al ser la computadora una máquina determinística sabemos que esto no es completamente cierto. De hecho, en lo que sigue, por simplicidad, omitiremos el prefijo `pseudo` y hablaremos de números aleatorios aunque no lo sean exactamente.

Valores discretos

Podemos generar números enteros aleatorios entre dos extremos. Por ejemplo, para simular la tirada de un dado podemos generar un número entre 1 y 6.

```
import random

dado = random.randint(1,6) # devuelve un entero aleatorio entre 1 y 6
```

Si queremos simular una primera tirada del juego [la generala](#) tendremos que generar cinco valores al azar:

```
import random

tirada=[]
for i in range(5):
    tirada.append(random.randint(1,6))

print(tirada)
```

Ejercicios:

Ejercicio 4.6: Generala servida

Queremos estimar la probabilidad de obtener una generala servida (cinco dados iguales) en una tirada de dados. Podemos hacer la cuenta usando un poco de teoría de probabilidades, o podemos *simular* que tiramos los dados muchas veces y ver cuántas de esas veces obtuvimos cinco dados iguales. En este ejercicio vamos a usar el segundo camino.

Escribí una función `tirar()` que devuelva una lista con cinco dados generados aleatoriamente. Escribí otra función llamada `es_generala(tirada)` que devuelve `True` si y sólo si los cinco dados de la lista `tirada` son iguales.

Luego analizá el siguiente código. Correlo con `N = 100000` varias veces y observá los valores que obtenés. Luego correlo algunas veces con `N = 1000000` (ojo, hace un millón de experimentos, podría tardar un poco):

```
G = sum([es_generala(tirar()) for i in range(N)])
prob = G/N
print(f'Tiré {N} veces, de las cuales {G} saqué generala servida.')
print(f'Podemos estimar la probabilidad de sacar generala servida mediante
{prob:.6f}.')
```

¿Por qué varían más los resultados obtenidos con $N = 100000$ que con $N = 1000000$? ¿Cada cuántas tiradas en promedio podrías decir que sale una generala servida? ¿Cómo se puede calcular la probabilidad de forma exacta?

Ejercicio 4.7: Generala no necesariamente servida

Si uno juega con las reglas originales (se puede volver a tirar algunos de los cinco dados hasta dos veces, llegando hasta a tres tiradas en total) siguiendo una estrategia que intente obtener generala (siempre guardar los dados que más se repiten y tirar nuevamente los demás) es más probable obtener una generala que si sólo consideramos la generala servida. Escribí un programa que estime la probabilidad de obtener una generala en las tres tiradas de una mano y guardalo en un archivo `generala.py`.

Extra: Hay gente que, si en la primera tirada le salen todos dados diferentes, los mete al cubilete y tira los cinco nuevamente. Otras personas, eligen uno de esos dados diferentes, lo guardan, y tiran sólo los cuatro restantes. ¿Podés determinar, por medio de simulaciones, si hay una de estas estrategias que sea mejor que la otra?

Semillas

A veces queremos generar números (pseudo-)aleatorios de una manera reproducible. Puede sonar contradictorio, pero no lo es: es aquí donde se ve claramente la naturalez pseudoaleatoria de estos números. Si fijamos una semilla con el comando `random.seed(semilla)`, donde `semilla` es un número entero, la secuencia de números aleatorios que obtengamos será reproducible utilizando la misma semilla.

Probá por ejemplo correr dos veces el siguiente código:

```
import random
random.seed(31415)

tirada=[]
for i in range(5):
    tirada.append(random.randint(1,6))

print(tirada)
```

Elecciones con reposición

A veces queremos elegir al azar un elemento de una lista y no solo un número. En el caso que vimos recién de los dados, nuestra lista sería `[1, 2, 3, 4, 5, 6]` pero podría ser también `['uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis']`.

La función `random.choice()` toma una secuencia y devuelve un elemento aleatorio.

```
caras = ['uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis']
print(random.choice(caras))
```

Si queremos realizar múltiples elecciones aleatorias de la lista podemos usar la función `random.choices()`

```
print(random.choices(caras, k=5))
```

Estos son experimentos *con reposición* en el sentido de que si en el primer dado sacamos un dos, al tirar el segundo dado podemos sacar otro dos, repitiendo el valor. El término *reposición* viene de pensar en una urna con bolitas. Si un dado lo pensamos como una urna con seis bolitas (etiquetadas del uno al seis), luego de sacar una bolita (tirar el dado una vez) *reponemos* la bolita que sacamos, de forma que en el siguiente experimento (tirar nuevamente el dado) podamos obtener el mismo valor.

Elecciones sin reposición

Si queremos modelar un juego con un mazo de naipes, es natural modelarlo sin reposición. Cuando le damos tres cartas a un jugador la segunda carta no puede ser igual a la primera y la tercera será diferente de las dos anteriores.

En un mazo de naipes españolas, cada carta tiene un palo y un valor. El mazo tiene 40 naipes. Los palos son oro, copa, espada y basto y los valores van del 1 al 7 y de del 10 al 12. Usaremos una comprensión doble de listas para generar los naipes (todas las combinaciones posibles de valores y palos).

```
valores = [1, 2, 3, 4, 5, 6, 7, 10, 11, 12]
palos = ['oro', 'copa', 'espada', 'basto']
naipes = [(valor,palo) for valor in valores for palo in palos]
```

Ahora podemos usar `random.choice(naipes)` para seleccionar un naipe. Sin embargo, si usáramos `random.choices(naipes, k=3)` para seleccionar tres naipes para un jugador, podríamos estar repitiendo el mismo naipe más de una vez, lo que es incorrecto. En este caso tenemos que usar elecciones múltiples *sin reposición*. Para eso usamos la función `sample` del módulo `random`: `random.sample(naipes, k=3)`.

A diferencia de `choices` donde el parámetro `k` podía tomar cualquier valor, al dar la instrucción `random.sample(naipes, k=?)` la variable `k` no puede ser mayor que la cantidad de naipes (es decir 40) ya que no se puede sacar *sin reposición* más elementos que la cantidad total.

Ejercicio 4.8: Envido

Teniendo en cuenta las reglas del [Truco](#), estimá la probabilidad de obtener 31, 32 o 33 puntos de envido en una mano. ¿Son iguales estas tres probabilidades? ¿Por qué?

Observación: como corresponde, en esta materia jugamos al truco sin flor. Si no conocés las reglas del Truco y no te dan ganas de aprenderlo ahora, simplemente salteá este ejercicio.

Guardá este ejercicio en un archivo `envido.py` para entregar.

Mezclar

La última función que queremos introducir es útil en muchos contextos. En los juegos de naipes, para continuar con nuestro ejemplo, es muy usual mezclar el mazo entero antes de repartir. En Python usamos la función `shuffle` del módulo `random`.

```
valores = [1, 2, 3, 4, 5, 6, 7, 10, 11, 12]
palos = ['oro', 'copa', 'espada', 'basto']
naipes = [(valor,palo) for valor in valores for palo in palos]
random.shuffle(naipes)
print(naipes)
```

Observá que la función `shuffle()` modificó la lista que le pasamos como parámetro. Una vez mezclado el mazo, podemos consultar las tres cartas que quedaron al final:

```
naipes[-3:]
```

o directamente sacarlas del mazo:

```
n1 = naipes.pop()
n2 = naipes.pop()
n3 = naipes.pop()
print(f'Repartí el {n1[0]} de {n1[1]}, el {n2[0]} de {n2[1]} y el {n3[0]} de {n3[1]}. Quedan {len(naipes)} naipes en el mazo.')
```

Valores continuos

Además de generar valores (pseudo)aleatorios discretos, también es posible generar valores continuos. La función `random.random()` genera un número de punto flotante entre 0 y 1.

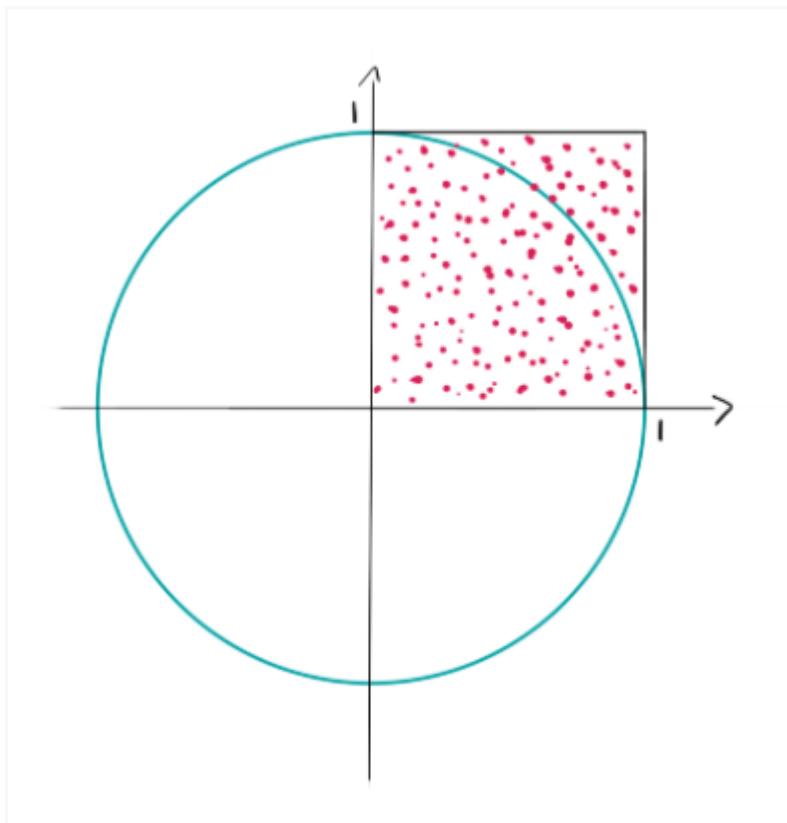
Ejercicio 4.9: Calcular pi

Es interesante ver cómo los algoritmos estocásticos (basados en elecciones aleatorias) también sirven para resolver problemas que no tienen nada de estocásticos. En este ejercicio vas a usar el generador `random()` para aproximar π .

Por definición π es el área del círculo de radio uno. Si generamos puntos (x,y) con:

```
def generar_punto():
    x = random.random()
    y = random.random()
    return x, y
```

tendremos puntos dentro del cuadrado $[0, 1] \times [0, 1]$. Algunos de estos puntos del cuadrado caerán dentro del círculo unitario (los que cumplan que $x^2 + y^2 < 1$) y otros puntos caerán afuera. La proporción de puntos que caigan dentro del cuarto de círculo guardará relación con la proporción entre el área del cuarto de círculo y el área del cuadrado. Obviamente hay una componente aleatoria, pero a medida que la cantidad de puntos crece, la proporción de puntos se acercará a la proporción entre las dos áreas.



Si el área del círculo completo es π , el área de nuestro cuarto de círculo es $\pi/4$. Por otro lado el área del cuadrado unitario es 1. Por lo tanto, si generamos N puntos con una distribución uniforme en el cuadrado unitario, esperamos que $\pi/4$ de estos N puntos caigan dentro del cuarto del círculo y el resto afuera. Es decir que, si llamamos M al número de puntos que caen dentro del círculo, esperamos que $M \sim (\pi/4 * N)$.

Despejando π de esta estimación, obtenemos que $\pi \sim 4*M/N$. Esto nos permite estimar π mirando cuántos puntos caen realmente dentro del círculo del total de puntos.

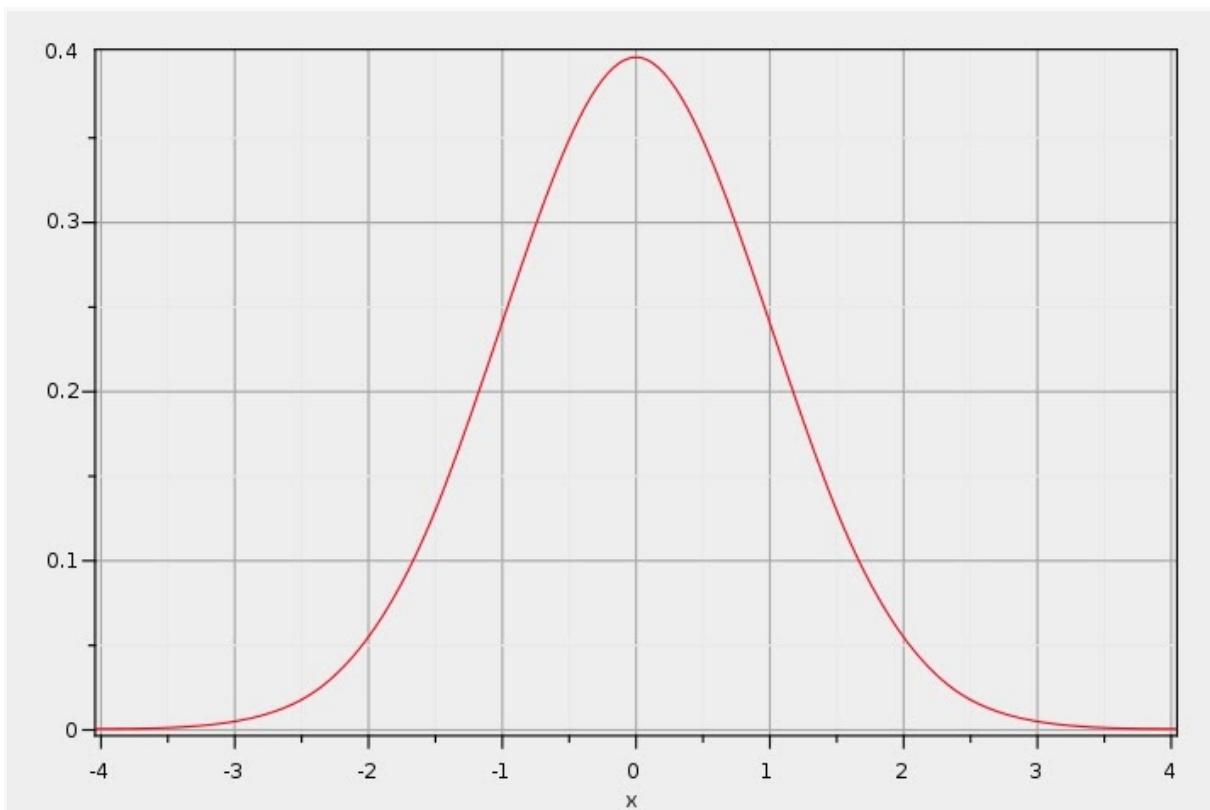
Escribí un programa `estimar_pi.py` que genere cien mil puntos aleatorios con la función `generar_punto()`, calcule la proporción de estos puntos que caen en el círculo unitario (usando `¿x^2 + y^2 < 1?`) y use este resultado para dar una aproximación de π .

Ejercicio 4.10: Gaussiana

Con `random.random()` generamos valores aleatorios entre 0 y 1 con una distribución *uniforme*. En esa distribución, todos los valores posibles tienen la misma probabilidad de ser seleccionados. También es posible generar valores aleatorios

con otras distribuciones. Una de las distribuciones más importantes es la distribución normal o [Gaussiana](#).

La distribución normal tiene dos parámetros, denominados media y desvío estándar y denotados usualmente con las letras griegas *mu* y *sigma*, respectivamente.



La función `random.normalvariate(mu, sigma)` genera números aleatorios según esta distribución de probabilidades. Por ejemplo, usando `mu = 0` y `sigma = 1` podemos generar 6 valores aleatorios así:

```
>>> for i in range(6):
...     print(f'{random.normalvariate(0,1):.2f}', end=', ')
-0.60, 0.06, -1.33, -0.62, -0.81, 0.63,
```

La distribución normal tiene muchos usos. Uno de ellos es modelar errores experimentales, es decir la diferencia entre el valor medido de una magnitud física y el valor real de dicha magnitud.

Hagamos algún ejercicio sencillo antes de terminar. Supongamos que una persona se compra un termómetro que mide la temperatura con un error aleatorio normal con media 0 y desvío estándar de 0.2 grados (error gaussiano). Si la temperatura real de la persona es de 37.5 grados, simulá usando `normalvariate()` (con `mu` y `sigma` adecuados) `n = 99` valores medidos por el termómetro.

Imprimí los valores obtenidos en las mediciones de temperatura simuladas y luego, como resumen, cuatro líneas indicando el valor máximo, el mínimo, el promedio y la mediana de estas n mediciones. Guardá tu programa en el archivo `termometro.py`.

Para encontrar el máximo y mínimo podés usar y agrandar tu código de `busqueda_en_listas.py` o usar las primitivas `max()` y `min()` de Python. El promedio es la suma de los valores dividido su cantidad; podés programarla desde cero o usar la primitiva `sum()` y un cociente por n . Finalmente, la mediana de una cantidad impar de valores es el valor en la posición central cuando los datos están ordenados. Acá podés usar el método `sort()` de listas. Y ya que estamos, ¿se te ocurre cómo encontrar los [cuartiles]?

4.3 NumPy

Esta es una introducción a la biblioteca NumPy (Numerical Python) de Python. Se trata de una colección de módulos de código abierto que tiene aplicaciones en casi todos los campos de las ciencias y de la ingeniería. Es el estándar para trabajar con datos numéricos en Python. Muchas otras bibliotecas y módulos de Python como Pandas, SciPy, Matplotlib, scikit-learn, scikit-image usan numpy.

Esta biblioteca permite trabajar cómodamente con matrices multidimensionales por medio del tipo `ndarray`, un objeto n -dimensional homogéneo (es decir, con todas sus entradas del mismo tipo), y con métodos para operar eficientemente sobre él. Numpy puede usarse para una amplia variedad de operaciones matemáticas sobre matrices. Le agrega a Python estructuras de datos muy potentes sobre las que puedes hacer cálculos y operar matemáticamente con eficiencia y a un alto nivel.

Instalar e importar numpy

Cuando quieras usar numpy en Python, primero tenés que importarlo:

```
import numpy as np
```

Acortamos `numpy` a `np` para ahorrar tiempo y mantener el código estandarizado. Todes escriben `np`.

Si no lo tenés instalado (te dará un error al importarlo) podés instalarlo escribiendo alguno de los siguientes comandos, según corresponda:

```
conda install numpy
```

```
pip install numpy  
pip3 install numpy
```

¿Cuál es la diferencia entre listas y arreglos?

numpy ofrece varias formas muy eficientes de crear vectores y manipular datos numéricos. Mientras que una lista de Python puede contener diferentes tipos de datos en su interior, los elementos de un vector numpy serán todos del mismo tipo. De esta forma numpy garantiza un muy alto rendimiento en las operaciones matemáticas.

Además, los arreglos están pensados para tener un tamaño fijo, mientras que las listas están diseñadas para agregar y sacar elementos. Son estructuras de datos similares desde un punto de vista superficial, pero muy diferentes en cuanto a las posibilidades que brindan.

Las operaciones matemáticas sobre vectores de numpy son más rápidas que sobre listas. Además los vectores ocupan menos memoria que las listas análogas. En cambio, modificar el tamaño de una lista es algo muy sencillo mientras que el de un vector es costoso. Y combinar diferentes tipos de datos es sencillo en las listas pero imposible en los vectores de numpy.

Arreglos n-dimensionales

Los vectores (unidimensionales) y matrices (bidimensiones) se generalizan a arreglos n-dimensionales. Esta estructura de datos es la central de la biblioteca numpy. Un arreglo (`ndarray`) tiene una grilla de valores (datos crudos) junto con información sobre cómo ubicarlos y cómo interpretarlos. Los elementos de esta grilla pueden ser indexados de diversas maneras y, como ya dijimos, son todos del mismo tipo. Este tipo es frecuentemente abreviado como `dtype` (por data type).

Un arreglo puede ser indexado por tuplas de enteros no negativos, por variables booleanas, por otro arreglo o por enteros. El rango (`rank`) de un arreglo es su número de dimensiones. Su forma (`shape`) es una tupla de enteros que dice su tamaño en cada dimensión.

Una forma de inicializar un arreglo de numpy es mediante una lista de números. Esto nos da un vector (arreglo de dimensión uno). Usando listas anidadas, podemos definir arreglos de más altas dimensiones.

Por ejemplo:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
```

o:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

Podemos acceder a los elementos de un arreglo usando corchetes. Acordate que los índices comienzan a contar en 0. Esto significa que si querés acceder al primer elemento, vas a acceder al elemento “0”.

```
>>> print(a[0]) # si tiene múltiples dimensiones, esto me da una "rebanada"  
de una dimensión menos  
[1 2 3 4]  
>>> print(a[2]) # otra rebanada  
[ 9, 10, 11, 12]  
>>> print(a[2][3]) # accedo al cuarto elemento del tercer vector de a  
12  
>>> print(a[2,3]) # o, equivalentemente, accedo al elemento en la tercera  
fila y cuarta columna de a  
12
```

Más información sobre arreglos

Ocasionalmente vas a ver que alguien se refiere a un arreglo como un “ndarray” que es una forma breve de decir arreglo n-dimensional. Un arreglo n-dimensional es simplemente un arreglo con n dimensiones. Recordemos que cuando son unidimensionales los llamamos vectores y si son bidimensionales los llamamos matrices.

¿Qué atributos tiene un arreglo?

Un arreglo es usualmente un contenedor de tamaño fijo de elementos del mismo tipo. Su forma (shape) es una tupla de enteros no negativos que especifica el tamaño del arreglo en cada dimensión. Un arreglo tiene tantas dimensiones como coordenadas en la tupla.

En numpy, las dimensiones se llaman axes (ejes). Esto significa que si tenés una arreglo bidimensional que se ve así:

```
[[0., 0., 0.],  
 [1., 1., 1.]]
```

el arreglo tendrá dos ejes. El primer eje tiene tamaño dos, el segundo tamaño tres (sí, se cuentan primero filas, luego columnas).

De la misma forma que los otros objetos contenedores de Python, los elementos de un arreglo pueden ser accedidos y modificados usando índices y rebanadas.

Crear un arreglo básico

Para crear un arreglo de numpy podés usar la función `np.array()`. Lo único que necesitás es pasarle una lista. Si querés, podés especificar el tipo de datos que querés que tenga.

```
>>> import numpy as np  
>>> a = np.array([1, 2, 3])
```

Vamos a representar la creación con este gráfico:



Ojo, estas visualizaciones son simplificaciones para representar lo que está pasando y darte un entendimiento básico de los conceptos y mecanismos de numpy. Los arreglos y sus operaciones tienen aspectos más complejos que los que quedan capturados en estos dibujitos.

Además de crear una arreglo a partir de una secuencia de elementos, podés crear un arreglo lleno de 0's:

```
>>> np.zeros(2)  
array([0., 0.])
```

O uno lleno de 1's:

```
>>> np.ones(2)  
array([1., 1.])
```

¡O incluso uno no inicializado! La función `empty` crea un arreglo cuyo contenido inicial depende del estado de la memoria. Lo bueno de usar `empty` en lugar de `zeros`

(o `ones`) es la velocidad - al no inicializar los valores no perdemos tiempo. ¡Pero asegurate de ponerle valores con sentido luego!

```
>>> # Crea un arreglo con dos elementos  
>>> np.empty(2)  
array([ 3.14, 42. ]) # puede variar
```

También podés crear vectores a partir de un rango de valores:

```
>>> np.arange(4)  
array([0, 1, 2, 3])
```

También un vector que contiene elementos equiespaciados, especificando el primer número, el límite, y el paso.

```
>>> np.arange(2, 9, 2) # o np.arange(2, 10, 2)  
array([2, 4, 6, 8])
```

El límite derecho nunca está en la lista.

También podés usar `np.linspace()` para crear un vector especificando el primer número, el último número, y la cantidad de elementos:

```
>>> np.linspace(0, 10, num=5)  
array([ 0. , 2.5, 5. , 7.5, 10. ])
```

Ejercicio 4.11: arange() y linspace()

Generá un vector que tenga los números impares entre el 1 y el 19 inclusive usando `arange()`. Repetí el ejercicio usando `linspace()`. ¿Qué diferencia hay en el resultado?

Especificar el tipo de datos

Si no lo especificás, el tipo de datos (por omisión) de los arreglos es el punto flotante (`np.float64`). Sin embargo, podés explicitar otro tipo de datos usando la palabra clave `dtype`.

```
>>> x = np.ones(2, dtype=np.int64)  
>>> x  
array([1, 1])
```

En estos dos casos el 64 de los tipos de datos se refiere a la cantidad de bits usados para representar el número en el sistema binario: 64 bits.

Agregar, borrar y ordenar elementos

Ordenar un vector es sencillo usando `np.sort()`. Por ejemplo, si comenzás con este vector:

```
>>> arr = np.array([2, 1, 5, 3, 7, 4, 6, 8])
```

Podés ordenar sus elementos con:

```
>>> np.sort(arr)
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Fijate que el vector `arr` quedó desordenado. `sort` simplemente devolvió una copia ordenada de los datos pero no modificó el original.

Otra operación usual es la concatenación. Si empezás con estos dos vectores:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([5, 6, 7, 8])
```

los podés concatenar usando `np.concatenate()`.

```
>>> np.concatenate((a, b))
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Un ejemplo un poco más complejo es el siguiente:

```
>>> x = np.array([[1, 2], [3, 4]])
>>> y = np.array([[5, 6]])
```

Los podés concatenar usando:

```
>>> np.concatenate((x, y), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Conocer el tamaño, dimensiones y forma de un arreglo

`ndarray.ndim` te dice la cantidad de ejes (o dimensiones) del arreglo.

`ndarray.shape` te va a dar una tupla de enteros que indican la cantidad de elementos en cada eje. Si tenés una matriz con 2 filas y 3 columnas de va a dar `(2, 3)`.

`ndarray.size` te dice la cantidad de elementos (cantidad de números) de tu arreglo. Es el producto de la tupla `shape`. En el ejemplo del renglón anterior, el `size` es 6.

Por ejemplo, si creás este arreglo de tres dimensiones:

```
>>> array_ejemplo = np.array([[ [0, 1, 2, 3],  
...                               [4, 5, 6, 7]],  
...                               [[0, 1, 2, 3],  
...                               [4, 5, 6, 7]],  
...                               [[0, 1, 2, 3],  
...                               [4, 5, 6, 7]]])
```

Vas a tener

```
>>> array_ejemplo.ndim # cantidad de dimensiones  
3  
>>> array_ejemplo.shape # cantidad de elementos en cada eje  
(3, 2, 4)  
>>> array_ejemplo.size # total de elementos 3*2*4  
24
```

Cambiar la forma de un arreglo

Usando `arr.reshape()` le podés dar una nueva forma a tu arreglo sin cambiar los datos. Solo tené en cuenta que antes y después del reshape el arreglo tiene que tener la misma cantidad de elementos. Por ejemplo, si comenzás con un arreglo con 12 elementos, tendrás que asegurarte que el nuevo arreglo siga teniendo 12 elementos.

Por ejemplo:

```
>>> a = np.arange(6)  
>>> print(a)  
[0 1 2 3 4 5]
```

Podés usar `reshape()` para cambiarle la forma y que en lugar de ser un vector de 6 elementos, sea una matriz de 3 filas y dos columnas:

```
>>> b = a.reshape(3, 2)  
>>> print(b)
```

```
[ [0 1]
 [2 3]
 [4 5] ]
```

Agregar un nuevo eje a un arreglo

A veces pasa que tenemos un vector con n elementos y necesitamos pensarlo como una matriz de una fila y n columnas o de n filas y una columna. Podés usar `np.newaxis` para agregarle dimensiones a un vector existente.

Usando `np.newaxis` una vez podés incrementar la dimensión de tu arreglo en uno. Por ejemplo podés pasar de un vector a una matriz o de una matriz a un arreglo tridimensional, etc.

Por ejemplo, si comenzás con este vector:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
>>> a.shape
(6,)
```

Podés usar `np.newaxis` para agregarle una dimensión y convertirlo en un vector fila:

```
>>> vec_fila = a[np.newaxis, :]
>>> vec_fila.shape
(1, 6)
```

O, para convertirlo en un vector columna, podés insertar un eje en la segunda dimensión:

```
>>> vec_col = a[:, np.newaxis]
>>> vec_col.shape
(6, 1)
```

Índices y rebanadas

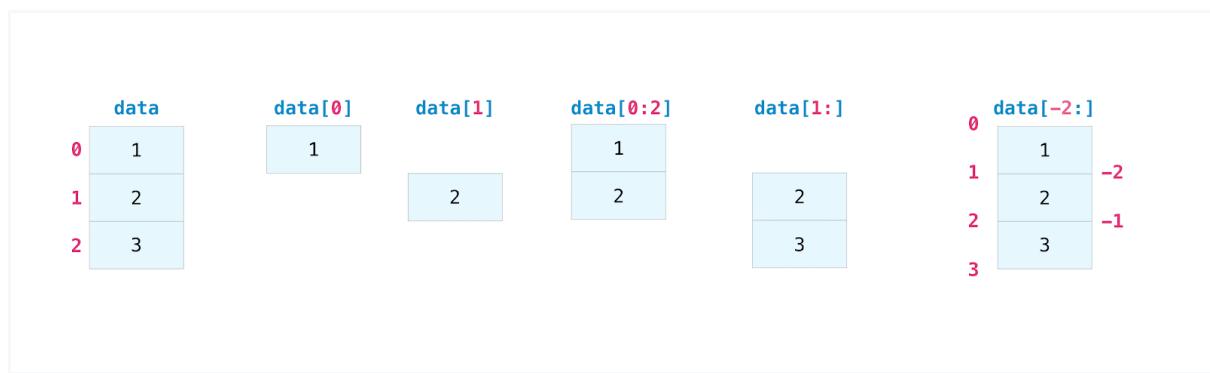
Podés indexar y rebanar arreglos de numpy como hicimos con las listas.

Para obtener elementos de un arreglo, lo más sencillo es usar los índices para seleccionar los que queremos conservar.

```
>>> data = np.array([1, 2, 3])
>>> data[1]
2
>>> data[0:2]
array([1, 2])
```

```
>>> data[1:]  
array([2, 3])  
>>> data[-2:]  
array([2, 3])
```

Lo podés visualizar así:



Otra operación muy útil es seleccionar los elementos que cumplen cierta condición. Por ejemplo, si comenzás con un arreglo así:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

Podés imprimir todos los valores menores que cinco.

```
>>> print(a[a < 5])  
[1 2 3 4]
```

También podés seleccionar, por ejemplo, aquellos elementos mayores o iguales que 5 y usar el resultado para indexar el arreglo.

```
>>> five_up = (a >= 5)  
>>> print(a[five_up])  
[ 5  6  7  8  9 10 11 12]
```

Es interesante que `five_up` da un arreglo de valores booleanos. `True` si satisface la condición y `False` si no la satisface.

Podés seleccionar los elementos pares:

```
>>> pares = a[a%2==0]  
>>> print(pares)  
[ 2  4  6  8 10 12]
```

Usando los operadores lógicos `&` y `|` podés combinar dos o más condiciones.

Ya sea para seleccionar elementos directamente:

```
>>> c = a[(a > 2) & (a < 11)]
>>> print(c)
[ 3  4  5  6  7  8  9 10]
```

o para definir una nueva variable booleana:

```
>>> five_up = (a > 5) | (a == 5)
>>> print(five_up)
[[False False False False]
 [ True  True  True  True]
 [ True  True  True  True]]
```

Finalmente, podés usar `np.nonzero()` para obtener las coordenadas de ciertos elementos de un arreglo.

Si empezamos con este arreglo:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

Podés usar `np.nonzero()` para imprimir los índices de los elementos que son, digamos, menores que 5:

```
>>> b = np.nonzero(a < 5)
>>> print(b)
(array([0, 0, 0, 0]), array([0, 1, 2, 3]))
```

En este ejemplo, la respuesta es una tupla de arreglos: uno por cada dimensión. El primer arreglo representa las filas de los elementos que satisfacen la condición y el segundo sus columnas.

Si querés generar la lista de coordenadas donde se encuentran estos elementos, podés zipear los arreglos, convertir el resultado en una lista e imprimirla:

```
>>> lista_de_coordenadas = list(zip(b[0], b[1]))
```

Surge naturalmente la pregunta: ¿porqué tengo que convertir el objeto `zip` a una lista? Veremos en la segunda mitad de la materia más detalles sobre *generadores* en Python para entender exactamente lo que está pasando aquí. Simplemente digamos que al zipear `b[0]` y `b[1]` no se genera la lista realmente, sino potencialmente. Sólo al solicitar sus elementos (iterando sobre ello o con `list`) se generan realmente las coordenadas.

```
>>> for coord in lista_de_coordenadas:
...     print(coord)
```

```
(0, 0)
(0, 1)
(0, 2)
(0, 3)
```

Podés usar `np.nonzero()` para imprimir o seleccionar los elementos del arreglo que son menores que 5:

```
>>> print(a[b])
[1 2 3 4]
```

Si la condición que ponés no la satisface ningún elemento del arreglo entonces el arreglo de índices que obtenés con `np.nonzero()` será vacío. Por ejemplo:

```
>>> no_hay = np.nonzero(a == 42)
>>> print(no_hay)
(array([], dtype=int64), array([], dtype=int64))
```

Crear arreglos usando datos existentes

Es sencillo crear un nuevo arreglo usando una sección de otro arreglo.

Suponete que tenés este:

```
>>> a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Podés crear otro arreglo a partir de una sección de `a`, simplemente especificando qué parte querés.

```
>>> arr1 = a[3:8]
>>> arr1
array([4, 5, 6, 7, 8])
```

Es importante saber que este método genera una *vista* del arreglo original y no una verdadera copia. Si modificás un elemento de la vista, ¡también se modificará en el original!

```
>>> arr1[0] = 44
>>> print(a)
[ 1  2  3 44  5  6  7  8  9 10]
```

El concepto de *vista* es importante para entender lo que está pasando. Las operaciones más frecuentes devuelven vistas y no copias. Esto ahorra memoria y es más veloz, pero si no lo sabés puede traerte problemas.

Veamos este ejemplo:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

Ahora creamos `b1` a partir de una rebanada de `a` y modificamos su primer elemento. ¡Esto va a modificar el elemento correspondiente de `a` también!

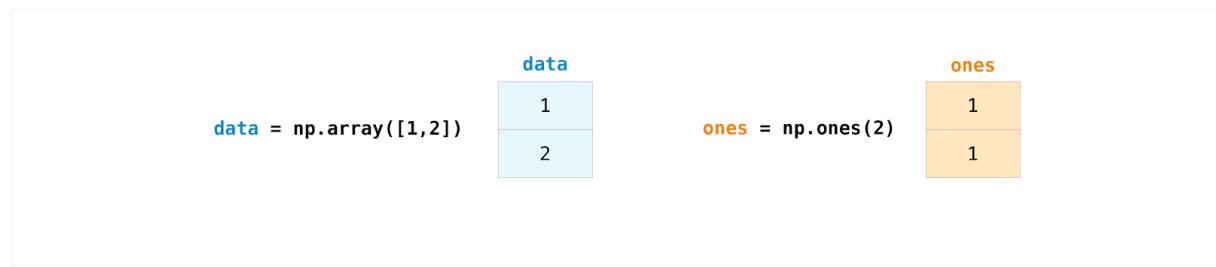
```
>>> b1 = a[0, :]
>>> b1
array([1, 2, 3, 4])
>>> b1[0] = 99
>>> b1
array([99, 2, 3, 4])
>>> a
array([[99, 2, 3, 4],
       [5, 6, 7, 8],
       [9, 10, 11, 12]])
```

Podés usar el método `copy` para copiar los datos. Por ejemplo:

```
>>> b2 = a[1, :].copy()
>>> b2
array([5, 6, 7, 8])
>>> b2[0] = 95
>>> b2
array([95, 6, 7, 8])
>>> a # ¡no se modifica el 5!
array([[99, 2, 3, 4],
       [5, 6, 7, 8],
       [9, 10, 11, 12]])
```

Operaciones básicas sobre arreglos

Una vez que sabés crear arreglos podés empezar a trabajar con ellos. Imaginemos que tenés dos arreglos, uno llamados “data” y otro llamado “ones”.



Podés sumarlos con el signo más.

```
>>> data = np.array([1, 2])
>>> ones = np.ones(2, dtype=int)
```

```
>>> data + ones  
array([2, 3])
```

$$\begin{array}{c} \text{data} \\ \hline 1 \\ 2 \end{array} + \begin{array}{c} \text{ones} \\ \hline 1 \\ 1 \end{array} = \begin{array}{c} 2 \\ 3 \end{array}$$

Obviamente, podés hacer otras operaciones.

```
>>> data - ones  
array([0, 1])  
>>> data * data  
array([1, 4])  
>>> data / data  
array([1., 1.])
```

$$\begin{array}{c} \text{data} \\ \hline 1 \\ 2 \end{array} - \begin{array}{c} \text{ones} \\ \hline 1 \\ 1 \end{array} = \begin{array}{c} 0 \\ 1 \end{array}$$

$$\begin{array}{c} \text{data} \\ \hline 1 \\ 2 \end{array} * \begin{array}{c} \text{data} \\ \hline 1 \\ 2 \end{array} = \begin{array}{c} 1 \\ 4 \end{array}$$

$$\begin{array}{c} \text{data} \\ \hline 1 \\ 2 \end{array} / \begin{array}{c} \text{data} \\ \hline 1 \\ 2 \end{array} = \begin{array}{c} 1 \\ 1 \end{array}$$

Estas operaciones básicas son simples con numpy. Si querés calcular la suma de los elementos del arreglo, podés usar `sum()`. Esto funciona para vectores, matrices y arreglos de dimensión más alta también.

```
>>> a = np.array([1, 2, 3, 4])  
>>> a.sum()  
10
```

Para sumar los valores por fila o por columna en una matriz, simplemente tenés que especificar el eje sobre el que se hará la suma.

Si tenés la matriz:

```
>>> b = np.array([[1, 1], [2, 2]])
```

podés sumar los datos de cada columna con:

```
>>> b.sum(axis=0)  
array([3, 3])
```

y los datos de cada fila usando:

```
>>> b.sum(axis=1)  
array([2, 4])
```

Broadcasting

Hay veces en que necesitás realizar una operación entre un arreglo y un número (en matemática le decimos, un *escalar*). Por ejemplo tenés un vector con distancias en millas (lo llamamos "data") y lo necesitás convertir a distancias en kilómetros. Podés hacer esta operación así:

```
>>> data = np.array([1.0, 2.0])  
>>> data * 1.6  
array([1.6, 3.2])
```

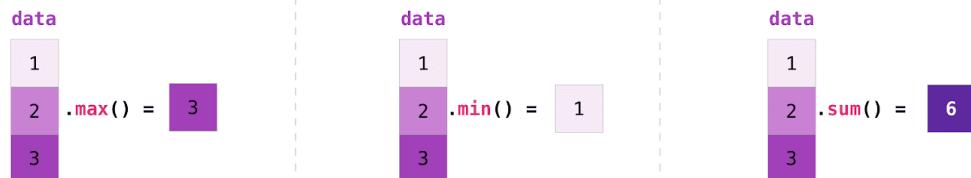
$$\begin{array}{c|c} 1 \\ \hline 2 \end{array} \quad * \quad \textcolor{purple}{1.6} \quad = \quad \begin{array}{c|c} 1 \\ \hline 2 \end{array} \quad * \quad \begin{array}{c|c} 1.6 \\ \hline 1.6 \end{array} \quad = \quad \begin{array}{c|c} 1.6 \\ \hline 3.2 \end{array}$$

numpy entiende que la multiplicación debe ocurrir en cada celda del vector. Este concepto se llama broadcasting. El mecanismo de broadcasting le permite a numpy realizar operaciones en arreglos de diferente tamaño, pero los tamaños deben ser compatibles. Por ejemplo si ambos arreglos tienen el mismo tamaño o si uno tiene tamaño 1 (escalar). Si los tamaños no son compatibles, te va a dar un `ValueError`.

Operaciones un poco más complejas

numpy también te permite realizar operaciones que resumen los datos. Además de `min`, `max`, y `sum`, podés usar `mean` para obtener el promedio, `prod` para calcular el producto, `std` para obtener el desvío estándar de los datos, y más.

```
>>> data.max()  
2.0  
>>> data.min()  
1.0  
>>> data.sum()  
3.0
```



Supongamos que tenemos un arreglo, llamado “a”

```
>>> a = np.array([[0.45053314, 0.17296777, 0.34376245, 0.5510652],
...                 [0.54627315, 0.05093587, 0.40067661, 0.55645993],
...                 [0.12697628, 0.82485143, 0.26590556, 0.56917101]])
```

Es usual procesar los datos por fila o por columna. Si no lo aclarás, numpy procesa los datos de todo el arreglo. Para encontrar la suma o el mínimo del arreglo, usá:

```
>>> a.sum()
4.8595784
```

O:

```
>>> a.min()
0.05093587
```

Podés especificar qué eje querés considerar. Por ejemplo, para calcular el mínimo de cada columna especificás `axis=0`.

```
>>> a.min(axis=0)
array([0.12697628, 0.05093587, 0.26590556, 0.5510652 ])
```

Son cuatro valores, porque la matriz tiene cuatro columnas.

Crear matrices

Podés usar listas de listas para crear arreglos bidimensionales (matrices).

```
>>> data = np.array([[1, 2], [3, 4]])
>>> data
array([[1, 2],
       [3, 4]])
```

```
np.array([[1,2],[3,4]])
```



1	2
3	4

Las técnicas de indexación y rebanadas son muy útiles para manipular matrices:

```
>>> data = np.array([[1, 2], [3, 4], [5, 6]])
>>> data[0, 1]
2
>>> data[1:3]
array([[3, 4], [5, 6]])
>>> data[0:2, 0]
array([1, 3])
```

data	data[0,1]	data[1:3]	data[0:2,0]
0 1 1 3 4 2 5 6			
0 1 1 3 4 2 5 6			
0 1 1 3 4 2 5 6			

Podés procesar los datos de matrices como lo hicimos con vectores:

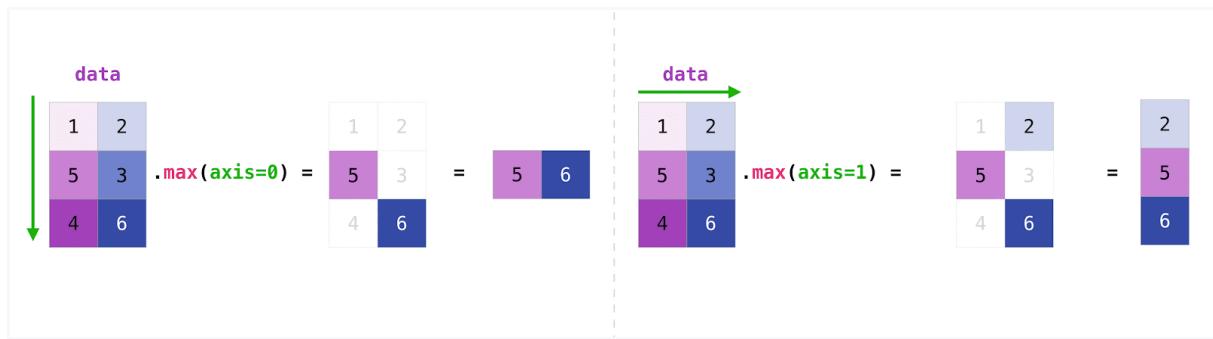
```
>>> data.max()
6
>>> data.min()
1
>>> data.sum()
21
```

data	data	data
1 2 3 4 5 6 .max() = 6	1 2 3 4 5 6 .min() = 1	1 2 3 4 5 6 .sum() = 21

Podés procesar todos, o hacerlo por fila o por columna usando el parámetro `axis`:

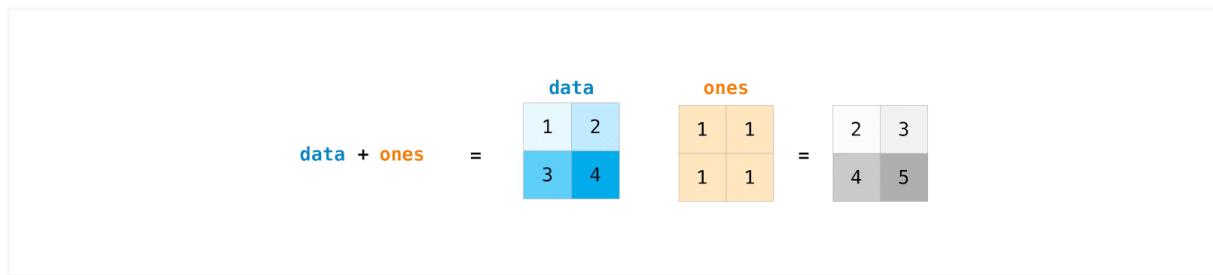
```
>>> data = np.array([[1, 2], [5, 3], [4, 6]])
>>> data.max(axis=0)
array([5, 6])
```

```
>>> data.max(axis=1)
array([2, 5, 6])
```



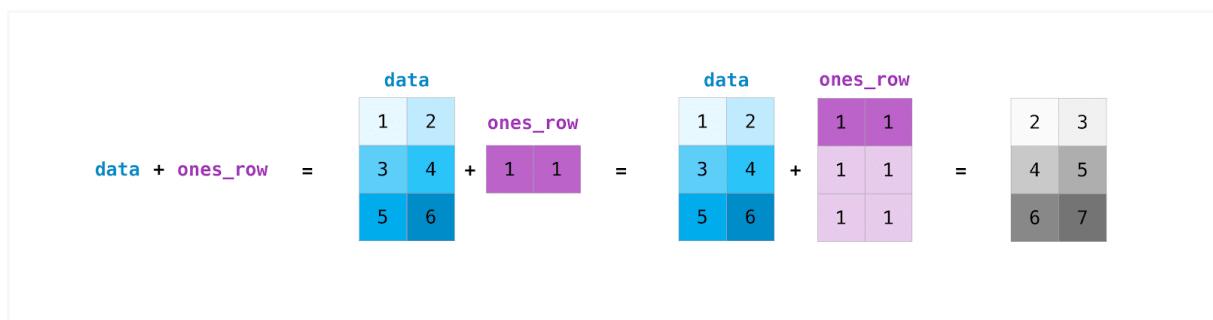
Una vez que tenés tus matrices creadas, podés hacer aritmética con pares de matrices del mismo tamaño.

```
>>> data = np.array([[1, 2], [3, 4]])
>>> ones = np.array([[1, 1], [1, 1]])
>>> data + ones
array([[2, 3],
       [4, 5]])
```



También se pueden sumar matrices de tamaños diferentes, pero sólo si una de ellas tiene una sola fila o una sola columna. En este caso, numpy va a usar las reglas de *broadcast* para la operación.

```
>>> data = np.array([[1, 2], [3, 4], [5, 6]])
>>> ones_row = np.array([[1, 1]])
>>> data + ones_row
array([[2, 3],
       [4, 5],
       [6, 7]])
```



Tené en cuenta que cuando numpy imprime arreglos n-dimensionales, el último eje se itera más rápido y el primero más lento. Por ejemplo:

```
>>> np.ones((4, 3, 2))
array([[[1., 1.],
       [1., 1.],
       [1., 1.]],

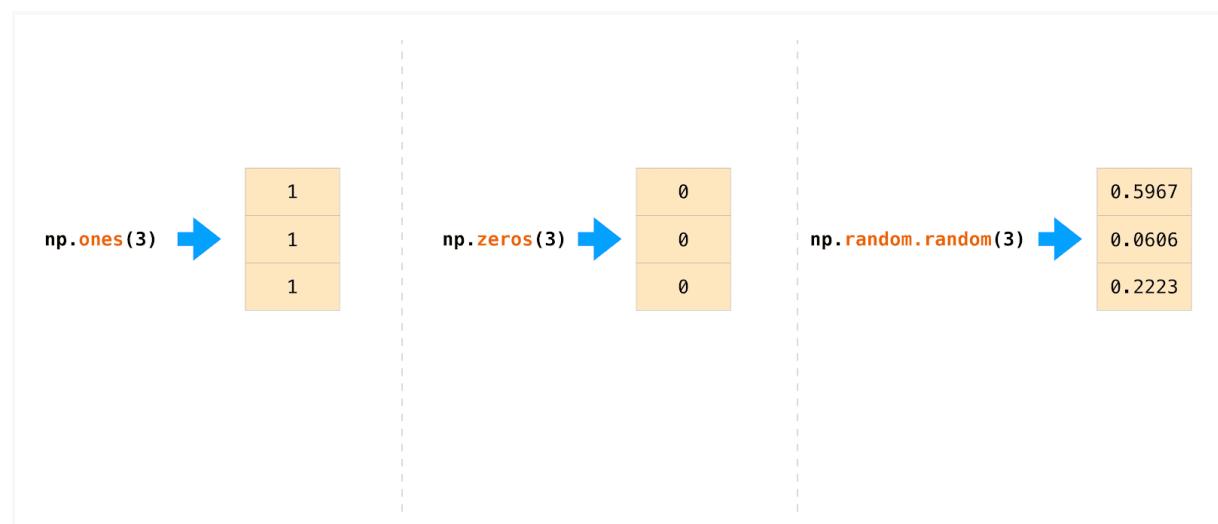
      [[1., 1.],
       [1., 1.],
       [1., 1.]],

      [[1., 1.],
       [1., 1.],
       [1., 1.]],

      [[1., 1.]]])
```

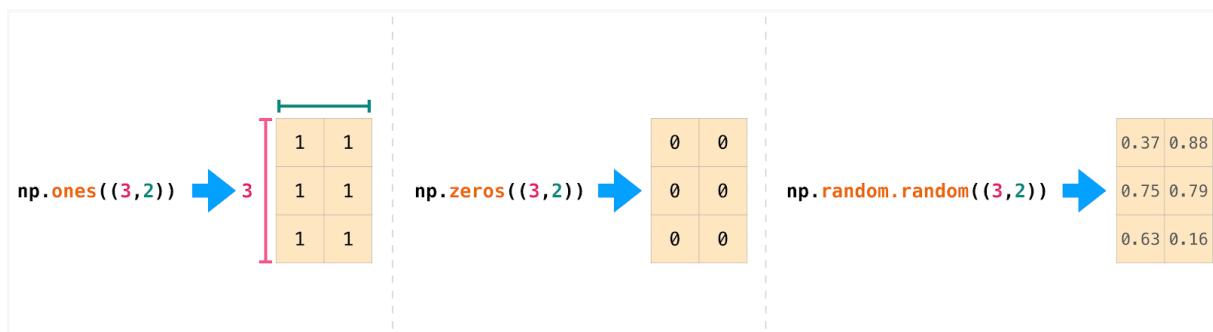
Es frecuente que queramos inicializar los valores de una matriz. numpy ofrece las funciones `ones()` y `zeros()`, así como también la clase `random.Generator` que genera número aleatorios. Sólo hay que pasarle la cantidad de elementos que queremos generar:

```
>>> np.ones(3)
array([1., 1., 1.])
>>> np.zeros(3)
array([0., 0., 0.])
>>> rng = np.random.default_rng(0)
>>> rng.random(3)
array([0.63696169, 0.26978671, 0.04097352]) # puede variar
```



También podés usar `ones()`, `zeros()`, y `random()` para crear matrices, si le pasás una tupla describiendo la forma de la matriz:

```
>>> np.ones((3, 2))
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
>>> np.zeros((3, 2))
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> rng.random((3, 2))
array([[0.01652764, 0.81327024],
       [0.91275558, 0.60663578],
       [0.72949656, 0.54362499]]) # puede variar
```



Esta idea se generaliza a dimensiones más altas.

Fórmulas matemáticas

La facilidad para implementar fórmulas matemáticas sobre un arreglo es una de las características de numpy que lo hacen tan ampliamente usado en la comunidad científica de Python.

Por ejemplo, ésta es la fórmula del error cuadrático medio:

$$\text{MeanSquareError} = \frac{1}{n} \sum_{i=1}^n (Y_{\text{prediction}_i} - Y_i)^2$$

Implementar esta fórmula es simple y directo con numpy:

```
error = (1/n) * np.sum(np.square(predictions - labels))
```

Lo genial de esta abstracción es que `predictions` y `labels` puede tener uno o mil valores. Sólo tienen que tener el mismo tamaño para que todo funcione.

Lo podés visualizar así:

	predictions	labels
	1	1
	1	2
	1	3

`error = (1/3) * np.sum(np.square(`

En este ejemplo, tanto las predicciones como las etiquetas tienen tres valores. Es decir `n` vale tres. Luego de hacer la resta los valores se elevan al cuadrado. Luego numpy suma los valores, divide por tres, y el resultado es el error de esa predicción y puede usarse como un *puntaje* que mide la calidad del modelo que predice.

`error = (1/3) * np.sum(np.square(`

0
-1
-2

`error = (1/3) * np.sum(`

0
1
4

`error = (1/3) * 5`

Guardar y cargar objetos de numpy

Si seguís usando Python después de este curso, es muy probable que en cierto punto quieras guardar tus matrices (o arreglos n-dimensionales) para cargarlas en otro momento sin tener que volver a correr el código que las genera. Hay un par de formas de guardar objetos de numpy. Los objetos ndarray pueden guardarse y leerse de disco con las funciones `loadtxt` y `savetxt` usando archivos de texto

(tienen la ventaja de que los podés ver con un editor de textos como el [sublime](#) o [geany](#)), y con las funciones `load` y `save` que guardan archivos binarios con extensión `.npy`. Los archivos `.npy` guardan los datos, la forma, el tipo del arreglo y otra información necesaria que permiten reconstruirlos correctamente, incluso en otra máquina con otra arquitectura.

Es sencillo guardar un arreglo con `np.save()`. Solo asegurate de especificar el arreglo que querés guardar y el nombre del archivo. Por ejemplo, si creás este vector:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
```

Lo podés guardar en “filename.npy” con:

```
>>> np.save('filename', a)
```

Y lo podés cargar con `np.load()` para reconstruir tu vector.

```
>>> b = np.load('filename.npy')
```

Para verificarlo, usá:

```
>>> print(b)
[1 2 3 4 5 6]
```

En formato de texto plano, lo podés guardar como `.csv` o `.txt` con `np.savetxt`.

Por ejemplo, si tenés este vector:

```
>>> csv_arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
```

lo podés guardar en un archivo `.csv` con nombre “`new_file.csv`” así:

```
>>> np.savetxt('new_file.csv', csv_arr)
```

Y lo podés cargar fácilmente usando `loadtxt()`:

```
>>> np.loadtxt('new_file.csv')
array([1., 2., 3., 4., 5., 6., 7., 8.])
```

Las funciones `savetxt()` y `loadtxt()` aceptan parámetros adicionales para especificar el encabezado y los delimitadores. Si bien los archivos de texto son sencillos para compartir, los archivos `.npy` (y `.npz`) son más pequeños y se leen más rápidamente.

Ejercicio 4.12: Guardar temperaturas

Ampliá el código de `termometro.py` que escribiste en el [Ejercicio 4.10](#) para que guarde el vector con las temperaturas simuladas en el directorio `Data` de tu carpeta de ejercicios, en un archivo llamado `Temperaturas.npy`. Hacé que corra 999 veces en lugar de solo 99.

Ejercicio 4.13: Empezando a plotear

En un rato vamos a empezar a hacer gráficos con Python. Aquí solo un botón de muestra.

Escribí un archivo `plotear_temperaturas.py` que lea el archivo de datos `Temperaturas.npy` con 999 mediciones simuladas que creaste recién y, usando el siguiente ejemplo, hacé un histograma de las temperaturas simuladas:

```
import matplotlib.pyplot as plt  
plt.hist(temperaturas,bins=25)
```

Ajustá la cantidad de *bins* para que el gráfico se vea lo mejor posible.

4.4 El album de Figuritas

Las figuritas del mundial

Esta es una adaptación de una actividad que diseñaron nuestros colegas de Exactas-Programa y amablemente nos dejaron usar aquí.

El objetivo de esta actividad es hacer un programa en Python que responda la pregunta: ¿Cuántas figuritas hay que comprar para completar el álbum del Mundial? Guardá todo lo que hagas en un archivo `figuritas.py`, te lo vamos a pedir al finalizar la clase.



Esta pregunta es noticia cada cuatro años:

- Mundial de Brasil 2014
- Mundial de Rusia 2018
- Incluso hay un paper que salió referido en el diario

Datos:

1. Álbum con 670 figuritas.
2. Cada figurita se imprime en cantidades iguales y se distribuye aleatoriamente.
3. Cada paquete trae cinco figuritas.

Vamos a utilizar este disparador para presentar conceptos clave.

Herramientas útiles de Python

Para que estén disponibles más funciones de Python, tenés que usar el comando `import`. En particular, en esta actividad vamos a usar dos módulos:

- El módulo `random` lo vamos a importar con el comando `import random` y lo vamos a usar para generar figuritas (pseudo) aleatoriamente.
- El módulo `numpy` lo vamos a importar con el comando `import numpy as np` y lo vamos a usar para operar numéricamente.

El modelo del álbum de figuritas

Vamos a representar un álbum de n figuritas utilizando un vector de NumPy con posiciones numeradas de 0 a $n-1$. Cada posición representa el estado de una figurita con dos valores: 0 para indicar que aún no la conseguimos y 1 para indicar que sí (o, si preferís, podés usar un número positivo para representar cuántas de esas figus tenés, contando repes).

Por ejemplo, si tuviéramos un álbum de seis figuritas vacío lo vamos a representar como `[0 0 0 0 0 0]`. Cuando consigamos la figurita 3 tendremos que indicarlo poniendo un 1 en el tercer lugar de la lista, es decir `album[2]=1` y el álbum nos va a quedar `[0 0 1 0 0 0]`, y si queremos representar que nos tocó dos veces la figurita 3, asignamos `album[2] += 1` y el álbum queda `[0 0 2 0 0 0]`.

Primera simplificación

Suponé por ahora que las figuritas se compran individualmente (de a una, no en un paquete con cinco). En este caso, la dinámica del llenado es la siguiente:

- Iniciamos con un álbum vacío y sin haber comprado ninguna figurita.
- Compramos figuritas (de a una) hasta llenar el álbum; es decir, se repite la acción (*el paso*) de comprar y pegar figuritas *mientras* (while) el álbum está incompleto.
- Al terminar nos interesa saber cuántas figuritas tuvimos que comprar para llenar el álbum.

Ejercicios con figus sueltas

Vamos ahora a implementar computacionalmente este modelo. Queremos definir las funciones:

Ejercicio 4.14: Crear

Implementá la función `crear_album(figus_total)` que devuelve un álbum (vector) vacío con `figus_total` espacios para pegar figuritas.

Ejercicio 4.15: Incompleto

¿Cuál sería el comando de Python que nos dice si hay al menos un cero en el vector que representa el álbum? ¿Qué significa que haya al menos un cero en nuestro vector?

Implemente la función `album_incompleto(A)` que recibe un vector y devuelve `True` si el vector contiene el elemento `0`. En el caso en que no haya ceros debe devolver `False`.

Estas funciones son tan sencillas --cada una puede escribirse en una sola línea-- que podría ponerse directamente esa línea cada vez que queremos llamar a la función. Sin embargo, en esta etapa nos parece conveniente que organices el código en funciones, por más que sean sencillas.

Ejercicio 4.16: Comprar

Alguna de las funciones que introdujimos en la [Sección 4.2](#) sirve para devolver un número entero aleatorio dentro de un rango (*¿cuál era?*). Implementá una función `comprar_figu(figus_total)` que reciba el número total de figuritas que tiene el álbum (dado por el parámetro `figus_total`) y devuelva un número entero aleatorio que representa la figurita que nos tocó.

Ejercicio 4.17: Cantidad de compras

Implementá la función `cuantas_figus(figus_total)` que, dado el tamaño del álbum (`figus_total`), genere un álbum nuevo, simule su llenado y devuelva la cantidad de figuritas que se debieron comprar para completarlo.

Ejercicio 4.18:

Ejecutá `n_repeticiones = 1000` veces la función anterior utilizando `figus_total = 6` y guardá en una lista los resultados obtenidos en cada repetición. Con los resultados obtenidos estimá cuántas figuritas hay que comprar, en promedio, para completar el álbum de seis figuritas.

Ayuda: El comando `np.mean(1)` devuelve el promedio de la lista `l`.

¿Podés crear esta lista usando una comprensión de listas?

Ejercicio 4.19:

Calculá `n_repeticiones=100` veces la función `cuantas_figus(figus_total=670)` y guardá los resultados obtenidos en cada repetición en una lista. Con los resultados obtenidos estimá cuántas figuritas hay que comprar, en promedio, para completar el álbum (de 670 figuritas).

Guardá todo lo que hiciste hasta aquí sobre figuritas en un archivo `figuritas.py`. Lo que sigue profundiza un poco más en el asunto.

Ahora con paquetes

Estos ejercicios te recomendamos que los pienses y discutas con un compañero o alguna de tus otras personalidades (si es que tenés):

1. ¿Cómo impacta en lo realizado tener paquetes con figuritas en lugar de figus sueltas?
2. ¿Cómo puede representarse un paquete?

Ejercicios con paquetes

Ejercicio 4.20:

Simulá la generación de un paquete con cinco figuritas, sabiendo que el álbum es de 670. Tené en cuenta que, como en la vida real, puede haber figuritas repetidas en un paquete.

Ejercicio 4.21:

Implementá una función `comprar_paquete(figus_total, figus_paquete)` que, dado el tamaño del álbum (`figus_total`) y la cantidad de figuritas por paquete (`figus_paquete`), genere un paquete (vector) de figuritas al azar.

Ejercicio 4.22:

Implementá una función `cuantos_paquetes(figus_total, figus_paquete)` que dado el tamaño del álbum y la cantidad de figus por paquete, genere un álbum nuevo, simule su llenado y devuelva cuántos paquetes se debieron comprar para completarlo.

Ejercicio 4.23:

Calculá `n_repeticiones = 100` veces la función `cuantos_paquetes`, utilizando `figus_total = 670, figus_paquete = 5`. Guarda los resultados obtenidos en una lista y calculá su promedio. Si te da la compu, hazlo con 1000 repeticiones.

Gráficar el llenado del álbum

El siguiente código usa las funciones que hiciste antes para graficar la curva de llenado de un álbum a medida que comprás paquetes de figuritas. Es un primer ejemplo de gráfico de líneas. En las próximas clases estudiaremos los detalles sobre gráficos de una manera sistemática. Por ahora solo un botón de muestra.

```
def calcular_historia_figus_pegadas(figus_total, figus_paquete):
    album = crear_album(figus_total)
    historia_figus_pegadas = [0]
    while album_incompleto(album):
        paquete = comprar_paquete(figus_total, figus_paquete)
        while paquete:
            album[paquete.pop()] = 1
        figus_pegadas = (album>0).sum()
        historia_figus_pegadas.append(figus_pegadas)
    return historia_figus_pegadas

figus_total = 670
figus_paquete = 5

plt.plot(calcular_historia_figus_pegadas(figus_total, figus_paquete))
plt.xlabel("Cantidad de paquetes comprados.")
plt.ylabel("Cantidad de figuritas pegadas.")
plt.title("La curva de llenado se desacelera al final")
plt.show()
```

Ejercicios un toque más estadísticos:

Los siguientes ejercicios suponen algunos conceptos un poco más avanzados de estadística. Son optativos pero interesantes.

Ejercicio 4.24:

Utilizando lo implementado en el ítem anterior, estimá la probabilidad de completar el álbum con 850 paquetes o menos.

Sugerencia: No leas esto antes de hacer el ejercicio. Hacelo primero y luego miralo. En este ejercicio resulta más compacto usar `n_paquetes_hasta_llenar=np.array(lista)` para convertir a vector la lista conteniendo cuántos paquetes compraste en cada experimento hasta llenar el álbum. Trabajar con vectores tiene ventajas. Por ejemplo probá la siguiente instrucción:

```
(n_paquetes_hasta_llenar <= 850).sum()
```

Ejercicio 4.25: Plotear el histograma

Usá un código similar al del [Ejercicio 4.13](#) para hacer un histograma de la cantidad de paquetes que se compraron en cada experimento, ajustando la cantidad de *bins* para que el gráfico se vea lo mejor posible.

Ejercicio 4.26:

Utilizando lo implementado, estimá cuántos paquetes habría que comprar para tener una chance del 90% de completar el álbum.

Ejercicio 4.27:

Repetí suponiendo que no hay figuritas repetidas en un paquete. ¿Cuánto cambian las probabilidades?

Ejercicio 4.28: Cooperar vs competir

Por último, suponé que cinco amigues se juntan y deciden compartir la compra de figuritas y el llenado de sus cinco álbumes solidariamente. Calculá cuántos paquetes deberían comprar si deben completar todos. Hacé 100 repeticiones y compará el resultado con la compra individual que calculaste antes.

Acordate de guardar todo lo que hiciste sobre figuritas en un archivo `figuritas.py`.

4.5 Gráficos del Arbolado porteño

Ploteando datos reales

En esta sección retomamos el dataset del arbolado porteño (`arbolado-en-espacios-verdes.csv`) para hacer algunos gráficos que nos permitan visualizar los datos. Te damos una guía muy elemental sobre cómo hacer esto y un par de punteros a la documentación oficial. Ya esperamos que empieces a poder buscar por tu cuenta la info que falte.

Seguiremos trabajando en el archivo `arboles.py`. Nos basaremos en el trabajo hecho con comprensión de listas la clase pasada. Los siguientes tres ejercicios

hacelos dentro de tres funciones diferentes, guardalas y entregá el archivo arboles.py con estos agregados.

Ejercicio 4.29: Histograma de altos de Jacarandás

Usando tu trabajo en el [Ejercicio 3.19](#), generá un histograma con las alturas de los Jacarandás en el dataset.

Tu código debería verse similar a este:

```
import os
import matplotlib.pyplot as plt
os.path.join('Data', 'arbolado-en-espacios-verdes.csv')
arboleda = leer_arboles(nombre_archivo)
altos = [comprensión de listas]
plt.hist(altos,bins=....)
```

Observación: Spyder tiene opciones para mostrar las figuras dentro de la misma ventana o en una ventana nueva (Tools -> Preferences -> IPython console -> Graphics -> Backend). Te recomendamos generarlas en una ventana nueva. Luego, con `plt.clf()` podés borrar la figura actual y con `plt.figure()` generás una nueva figura por si querés dejar varias abiertas a la vez.

Ejercicio 4.30: Scatterplot (diámetro vs alto) de Jacarandás

En este ejercicio introducimos un nuevo tipo de gráfico: *el gráfico de dispersión o scatterplot*. El mismo usa coordenadas cartesianas para mostrar los valores de dos variables para un conjunto de datos.

En este caso vamos a graficar un punto en el plano (x,y) por cada árbol en el dataset (o para cada arbol de cierta especie). El punto correspondiente a un árbol con diámetro d y altura h será ubicado en la posición $x=d$ y $y=h$. Este tipo de gráfico permite visualizar relaciones o tendencias entre las variables y es muy útil en el análisis exploratorio de datos.

Usando como base tu trabajo del [Ejercicio 3.20](#), vas a generar un scatterplot para visualizar la relación entre diámetro y alto de los Jacarandás del dataset.

Si ya tenés una lista o un vector d con diámetros y otra h con altos, es sencillo hacer un primer scatterplot:

```
import matplotlib.pyplot as plt
plt.scatter(d,h)
```

Algunas recomendaciones:

1. Convertí la lista generada en un `ndarray` de `numpy`, de esa forma podés usar rebanadas para obtener un vector `d` con diámetros y otro `h` con alturas inmediatamente.
2. Mirá algún ejemplo como [este](#) y tratá de entender cómo se usan los parámetros opcionales `s` (de size, tamaño) y `c` (de color) y `alpha` (de transparencia) de la función `matplotlib.pyplot.scatter`.
3. Usando el parámetro `alpha` hacé que el gráfico permita visualizar dónde hay mayor densidad de datos.

¿Ves alguna relación entre el diámetro y el alto de los Jacarandás? ¿Te parece que es una relación lineal o de otro tipo?

Agregale nombres a los ejes y a la figura usando los siguientes comandos:

```
plt.xlabel("diametro (cm)")  
plt.ylabel("alto (m)")  
plt.title("Relación diámetro-alto para Jacarandás")
```

Ejercicio 4.31: Scatterplot para diferentes especies

Ahora vamos a usar la función `medidas_de_especies()` definida en el [Ejercicio 3.21](#).

Comenzando con éste código, hacé tres gráficos como en el ejercicio anterior, uno por cada especie.

```
import os  
import matplotlib.pyplot as plt  
  
os.path.join('Data', 'arbolado-en-espacios-verdes.csv')  
arboleda = leer_arboles(nombre_archivo)  
especies = ['Eucalipto', 'Palo borracho rosado', 'Jacarandá']  
medidas = medidas_de_especies(especies, arboleda)
```

¿Se mantienen las relaciones que viste en el ejercicio anterior para las tres especies? ¿Hay diferencias entre las especies? Para un mismo alto, ¿cuál tiene mayor diámetro (tipicamente)?

Para poder comparar diferentes especies resulta conveniente fijar los límites en los ejes `x` e `y` en las diferentes figuras usando las funciones `xlim()` e `ylim()`. A continuación un ejemplo:

```
plt.xlim(0,30)  
plt.ylim(0,100)
```

Acordate siempre de ponerle título a las figuras y nombres y unidades a los ejes. Guardá los últimos tres ejercicios dentro de tres funciones diferentes en tu archivo `arboles.py`. Te pediremos que lo entregues en la próxima página.

Extra: ¿podés hacer un solo gráfico que muestre dos de estas tres especies en diferentes colores y resulte claro? ¿Y las tres especies?

4.6 Cierre de la cuarta clase

En esta cuarta clase trabajamos con la generación de números (pseudo)aleatorios, el uso de la biblioteca NumPy y algunos ejemplos de aplicación de estos conceptos. También aprendimos a hacer algunos gráficos elementales en Python.

Para cerrar esta clase te pedimos dos cosas:

- Que recopiles las soluciones de los siguientes ejercicios:
 1. El archivo `generala.py` del [Ejercicio 4.7](#).
 2. El archivo `termometro.py` del [Ejercicio 4.10](#) y el [Ejercicio 4.12](#).
 3. El archivo `plotear_temperaturas.py` del [Ejercicio 4.13](#).
 4. El archivo `figuritas.py` abarcando lo hecho con figuritas (al menos) hasta el [Ejercicio 4.19](#).
 5. El archivo `arboles.py` incluyendo al menos el [Ejercicio 4.30](#).
- Que completes [este formulario](#) 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! Nos vemos en la próxima clase.

5. Complejidad y Organización de programas

Hasta aquí aprendimos algunas cosas básicas de Python y escribimos nuestros primeros programa. A medida que escribas programas más grandes, vas a necesitar organizarlos un poco mejor. En esta clase trataremos con mayor detalle cómo escribir funciones y módulos propios.

En la segunda mitad retomamos nuestra discusión sobre algoritmos de búsqueda e introducimos la búsqueda binaria, un algoritmo mucho más eficiente para buscar un elemento en un vector. Discutimos algunos conceptos de la teoría de la complejidad y finalmente haremos unos gráficos para comparar visualmente la cantidad de operaciones que realizan dos métodos de búsqueda.

- [5.1 Scripting](#)
- [5.2 Funciones](#)
- [5.3 Módulos](#)
- [5.4 Búsqueda binaria](#)
- [5.5 Complejidad de algoritmos](#)
- [5.6 Gráficos de complejidad](#)
- [5.7 Cierre de la quinta clase](#)

5.1 Scripting

En esta sección profundizaremos en el proceso de crear scripts en Python.

¿Qué es un script?

Un *script* es un programa que ejecuta una serie de comandos y termina. *Programa* en el sentido clásico de la palabra: una secuencia de eventos. Su traducción literal es guión, como el guión de una película, con introducción, nudo y desenlace.

```
# programa.py  
  
comando1  
comando2  
comando3  
...
```

Hasta aquí mayormente hemos escrito scripts.

Un problema

Cuando hayas escrito un script útil, éste va a comenzar a crecer en funciones y opciones. Vas a querer aplicarlo a otros problemas. Con el tiempo puede convertirse en un programa esencial, pero si no lo cuidás puede convertirse en un lío enorme, en un gran embrollo. Veamos como lo organizamos.

Definir nombres

Los nombres deben estar definidos antes de usarse.

```
def cuadrado (x):  
    return x*x  
  
a = 42  
b = a + 2      # Requiere que 'a' haya sido definido antes.  
  
z = cuadrado (b) # Requiere que 'cuadrado' y 'b' estén definidos.
```

El orden importa. Casi siempre definimos las variables y las funciones al comienzo.

Definir funciones

Es muy útil agrupar todo el código relevante a una misma *tarea* en el mismo lugar. Para eso sirven las funciones.

```
def leer_precios(nombre_archivo):  
    precios = {}  
    with open(nombre_archivo) as f:  
        f_csv = csv.reader(f)  
        for linea in f_csv:  
            precios[linea[0]] = float(linea[1])  
    return precios
```

Una función simplifica las operaciones repetitivas.

```
preciosviejos = leer_precios('preciosviejos.csv')  
preciosnuevos = leer_precios('preciosnuevos.csv')
```

¿Qué es una función?

Una función es una secuencia de comandos, con un nombre.

```
def nombrefunc(args):  
    comando  
    comando  
    ...  
    return resultado
```

Cualquier comando de Python puede usarse dentro de una función.

```
def foo():  
    import math  
    print(math.sqrt(2))  
    help(math)
```

No existen comandos *especiales* en Python (lo cual es muy fácil de recordar).

Dónde definir funciones

En Python podemos *definir* funciones en cualquier orden.

```
def foo(x):
    bar(x)

def bar(x):
    comandos

# OR
def bar(x):
    comandos

def foo(x):
    bar(x)
```

El único requisito es que la función esté definida al momento de ser *usada* (o llamada) durante la ejecución de un programa.

```
foo(3)          # foo tiene que haber sido definida antes
```

El estilo que preferimos es definir funciones desde abajo hacia arriba ("*bottom-up*")

El estilo *Bottom-Up*

Este estilo trata a las funciones como ladrillos. Los ladrillos simples ó más pequeños se definen primero, y luego se usan para armar funciones más complejas.

```
# miprograma.py
def foo(x):
    ...

def bar(x):
    ...
    foo(x)          # Definida antes
    ...

def spam(x):
    ...
    bar(x)          # Definida antes
    ...

spam(42)          # El código que *usa* la función está al final
```

Las funciones complejas se basan en funciones más simples, definidas antes; aunque esto es sólo una cuestión de estilo. Lo único que realmente importa en ése programa es que la llamada a `spam(42)` esté después que la declaración de las funciones que éste invoca. El orden de las definiciones puede variar, siempre que sea anterior a su uso real.

Diseño de funciones

Lo ideal es que una función sea una *caja negra*. Una función debería operar únicamente sobre los parámetros provistos, evitar variables globales y efectos secundarios no esperados. Hay dos conceptos clave: Diseño Modular y Predecibilidad.

Doc-strings

Es buena costumbre incluir documentación en forma de doc-strings. Un doc-string ó "texto de documentación" es texto ubicado en la línea inmediata después del nombre de la función. El doc-string provee información a quien lee la función, pero también se integra con la función `help()`, IDEs y otras herramientas de programación y documentación.

```
def leer_precios(nombre_archivo):
    """
    Lee precios de un archivo de datos CSV con dos columnas.
    La primer columna debe contener un nombre y la segunda un precio.
    """
    precios = {}
    with open(nombre_archivo) as f:
        f_csv = csv.reader(f)
        for row in f_csv:
            precios[linea[0]] = float(linea[1])
    return precios
```

Un doc-string debe ser conciso e indicar qué hace la función. Si es necesario, podés incluir un ejemplo corto de uso y una descripción de los argumentos.

Veremos también la clase que viene que es posible incluir en el doc-string una descripción de lo que se espera que cumplan los parámetros y lo que garantizamos que cumpla la salida (como un contrato).

Notas sobre el tipo de datos

También podés agregar, en la definición de funciones, notas sobre el tipo de datos de los parámetros y de la función.

```
def leer_precios(nombre_archivo: str) -> dict:
    """
    Lee precios de un archivo de datos CSV con dos columnas.
    La primer columna debe contener un nombre y la segunda un precio.
    Devuelve un diccionario {nombre:precio, ...}
    """
    precios = {}
    with open(nombre_archivo) as f:
        f_csv = csv.reader(f)
        for linea in f_csv:
            precios[linea[0]] = float(linea[1])
    return precios
```

Estas notas no modifican al programa y son puramente informativas. Aún así pueden ser usadas por IDEs, comprobadores de código, y otras herramientas.

Aunque `-> dict` indica al programador que la función devuelve un diccionario, es útil anotar en el doc-string la estructura del diccionario devuelto.

Ejercicios

En el [Ejercicio 2.33](#) (o el [Ejercicio 2.32](#)) escribiste un programa llamado `tabla_informe.py` que imprime un informe con el balance de compra y venta de frutas en un camión.

El programa tiene algunas funciones, como:

```
# informe.py
import csv

def leer_camion(nombre_archivo):
    """
    Read a cajon camion file into a list of dictionaries with keys
    name, cajones, and precio.
    """
    camion = []
    with open(nombre_archivo) as f:
        rows = csv.reader(f)
        headers = next(rows)

        for row in rows:
            record = dict(zip(headers, row))
            cajon = {
                'name' : record['name'],
                'cajones' : int(record['cajones']),
                'precio' : float(record['precio'])}
```

```

        'precio' : float(record['precio'])
    }
    camion.append(cajon)
return camion
...

```

Sin embargo había también partes del programa que ejecutaban una serie de cálculos en forma de script. Este código estaba casi al final del programa. Por ejemplo:

...

```

# Output the informe

headers = ('Nombre', 'Cajones', 'Precio', 'Cambio')
print('%10s %10s %10s %10s' % headers)
print(( '-' * 10 + ' ') * len(headers))
for row in informe:
    print('%10s %10d %10.2f %10.2f' % row)
...

```

En el siguiente ejercicio vamos a volver a ese programa y organizarlo mejor usando funciones.

Ejercicio 5.1: Estructurar un programa como una colección de funciones

Volvé a tu programa `tabla_informe.py` y modificalo de modo que todas las operaciones principales, incluyendo cálculos e impresión, sean llevados a cabo por una colección de funciones. Guarda la nueva versión en un archivo `informe_funciones.py`. Más específicamente:

- Creá una función `imprimir_informe(informe)` que imprima el informe.
- Cambiá la última parte del programa de modo que consista sólo en una serie de llamados a funciones, sin ningún cómputo.

Ejercicio 5.2: Crear una función de alto nivel para la ejecución del programa.

Juntá la última parte de tu programa en una única función `informe_camion(nombre_archivo_camion, nombre_archivo_precios)`. Deberías obtener una función que al llamarla como sigue, imprima el informe:

```
informe_camion('Data/camion.csv', 'Data/precios.csv')
```

En su versión final tu programa será una serie de definiciones de funciones seguidos por un único llamado a la función `informe_camion()` (la cual ejecuta todos los pasos que constituyen tu programa).

Cuando tu programa es una única función, es muy simple ejecutarlo con diferentes entradas. Por ejemplo, después de ejecutar tu programa probá estos comandos en modo interactivo:

```
>>> informe_camion('Data/camion2.csv', 'Data/precios.csv')
... mirá el resultado ...

>>> files = ['Data/camion.csv', 'Data/camion2.csv']
>>> for name in files:
    print(f'{name:-^43s}')
    informe_camion(name, 'Data/precios.csv')
    print()

... mirá el resultado ...
>>>
```

Comentario

En Python es muy fácil escribir código en forma de un script relativamente poco estructurado, en el que tenés un archivo que contiene una secuencia de comandos. A la larga, casi siempre es mejor convertir estos scripts en funciones para organizar el código.

En algún momento, si ese script crece, vas a desear haber sido un poco más organizado desde el comienzo. Tratá de organizar tu código en funciones simples. Es un buen principio es que cada función haga una sola cosa sencilla y concreta, que tenga una sola responsabilidad.

5.2 Funciones

Aunque ya hablamos sobre funciones, dimos pocos detalles sobre su funcionamiento a un nivel algo más profundo. En esta sección esperamos completar algunos conceptos como convenciones de uso, alcance (*scope*) y otros temas.

Llamando a una función

Imaginá la siguiente función:

```
def leer_precios(nombre_archivo, debug):
```

...

Podés llamar a la función pasando los argumentos por orden:

```
precios = leer_precios('precios.csv', True)
```

O podés llamarla usando palabras clave (*keywords*):

```
precios = leer_precios(nombre_archivo='precios.csv', debug=True)
```

Argumentos por omisión

Si preferís que un argumento sea opcional (que tenga un valor *por omisión* o *by default*), en ese caso asigne un valor en la definición de la función. Ése será el valor del argumento si llamás a la función sin especificar un valor para ese argumento.

```
def leer_precios(nombre_archivo, debug=False):  
    ...
```

En la declaración de la función podés asignar un valor a un argumento. Entonces, ese argumento será opcional al invocar a esa función y si lo omitís al invocar a la función va a tomar su valor asignado. A ese valor lo llamamos valor por omisión.

```
d = leer_precios('precios.csv')  
e = leer_precios('precios.dat', True)
```

Nota: Todos los argumentos con valores por omisión deben aparecer al final de la lista de argumentos (primero se declaran todos los argumentos no-opcionales)

Si un argumento es opcional, dale un nombre.

Comparemos estos dos estilos de invocar funciones:

```
cortar_datos(data, False, True) # ?????
```

```
cortar_datos(data, ignore_errores=True)
```

```
cortar_datos(data, debug=True)
```

```
cortar_datos(data, debug=True, ignore_errores=True)
```

En la mayoría de los casos los argumentos con nombre hacen al código más claro, más fácil de entender, especialmente si estos argumentos son booleanos, que determinan opciones si-no.

Buenas prácticas de diseño

Compará estas dos formas de declarar una misma función. Para comprender cómo usar la primera, tendríamos que explorar dentro de la función y saber qué significan sus parámetros. Usá siempre nombres cortos para los argumentos, pero con significado.

```
def leer_precios(f, d=False):
    ...
def leer_precios(nombre_archivo, debug=False):
    ...
```

Quien use la función podría elegir llamarla con argumentos nombrados.

```
d = leer_precios('precios.csv', debug=True)
```

Hay herramientas que crean automáticamente documentación sobre el uso de las funciones y sus argumentos. Si los nombres tienen significado, la documentación resulta más clara.

Devolver un resultado

El comando `return` termina la función y devuelve un valor.

```
def cuadrado(x):
    return x * x
```

Si no se define un resultado, o si falta el comando `return`, la función devuelve la constante `None`.

```
def bar(x):
    instrucciones
    return

a = bar(4)           # a = None

# O TAMBIEN...
def foo(x):
    instrucciones    # No hay `return`

b = foo(4)          # b = None
```

Devolver múltiples resultados

Las funciones sólo pueden devolver una cosa. Si necesitás devolver más de un valor, podés armar una tupla con ellos y devolver la tupla.

```
def dividir(a,b):
    c = a // b      # Cociente
    r = a % b      # Resto
    return c, r    # Devolver una tupla con c y r
```

Ejemplo:

```
x, y = dividir(37,5) # x = 7, y = 2
x = dividir(37, 5)   # x = (7, 2)
```

Alcance de variables

En un programa se declaran variables y se les asignan valores. Esto ocurre dentro y fuera de funciones.

```
x = valor # Variable Global

def foo():
    y = valor # Variable Local
```

Las variables declaradas fuera de funciones son "globales". Las variables declaradas dentro de funciones son "locales". A esto se llama el alcance (*scope*) de una variable.

Variables locales

Las variables locales, declaradas dentro de funciones, son privadas.

```
def leer_camion(nombre_archivo):
    camion = []
    for linea in open(nombre_archivo):
        campos = linea.split(',')
        s = (campos[0], int(campos[1]), float(campos[2]))
        camion.append(s)
    return camion
```

En este ejemplo, `nombre_archivo`, `camion`, `linea`, `campos` y `s` son variables locales.

```
>>> cajones = leer_camion('camion.csv')
```

```
>>> campos
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError: name 'campos' is not defined
>>>
```

El error significa: *Error de Nombre: el nombre 'campos' no está definido.*

No hay conflicto entre variables locales y variables declaradas en otras partes (funciones o globales).

Variables globales

Desde cualquier función se puede acceder a las variables globales declaradas en ese mismo archivo.

```
nombre = 'Dave'

def saludo():
    print('Hola', nombre) # Usa la variable global `nombre`
```

Las funciones, sin embargo, no alteran normalmente el valor de una variable global.

```
nombre = 'Dave'

def spam():
    nombre = 'Guido'

spam()
print(nombre) # imprime 'Dave'
```

Aquí hay dos variables diferentes: `nombre` global, que vale '`Dave`', y `nombre` local, declarada dentro de la función `spam()` que vale '`Guido`'. Cambiar una no cambia la otra: al cambiar el valor de `nombre` local, `nombre` global no cambia.

Acordate: Las asignaciones de valores a variables y las declaraciones de variables dentro de funciones son locales.

Modificar el valor de una variable global

Si necesitás modificar el valor de una variable global desde dentro de una función, la variable tiene que estar declarada como `global` dentro de la misma función.

```
nombre = 'Dave'

def spam():
    global nombre
    nombre = 'Spam'
```

```
global nombre
nombre = 'Guido' # Cambia el valor de la variable global
```

Si declaramos `global nombre` dentro de la función, entonces `nombre` fuera de la función `spam()` y dentro de la función `spam()` refieren a la misma variable, y al modificar una de ellas modificás la otra.

La declaración de globalidad de la variable (con la palabra reservada `global`) tiene que aparecer antes del uso de la variable dentro de una función, y la declaración de la variable global fuera de la función debe ocurrir en el mismo archivo que ésta.

Dicho esto, hay que decir también que usar variables globales se considera una mala práctica. Tratá de evitar completamente el uso de `global`. Si tenés una función que depende del estado de una variable global, tu programa es menos modular: no podés reutilizar la función en otro contexto sin agregar una variable global. Si necesitás que una función modifique el estado de algo fuera de esa función, es mejor entonces usar una clase en lugar de una función. Hablaremos de ésto más adelante, en la segunda mitad de la materia.

Pasaje de argumentos

Cuando llamas a una función, los argumentos son los nombres que refieren a los valores que le pasás. Estos valores no son copias de los originales (ver [Sección 3.5](#)). Si le pasás tipos mutables, como listas o diccionarios, la función sí los puede modificar.

```
def foo(items):
    items.append(42)      # Cambia el valor de items

a = [1, 2, 3]
foo(a)
print(a)                  # [1, 2, 3, 42]
```

Fundamental: Las funciones no reciben una *copia* de los argumentos, sino los argumentos mismos.

Reasignar versus modificar

Existe una sutil pero importante diferencia entre *modificar* el valor de una variable y *reasignar* una variable.

Es importante que entiendas esta diferencia.

```
def foo(items):
```

```

items.append(42)      # Modifica el valor de 'items'

a = [1, 2, 3]
foo(a)
print(a)             # [1, 2, 3, 42]

# Versus

def bar(items):
    items = [4,5,6]    # Cambia la variable local 'items' y
    # hace que apunte a otro objeto completamente diferente.

b = [1, 2, 3]
bar(b)
print(b)             # imprime [1, 2, 3]

```

Recordá: reasignar una variable nunca sobreescribe la memoria que ocupaba. Sólo se asocia el nombre de la variable a un nuevo valor.

Ejercicios

Este conjunto de ejercicios te llevan a implementar un programa medianamente complejo. Es no trivial. Hay varios pasos involucrados e implica articular muchos conceptos al mismo tiempo.

La solución que vas a desarrollar involucra sólo unas 25 líneas de código, pero tomate tu tiempo para asegurarte de que entendés cada concepto y cada parte del código por separado.

La parte central del programa `informe_funciones.py` resuelve la lectura de archivos de tipo CSV. Por ejemplo, la función `leer_camion()` lee un archivo que contiene los datos de un camión organizados como filas, y la función `leer_precios()` lee un archivo que contiene precios. En ambas funciones hay una variedad de acciones detallistas y minuciosas, por ejemplo, ambos abren un archivo y lo envuelven con el módulo `csv` y ambos convierten cada uno de los campos a un tipo de datos diferente.

Si tu tarea fuera de verdad leer datos de archivos, entonces querrías limpiar este código un poco, hacerlo más prolífico, y aplicable a un uso más general. Ésa es nuestra intención ahora:

Comenzá este ejercicio creando un nuevo archivo llamado `ejercicios_python/fileparse.py`. Ahí vamos a trabajar.

Nota: En inglés *to parse* significa analizar gramaticalmente (por ejemplo una frase), separándola en sus partes constitutivas. Es un término muy usado en ciencias de la

computación que no tiene una traducción compacta al castellano. Mucha gente usa el anglicismo *parsear* para referirse a esta actividad.

Ejercicio 5.3: Parsear un archivo CSV

Vamos a empezar por el problema simple de leer un archivo CSV para guardar los datos que contiene en una lista de diccionarios. En el archivo `fileparse.py` definí la siguiente función:

```
# fileparse.py
import csv

def parse_csv(nombre_archivo):
    """
    Parsea un archivo CSV en una lista de registros
    """
    with open(nombre_archivo) as f:
        rows = csv.reader(f)

        # Lee los encabezados
        headers = next(rows)
        registros = []
        for row in rows:
            if not row:      # Saltea filas sin datos
                continue
            registro = dict(zip(headers, row))
            registros.append(registro)

    return registros
```

Esta función lee un archivo CSV y arma una lista de diccionarios a partir del contenido del archivo CSV. La función aísla al programador de los múltiples pequeños pasos necesarios para abrir un archivo, "envolverlo" con el módulo `csv`, ignorar líneas vacías, y demás minucias.

(un "wrapper" (*envoltorio*) en programación es una estructura que expone la interfase de un objeto, pero aísla al usuario de los detalles de funcionamiento de ese objeto.)

Probémoslo en tu IDE o con `python3 -i fileparse.py`.

```
>>> camion = parse_csv('Data/camion.csv')

>>> camion
[{'nombre': 'Lima', 'cajones': '100', 'precio': '32.2'}, {'nombre': 'Naranja', 'cajones': '50', 'precio': '91.1'}, {'nombre': 'Caqui', 'cajones': '150', 'precio': '103.44'}, {'nombre': 'Mandarina', 'cajones': '200', 'precio': '51.23'}, {'nombre': 'Durazno', 'cajones': '95', 'precio': '32.34'}]
```

```
'40.37'}, {'nombre': 'Mandarina', 'cajones': '50', 'precio': '65.1'},  
{'nombre': 'Naranja', 'cajones': '100', 'precio': '70.44'}]
```

```
>>>
```

La función hace lo que queríamos, pero no podemos usar los resultados para hacer cálculos porque todos los datos recuperados son de tipo cadena (*string*). Ya vamos a solucionar esto. Por ahora sigamos extendiendo sus funciones.

Ejercicio 5.4: Selector de Columnas

La mayoría de los casos, uno no está interesado en todos los datos que contiene el archivo CSV, sino sólo en algunas columnas. Modifiquemos la función `parse_csv` de modo que permita al usuario elegir (opcionalmente) algunas columnas del siguiente modo:

```
>>> # Lee todos los datos  
  
>>> camion = parse_csv('Data/camion.csv')  
  
>>> camion  
  
[{'nombre': 'Lima', 'cajones': '100', 'precio': '32.2'}, {'nombre':  
'Naranja', 'cajones': '50', 'precio': '91.1'}, {'nombre': 'Caqui',  
'cajones': '150', 'precio': '103.44'}, {'nombre': 'Mandarina', 'cajones':  
'200', 'precio': '51.23'}, {'nombre': 'Durazno', 'cajones': '95', 'precio':  
'40.37'}, {'nombre': 'Mandarina', 'cajones': '50', 'precio': '65.1'},  
{'nombre': 'Naranja', 'cajones': '100', 'precio': '70.44'}]
```

```
>>> # Lee solo algunos datos
```

```
>>> cajones_retenidos = parse_csv('Data/camion.csv',  
select=['nombre', 'cajones'])
```

```
>>> cajones_retenidos
```

```
[{'nombre': 'Lima', 'cajones': '100'}, {'nombre': 'Naranja', 'cajones':  
'50'}, {'nombre': 'Caqui', 'cajones': '150'}, {'nombre': 'Mandarina',  
'cajones': '200'}, {'nombre': 'Durazno', 'cajones': '95'}, {'nombre':  
'Mandarina', 'cajones': '50'}, {'nombre': 'Naranja', 'cajones': '100'}]
```

```
>>>
```

Vimos un ejemplo de un selector de columnas en el [Ejercicio 3.14](#). De todos modos, ésta es otra forma de resolverlo:

```
# fileparse.py  
import csv
```

```

def parse_csv(nombre_archivo, select = None):
    """
    Parsea un archivo CSV en una lista de registros.
    Se puede seleccionar sólo un subconjunto de las columnas, determinando
    el parámetro select, que debe ser una lista de nombres de las columnas a
    considerar.
    """
    with open(nombre_archivo) as f:
        filas = csv.reader(f)

        # Lee los encabezados del archivo
        encabezados = next(filas)

        # Si se indicó un selector de columnas,
        # buscar los índices de las columnas especificadas.
        # Y en ese caso achicar el conjunto de encabezados para
        # diccionarios

        if select:
            indices = [encabezados.index(nombre_columna) for nombre_columna
in select]
            encabezados = select
        else:
            indices = []

        registros = []
        for fila in filas:
            if not fila:    # Saltar filas vacías
                continue
            # Filtrar la fila si se especificaron columnas
            if indices:
                fila = [fila[index] for index in indices]

            # Armar el diccionario
            registro = dict(zip(encabezados, fila))
            registros.append(registro)

    return registros

```

Esta parte es un toque técnica y merece una mirada de más cerca. El paso más delicado es traducir los nombres de las columnas seleccionadas a índices. Por ejemplo, supongamos que los encabezados en el archivo de entrada fueran los siguientes:

```

>>> encabezados = ['nombre', 'dia', 'hora', 'cajones', 'precio']
>>>

```

Y que las columnas seleccionadas fueran:

```
>>> select = ['nombre', 'cajones']
>>>
```

Para hacer la selección correctamente, tenés que convertir los nombres de las columnas listadas en `select` a índices (posiciones) de columnas en el archivo. Esto es exactamente lo que hace este paso:

```
>>> indices = [encabezados.index(nombre_columna) for nombre_columna in
select]
>>> indices
[0, 3]
>>>
```

En otras palabras, "nombre" es la columna 0 y "cajones" es la columna 3. Al leer una línea de datos del archivo, usás los índices para filtrarla y rescatar sólo las columnas que te interesan:

```
>>> linea = ['Lima', '6/11/2007', '9:50am', '100', '32.20']
>>> linea = [linea[indice] for indice in indices]
>>> linea
['Lima', '100']
>>>
```

Ejercicio 5.5: Conversión de tipo

Modificá la función `parse_csv()` de modo que permita, opcionalmente, convertir el tipo de los datos recuperados antes de devolverlos.

```
>>> camion = parse_csv('Data/camion.csv', types=[str, int, float])

>>> camion

[{'nombre': 'Lima', 'cajones': 100, 'precio': 32.2}, {'nombre': 'Naranja',
'cajones': 50, 'precio': 91.1}, {'nombre': 'Caqui', 'cajones': 150,
'precio': 103.44}, {'nombre': 'Mandarina', 'cajones': 200, 'precio':
51.23}, {'nombre': 'Durazno', 'cajones': 95, 'precio': 40.37}, {'nombre':
'Mandarina', 'cajones': 50, 'precio': 65.1}, {'nombre': 'Naranja',
'cajones': 100, 'precio': 70.44}]
```

```
>>> cajones_lote = parse_csv('Data/camion.csv', select=['nombre',
'cajones'], types=[str, int])

>>> cajones_lote
```

```
[{'nombre': 'Lima', 'cajones': 100}, {'nombre': 'Naranja', 'cajones': 50},  
{'nombre': 'Caqui', 'cajones': 150}, {'nombre': 'Mandarina', 'cajones': 200},  
{'nombre': 'Durazno', 'cajones': 95}, {'nombre': 'Mandarina', 'cajones': 50},  
{'nombre': 'Naranja', 'cajones': 100}]
```

```
>>>
```

Ya vimos esto en el [Ejercicio 3.15](#). Vas a necesitar insertar la siguiente porción de código en tu implementación:

```
...  
if types:  
    fila = [func(val) for func, val in zip(types, fila)]  
...
```

Ejercicio 5.6: Trabajando sin encabezados

Algunos archivos CSV no tiene información de los encabezados. Por ejemplo, el archivo `precios.csv` se ve así:

```
Lima,40.22  
Uva,24.85  
Ciruela,44.85  
Cereza,11.27  
...
```

Modificá la función `parse_csv()` de forma que (opcionalmente) pueda trabajar con este tipo de archivos, creando tuplas en lugar de diccionarios cuando no haya encabezados. Por ejemplo:

```
>>> precios = parse_csv('Data/precios.csv', types=[str,float],  
has_headers=False)  
  
>>> precios  
  
[(Lima,40.22), (Uva,24.85), (Ciruela,44.85), (Cereza,11.27),  
(Frutilla,53.72), (Caqui,105.46), (Tomate,66.67), (Berenjena,28.47),  
(Lechuga,24.22), (Durazno,73.48), (Remolacha,20.75), (Habas,23.16),  
(Frambuesa,34.35), (Naranja,106.28), (Bruselas,15.72), (Batata,55.16),  
(Rúcula,36.9), (Radicheta,26.11), (Repollo,49.16), (Cebolla,58.99),  
(Cebollín,57.1), (Puerro,27.58), (Mandarina,80.89), (Ajo,15.19),  
(Rabanito,51.94), (Zapallo,24.79), (Espinaca,52.61), (Acelga,29.26),  
(Zanahoria,49.74), (Papa,69.35)]  
  
>>>
```

Para hacer este cambio, vas a tener que modificar el código de forma que, si le pasás el parámetro `has_headers = False`, la primera línea de datos no sea

interpretada como encabezado. Además, en ese caso, vas a tener que asegurarte de no crear diccionarios, dado que no tenés más los nombres de las columnas para usar en el encabezado. Vale aclarar que este parámetro debe tener como valor por omisión `True`, con lo que la función sigue funcionando igual que antes si no se especifica `has_headers = False`.

Si bien no es difícil, este es un cambio muy grande en esta función. Un camino posible es poner un `if has_headers` al principio y resolver cada caso por separado. Otro camino es poner condicionales en cada paso donde sea necesario operar de manera diferente.

Incorporá todos estos cambios en el archivo `fileparse.py`.

Comentario

Llegaste lejos. Hasta este punto creaste una biblioteca de funciones que es genuinamente útil. La podés usar para parsear archivos CSV de formato arbitrario, eligiendo las columnas relevantes y cambiando el tipo de datos devuelto, todo esto sin tener que preocuparte mucho por el manejo de archivos o entender cómo funciona el módulo `csv`.

5.3 Módulos

En esta sección vamos a introducir conceptos que nos permiten crear módulos y trabajar con programas cuyas partes están repartidas en múltiples archivos.

Módulos y la instrucción `import`

Todos los archivos con código Python son módulos.

```
# foo.py
def grok(a):
    ...
def spam(b):
    ...
```

El comando `import` carga un módulo y lo ejecuta.

```
# programa.py
import foo
```

```
a = foo.grok(2)
b = foo.spam('Hola')
...
```

Namespaces

Se puede decir que un módulo es una colección de valores asignados a nombres. A ésto se lo llama un *namespace* (espacio de nombres). Es el contexto en el cual esos nombres existen: todas las variables globales y las funciones definidas en un módulo *pertenecen* a ese módulo. Una vez importado, el nombre del módulo se usa como un prefijo al nombrar esas variables y funciones. Por eso se llama un namespace.

```
import foo

a = foo.grok(2)
b = foo.spam('Hello')
...
```

El nombre del módulo es el nombre del archivo que lo contiene.

Definiciones globales

El espacio de nombres contiene todo aquello definido con visibilidad *global*. Supongamos dos módulos diferentes que definen cada uno una variable `x`:

```
# foo.py
x = 42
def grok(a):
    ...
# bar.py
x = 37
def spam(a):
    ...
```

Entonces hay dos definiciones de `x` y cada una refiere a una variable diferente. Una de ellas es `foo.x` y la otra es `bar.x`. De este modo, diferentes módulos tienen la libertad de definir variables con el mismo nombre sin que existan ambigüedades ni conflictos.

Los módulos están aislados uno de otro.

Módulos como entornos

Los módulos crean un entorno que contiene a todo el código definido ahí.

```
# foo.py
x = 42

def grok(a):
    print(x)
```

Incluso las variables *globales* son visibles sólo dentro del módulo en que fueron definidas (el mismo archivo). Cada módulo es un pequeño universo.

Ejecución de módulos

Cuando importás un módulo se ejecutan *todas* las instrucciones en ese módulo, una tras otra, hasta llegar al final del archivo. El *namespace* del módulo está poblado por todas las funciones y variables globales cuya definición siga vigente al terminar de ejecutar el módulo. Si existen comandos que se ejecutan en el *namespace* global del módulo y hacen tareas como crear archivos, imprimir mensajes, etc., se van a ejecutar al importar el módulo.

El comando `import as`

En el momento de importar un módulo, podés cambiar el nombre que le asignás dentro del contexto en que lo importás.

```
import math as m
def rectangular(r, theta):
    x = r * m.cos(theta)
    y = r * m.sin(theta)
    return x, y
```

Funciona del mismo modo que un `import` común salvo que, para quien lo importa, el nombre del módulo cambia.

`from módulo import nombre`

Este comando toma ciertos nombres selectos de un módulo, y los hace accesibles localmente.

```
from math import sin, cos

def rectangular(r, theta):
    x = r * cos(theta)
    y = r * sin(theta)
    return x, y
```

Esta forma de importar te permite usar partes de un módulo sin necesidad de especificar la pertenencia a un módulo como prefijo. Es útil para nombres (funciones o variables) que se usan mucho.

Si usás `from math import *` vas a importar *todas* las funciones y constantes del módulo `math` como si estuvieran definidas localmente. No es conveniente hacer esto ya que se pierden las ventajas que da trabajar con namespaces.

Notas sobre `import`

Estas distintas formas de usar `import` *no modifican* el funcionamiento de un módulo.

```
import math
# vs
import math as m
# vs
from math import cos, sin
...
```

Más específicamente, `import` siempre ejecuta el módulo completo, y los módulos siguen siendo pequeños entornos aislados. El comando `import módulo as` sólo cambia el nombre local del módulo. El comando `from módulo import función`, aunque sólo hace accesibles las funciones `sin` y `cos`, de todos modos carga todo el módulo y lo ejecuta. La única diferencia es que copia los nombres de las funciones `sin` y `cos` al namespace local.

Carga de módulos

Cada módulo es cargado y ejecutado sólo *una* vez.

Observación: Repetir la instrucción `import` sólo devuelve una referencia al módulo ya cargado.

La variable `sys.modules` es un diccionario de los módulos cargados.

```
>>> import sys
>>> sys.modules.keys()
['copy_reg', '__main__', 'site', '__builtin__', 'encodings',
'encodings.encodings', 'posixpath', ...]
>>>
```

Precaución: Si cambiás el código de un módulo y lo volvés a cargar sucede algo que suele causar confusión hasta que lo entendés: Dado que existe la lista de módulos cargados `sys.modules`, un pedido de cargar un módulo por segunda vez siempre devolverá el módulo ya cargado, aún si el módulo fue modificado, si se trata de una

versión nueva de ese módulo y si el archivo en disco ha sido modificado. Es posible usar `reload(módulo)` pero sólo en ciertos casos. El método que asegura que el módulo se vuelva a cargar es cerrar y volver a abrir el intérprete de Python.

Ejercicios

Para estos ejercicios que involucran módulos, es de suma importancia que te asegures de que estás ejecutando Python en el directorio adecuado.

Ejercicio 5.7: Importar módulos

En el [Ejercicio 5.3](#) creamos una función llamada `parse_csv()` para parsear el contenido de archivos de datos en formato CSV. Ahora vamos a ver cómo usar esa función en otros programas.

Empezá por asegurarte que el directorio de trabajo es `ejercicios_python/` y que en el mismo tengas tus ejercicios anteriores (como `hipoteca.py` y el archivo `fileparse.py` con la función `parse_csv()` que armaste antes). Los vamos a importar.

Con el directorio de trabajo adecuado (puede que tengas que reiniciar tu intérprete para que tome efecto un cambio), intentá importar los programas que escribiste antes. Con sólo importarlos deberías ver su salida exactamente como cuando los terminaste de escribir.

Repetimos: al importar un módulo ejecutás su código.

```
>>> import rebotes  
... mirá la salida ...  
>>> import hipoteca  
... mirá la salida ...  
>>> import informe  
... mirá la salida ...  
>>>
```

Si nada de esto funciona, es probable que estés ejecutando Python desde la carpeta equivocada.

Ahora probá importar tu módulo `fileparse` y pedile `help`.

```
>>> import fileparse  
>>> help(fileparse)  
... mirá la salida ...  
>>> dir(fileparse)  
... mirá la salida ...  
>>>
```

Intentá usar el módulo para leer datos de un archivo:

```
>>> camion =  
fileparse.parse_csv('Data/camion.csv', select=['nombre', 'cajones', 'precio'],  
types=[str, int, float])  
>>> camion  
... mirá la salida ...  
>>> lista_precios =  
fileparse.parse_csv('Data/precios.csv', types=[str, float],  
has_headers=False)  
>>> lista_precios  
... mirá la salida ...  
>>> precios = dict(lista_precios)  
>>> precios  
... fijate la salida ...  
>>> precios['Naranja']  
106.11  
>>>
```

Importá sólo la función para evitar escribir el nombre del módulo:

```
>>> from fileparse import parse_csv  
>>> camion = parse_csv('Data/camion.csv',  
select=['nombre', 'cajones', 'precio'], types=[str, int, float])  
>>> camion  
... fijate la salida ...  
>>>
```

Ejercicio 5.8: Usemos tu módulo

En el [Ejercicio 5.1](#) escribiste un programa `informe_funciones.py` que produce un informe como éste:

Nombre	Cajones	Precio	Cambio
Lima	100	\$32.20	8.02
Naranja	50	\$91.10	15.18
Caqui	150	\$103.44	2.02
Mandarina	200	\$51.23	29.66
Durazno	95	\$40.37	33.11
Mandarina	50	\$65.10	15.79
Naranja	100	\$70.44	35.84

Retomá ese programa (si lo perdiste, te dejamos una versión para que la leas y la puedas usar) y modificalo de modo que todo el procesamiento de archivos de entrada de datos se haga usando funciones del módulo `fileparse`. Para lograr éso,

```
importá fileparse como un módulo y cambiá las funciones leer_camion() y leer_precios() para que usen la función parse_csv() .
```

Guíate por el ejemplo interactivo que dimos un poco más arriba. Al final, deberías obtener exactamente el mismo resultado que al principio.

Ejercicio 5.9: Un poco más allá

En [Ejercicio 2.5](#) escribiste el programa `costo_camion.py` que lee, mediante una función llamada `costo_camion()` los datos de un camión y calcula su costo.

```
>>> import costo_camion  
>>> costo_camion.costo_camion('Data/camion.csv')  
47671.15  
>>>
```

Modificá el archivo `costo_camion.py` para que use la función `informe.leer_camion()` del programa `informe_funciones.py`.

Comentario

Al terminar este ejercicio tenés tres programas. `fileparse.py` contiene una función para parsear datos de archivos CSV en general, `parse_csv()`. Por otra parte, `informe_funciones.py` que produce un bello informe, y que contiene las funciones `leer_camion()` y `leer_precios()`. Finalmente, `costo_camion.py` calcula el costo de un camión, pero usando la función `leer_camion()` que fue escrita para el programa que genera el informe.

5.4 Búsqueda binaria

Hace un par de clases vimos la búsqueda secuencial de un elemento en una lista. Si la lista está previamente ordenada, ¿podemos encontrar una manera más eficiente de buscar elementos sobre ella?

Búsqueda sobre listas ordenadas

Si la lista está ordenada, hay una modificación muy simple que podemos hacer sobre el algoritmo de búsqueda lineal: si estamos buscando el elemento `e` en una lista que está ordenada de menor a mayor, en cuanto encontramos algún elemento

mayor a `e` podemos estar seguros de que `e` no está en la lista, por lo que no es necesario continuar recorriendo el resto.

Ejercicio 5.10: Búsqueda lineal sobre listas ordenadas.

Modificá la función `busqueda_lineal(lista, e)` de la [Sección 3.3](#) para el caso de listas ordenadas, de forma que la función pare cuando encuentre un elemento mayor a `e`. Llamá a tu nueva función `busqueda_lineal_lordenada(lista,e)` y guardala en el archivo `busqueda_en_listas.py`.

En el peor caso, ¿cuál es nuestra nueva hipótesis sobre comportamiento del algoritmo? ¿Es realmente más eficiente?

Búsqueda binaria

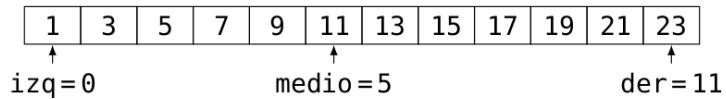
¿Podemos hacer algo mejor? Trataremos de aprovechar el hecho de que la lista está ordenada y vamos a hacer algo distinto: nuestro espacio de búsqueda se irá achicando a segmentos cada vez menores de la lista original. La idea es descartar segmentos de la lista donde el valor seguro que no puede estar:

- Consideramos como segmento inicial de búsqueda a la lista completa.
- Analizamos el punto medio del segmento (el valor central); si es el valor buscado, devolvemos el índice del punto medio.
- Si el valor central es mayor al buscado, podemos descartar el segmento que está desde el punto medio hacia la derecha.
- Si el valor central es menor al buscado, podemos descartar el segmento que está desde el punto medio hacia la izquierda.
- Una vez descartado el segmento que no nos interesa, volvemos a analizar el segmento restante, de la misma forma.
- Si en algún momento el segmento a analizar tiene longitud 0 significa que el valor buscado no se encuentra en la lista.

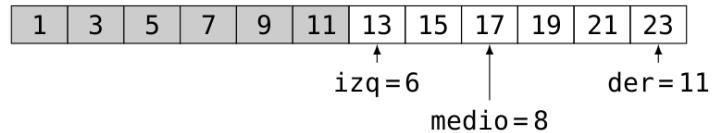
Para señalar la porción del segmento que se está analizando a cada paso, utilizaremos dos variables (`izq` y `der`) que contienen la posición de inicio y la posición de fin del segmento que se está considerando. De la misma manera usaremos la variable `medio` para contener la posición del punto medio del segmento.

A continuación ilustramos qué pasa cuando se busca el valor 18 en la lista `[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]`.

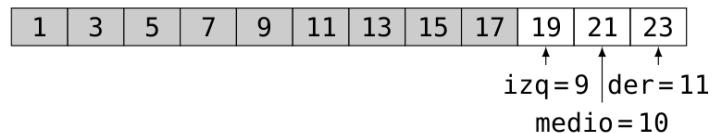
El arreglo inicial:



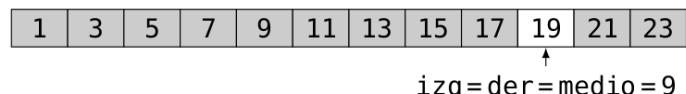
Paso 2 (`lista[5] < 18`):



Paso 3 (`lista[8] < 18`):



Paso 4 (lista[9] >= 18):



Ejemplo de una búsqueda usando el algoritmo de búsqueda binaria. Como no se encontró al valor buscado, devuelve -1.

El siguiente fragmento de código muestra una implementación de este algoritmo, incluyendo una instrucción de depuración (debug) con `print` para verificar su funcionamiento.

```
def busqueda_binaria(lista, x, verbose = False):
    '''Búsqueda binaria
    Precondición: la lista está ordenada
    Devuelve -1 si x no está en lista;
    Devuelve p tal que lista[p] == x, si x está en lista
    '''
    if verbose:
        print(f'[DEBUG] izq {izq} |der {der} |medio {medio}')
    pos = -1 # Inicializo respuesta, el valor no fue encontrado
    izq = 0
    der = len(lista) - 1
    while izq <= der:
        medio = (izq + der) // 2
        if verbose:
            print(f'[DEBUG] {izq:3d} |{der:3d} |{medio:3d}')
        if lista[medio] == x:
            pos = medio # elemento encontrado!
        if lista[medio] > x:
            der = medio - 1 # descarto mitad derecha
        else:
            izq = medio + 1 # descarto mitad izquierda
    return pos
```

A continuación varias ejecuciones de prueba:

```

>>> busqueda_binaria([1, 3, 5], 0, verbose = True)
[DEBUG] izq |der |medio
[DEBUG] 0 | 2 | 1
[DEBUG] 0 | 0 | 0
-1
>>> busqueda_binaria([1, 3, 5], 1, verbose = True)
[DEBUG] izq |der |medio
[DEBUG] 0 | 2 | 1
[DEBUG] 0 | 0 | 0
0
>>> busqueda_binaria([1, 3, 5], 2, verbose = True)
[DEBUG] izq |der |medio
[DEBUG] 0 | 2 | 1
[DEBUG] 0 | 0 | 0
-1
>>> busqueda_binaria([1, 3, 5], 3, verbose = True)
[DEBUG] izq |der |medio
[DEBUG] 0 | 2 | 1
[DEBUG] 2 | 2 | 2
1
>>> busqueda_binaria([1, 3, 5], 5, verbose = True)
[DEBUG] izq |der |medio
[DEBUG] 0 | 2 | 1
[DEBUG] 2 | 2 | 2
2
>>> busqueda_binaria([1, 3, 5], 6, verbose = True)
[DEBUG] izq |der |medio
[DEBUG] 0 | 2 | 1
[DEBUG] 2 | 2 | 2
-1
>>> busqueda_binaria([], 0, verbose = True)
[DEBUG] izq |der |medio
-1
>>> busqueda_binaria([1], 1, verbose = True)
[DEBUG] izq |der |medio
[DEBUG] 0 | 0 | 0
0
>>> busqueda_binaria([1], 3, verbose = True)
[DEBUG] izq |der |medio
[DEBUG] 0 | 0 | 0
-1
>>> busqueda_binaria([1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23], 18,
verbose = True)
[DEBUG] izq |der |medio
[DEBUG] 0 | 11 | 5
[DEBUG] 6 | 11 | 8
[DEBUG] 9 | 11 | 10
[DEBUG] 9 | 9 | 9
-1

```

Pregunta: En la línea `medio = (izq + der) // 2` efectuamos la división usando el operador `//` en lugar de `/`. ¿Qué pasaría si usáramos `/`?

¿Cuántas comparaciones hace este programa?

Para responder esto pensemos en el peor caso, es decir, que se descartaron varias veces partes del segmento para finalmente llegar a un segmento vacío y el valor buscado se encontró en este último paso o directamente no se encontraba en la lista.

En cada paso el segmento se divide por la mitad y se desecha una de esas mitades, y en cada paso se hace una comparación con el valor buscado. Por lo tanto, la cantidad de comparaciones que hacen con el valor buscado es aproximadamente igual a la cantidad de pasos necesarios para llegar a un segmento de tamaño 1. Veamos el caso más sencillo para razonar, y supongamos que la longitud de la lista es una potencia de 2, digamos `len(lista) = 2^k`:

1. Antes del primer paso, el segmento a tratar es de tamaño 2^k .
2. Antes del segundo paso, el segmento a tratar es de tamaño $2^{(k-1)}$.
3. Antes del tercer paso, el segmento a tratar es de tamaño $2^{(k-2)}$
4. Antes del paso k , el segmento a tratar es de tamaño $2^{(k-k)}=2^0=1$.

Por lo tanto este programa hace a lo sumo (en el peor caso) k comparaciones con el valor buscado cuando `len(lista) = 2^k`. Pero si despejamos k de la ecuación anterior, podemos ver que este programa realiza aproximadamente `log2(len(lista))` comparaciones.

Cuando `len(lista)` no es una potencia de 2 el razonamiento es menos prolífico, pero también vale que este programa realiza aproximadamente `log2(len(lista))` comparaciones. Concluimos entonces que:

Comparación entre ambos métodos

Veamos un ejemplo para entender cuánto más eficiente es la búsqueda binaria. Supongamos que tenemos una lista con un millón de elementos.

1. El algoritmo de búsqueda lineal hará una cantidad de operaciones proporcional a un millón; es decir que en el peor caso hará 1,000,000 comparaciones, y en un caso promedio, 500,000 comparaciones.
2. El algoritmo de búsqueda binaria hará como máximo $\log_2(1,000,000)$ comparaciones, o sea ¡no más que 20 comparaciones!.

Conclusión: Si una lista está previamente ordenada, podemos utilizar el algoritmo de búsqueda binaria, cuyo comportamiento es proporcional al *logaritmo* de la cantidad de elementos de la lista, y por lo tanto muchísimo más eficiente que la búsqueda lineal, especialmente si la lista es larga.

Ejercicio 5.11: Búsqueda binaria

Modificando la función `busqueda_binaria(lista, x)` adecuadamente, definí una función `donde_insertar(lista, x)` de forma que reciba una lista ordenada y un elemento y devuelva la posición de ese elemento en la lista (si se encuentra en la lista) o la posición donde se podría insertar el elemento para que la lista permanezca ordenada (si no está en la lista).

Por ejemplo: el elemento `3` podría insertarse en la posición `2` en la lista `[0, 2, 4, 6]` para mantenerla ordenada. Por lo tanto, el llamado `donde_insertar([0, 2, 4, 6], 3)` deberá devolver `2`, al igual que el llamado `donde_insertar([0, 2, 4, 6], 4)`.

Guarda tu modificación en un archivo `bbin.py`.

5.5 Complejidad de algoritmos

Resumen de algoritmos de Búsqueda

1. La búsqueda de un elemento en una secuencia es un algoritmo básico pero importante. El problema que intenta resolver puede plantearse de la siguiente manera: Dada una secuencia de valores y un valor, devolver el índice del valor en la secuencia, si se encuentra, de no encontrarse el valor en la secuencia señalarlo apropiadamente.
2. Una de las formas de resolver el problema es mediante la búsqueda lineal, que consiste en ir revisando uno a uno los elementos de la secuencia y comparándolos con el elemento a buscar. Este algoritmo no requiere que la secuencia se encuentre ordenada, la cantidad de comparaciones que realiza es proporcional a `len(secuencia)`.
3. Cuando la secuencia sobre la que se quiere buscar está ordenada, se puede utilizar el algoritmo de búsqueda binaria. Al estar ordenada la secuencia, se puede desacartar en cada paso la mitad de los elementos, quedando entonces con una eficiencia algorítmica proporcional a `log2(len(secuencia))`.

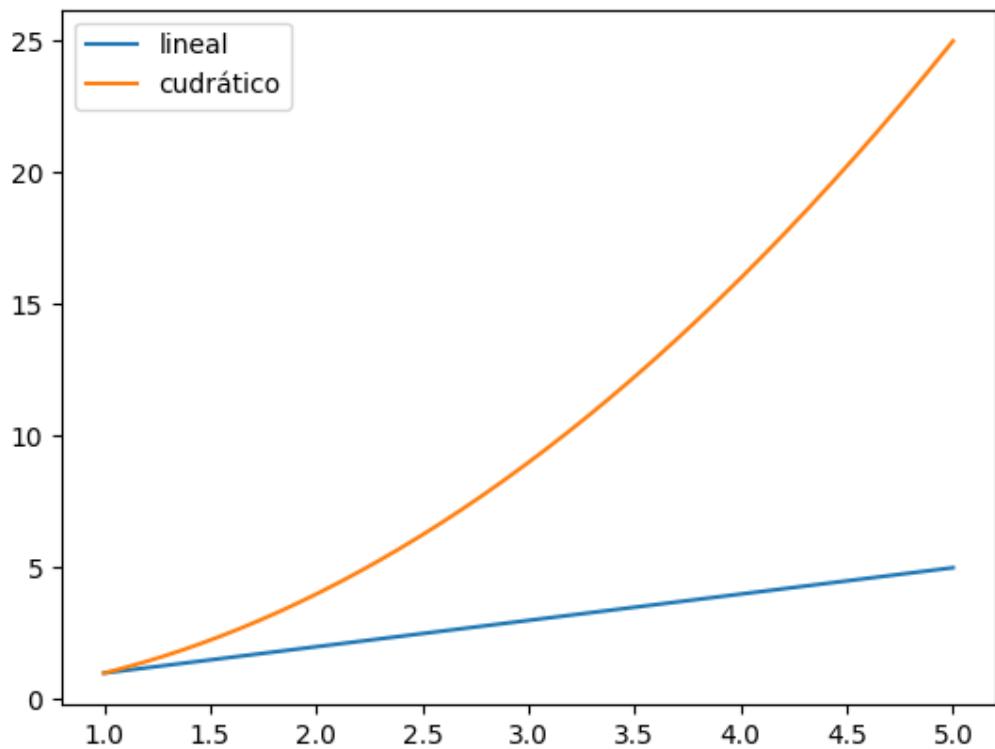
El análisis del comportamiento de un algoritmo puede ser muy engañoso si se tiene en cuenta el mejor caso, por eso suele ser mucho más ilustrativo tener en cuenta el peor caso. En algunos casos particulares podrá ser útil tener en cuenta, además, el caso promedio.

Complejidad de algoritmos

En las ciencias de la computación, el análisis de algoritmos es el proceso que permite determinar la complejidad de un algoritmo. Esta complejidad está típicamente medida en unidades de tiempo, o, análogamente, en la cantidad de operaciones que realiza el procesador antes de dar la respuesta. Esto permite comparar la eficiencia de diferentes algoritmos. Optimizar la eficiencia de los algoritmos es central en la tarea de un buen programador. Un algoritmo ineficiente puede no servir para nada. Esto es obvio en algoritmos que corren en tiempo real (imaginemos un algoritmo que conduce un vehículo y tarda demasiado en detectar a un peatón), pero también es importante en otros algoritmos.

Escribir programas eficientes no es una tarea sencilla. Muchas veces, las soluciones más directas no son las más eficientes. Los algoritmos más eficientes suelen aprovechar sutilezas que no son simples de comprender de un vistazo. En muchos casos los programadores deben incrementar la complejidad conceptual de un algoritmo para disminuir la complejidad computacional.

Por ejemplo, buscar la posición de un número en una lista clave recorriendo la lista lugar a lugar (búsqueda secuencial) demanda una cantidad de operaciones proporcional a la longitud de la lista (por cada elemento de la lista hacemos algunas operaciones fijas: comparar el elemento contra la clave, incrementar un contador, etc). Solemos decir que el algoritmo de búsqueda secuencial tiene un complejidad lineal en la longitud de la lista (ya que toma un tiempo $f(n)$, donde f es una función lineal en n , la cantidad de elementos de la lista). No vamos a preocuparnos aquí si $f(n) = 3 \cdot n + 5$ ó $f(n) = 2 \cdot n + 18$. No nos importan las constantes: simplemente diremos que $f(n)$ es lineal en n . En la literatura esto se escribe $f(n) = O(n)$ y se lee '*la función f tiene orden n* ', o ' *f es un O de n* '.



En cambio, la búsqueda binaria que vimos anteriormente, si bien es conceptualmente más compleja, resulta mucho más eficiente. Dada una clave y una lista ordenada, este algoritmo aprovecha el orden de la lista para no tener que comparar la clave con todos los elementos. En un primer paso compara con el elemento central de la lista y descarta toda una mitad de la lista realizando una sola comparación. No es obvio cómo calcular la complejidad de este método, pero explicamos que si la lista tiene longitud $n = 2^k$, el algoritmo de búsqueda binaria realiza a lo sumo $\log_2(n) + 1 = k + 1$ comparaciones antes de dar la respuesta (hacé un ejemplo con $n = 2^3 = 8$ o $n = 2^4 = 16$ para convencerte). En general, procediendo de esta forma el algoritmo encuentra la posición de la clave en $O(\log_2(n))$ pasos. Decimos en este caso que el algoritmo requiere tiempo logarítmico.

Comparando la función $f(n) = n$ con $g(n) = \log_2(n)$ para valores grandes de n resulta claro que la búsqueda binaria es mucho más eficiente que la búsqueda secuencial para listas ordenadas. En la próxima sección te vamos a proponer que hagas esta comparación gráficamente.

Un algoritmo cuadrático

Para dar un último ejemplo, supongamos que dada una lista de numeros (de longitud n) y un valor m queremos ver si $m = p \cdot q$ con p y q en la lista dada. Consideremos el siguiente algoritmo:

```
for p in lista :  
    for q in lista :  
        if m == p * q :  
            print ( " %d= %d* %d " %(m, p , q ) )
```

Este algoritmo realiza una comparación ($m == p * q$) para cada elemento p y cada elemento q de la lista. Es decir, realiza $n * n = n^2$ comparaciones. Es un algoritmo cuadrático. Su complejidad es $O(n^2)$.

Complejidad en el peor caso

El término análisis de algoritmos fue acuñado por Donald Knuth, uno de los fundadores de las ciencias de la computación. El análisis de algoritmos es una parte de la teoría de la complejidad computacional que no solo estudia la complejidad de los algoritmos sino de los problemas computacionales (la pregunta general de la teoría de la complejidad no sería cuál es la complejidad de la búsqueda secuencial o binaria, sino cuál es la complejidad mínima que puede tener un algoritmo que realice la tarea de buscar un elemento en una lista ordenada). En general, y sin mencionarlo, hablamos de la complejidad en el peor caso de un algoritmo. En algunos casos puede ocurrir que la búsqueda secuencial sea más eficiente que la búsqueda binaria (por ejemplo, considerá el caso en que el elemento buscado es justo el primer elemento de la lista, ¿cuánto tarda cada método?). Al hablar de la complejidad de una algoritmo (salvo que se mencione otra cosa) hablamos del tiempo que tarda ese algoritmo en el peor caso posible.

Estructuras de datos y Tipos Abstractos de Datos

El diseño de un algoritmo eficiente para resolver un problema requiere comprender profundamente los datos que este algoritmo manipulará para poder diseñar adecuadamente las estructuras de datos que los contendrán. El diseño de algoritmos eficientes requiere del diseño simultáneo de algoritmos y estructuras de datos adecuadas. Diferentes estructuras de datos son adecuadas para diferentes tipos de aplicaciones y algunas estructuras están diseñadas especialmente para un problema concreto. Una estructura de datos eficiente puede ser la clave para el diseño de un algoritmo eficiente.

La estructura lógica de las estructuras de datos se llaman Tipos Abstractos de Datos (TAD). Estos TAD son el modelo matemático de las estructuras de datos. Un TAD es una abstracción del tipo de datos: define su comportamiento desde el punto de vista

del usuario pero no dice cómo lo hace, no se mete en la implementación. Una estructura de datos concreta surge idealmente de la implementación de un TAD.

Ejercicios:

Ejercicio 5.12: Insertar un elemento en una lista

Uno de los problemas de la búsqueda binaria es que requiere que la lista esté ordenada. Si la lista se encuentra ordenada podemos mantener el orden evitando adjuntar nuevos elementos de forma desordenada.

Usando lo que hiciste en el [Ejercicio 5.11](#), agregale al archivo `bbin.py` una función `insertar(lista, x)` que reciba una lista ordenada y un elemento. Si el elemento se encuentra en la lista solamente devuelve su posición; si no se encuentra en la lista, lo inserta en la posición correcta para mantener el orden. En este segundo caso, también debe devolver su posición.

Ejercicio 5.13: Cálcular la complejidad de dos resoluciones de propagar

Ahora que tenés algunas herramientas teóricas más, volvé a leer las dos versiones de `propagar` del [Ejercicio 4.3](#) y el [Ejercicio 4.4](#) y compará sus complejidades.

Secuencias binarias

Para nosotros, una secuencia binaria es una lista que contiene solo 0's y 1's. Por ejemplo `s = [0, 1, 0, 0, 1]` es una secuencia binaria de longitud 5. La *primera* secuencia binaria de esa longitud es `[0, 0, 0, 0, 0]`, mientras que *la última* es `[1, 1, 1, 1, 1]`. Cada secuencia tiene una *siguiente* (salvo la última). No vamos a dar una definición precisa, pero esencialmente las secuencias pueden pensarse como representando números enteros en base dos y *la siguiente* secuencia es la que representa al siguiente número. Por convención, diremos que la secuencia siguiente de la última es la primera.

Ejemplos:

<code>[0, 0, 0, 0, 0]</code>	<code>-></code>	<code>[0, 0, 0, 0, 1]</code>
<code>[0, 0, 1, 1, 0]</code>	<code>-></code>	<code>[0, 0, 1, 1, 1]</code>
<code>[0, 0, 1, 1, 1]</code>	<code>-></code>	<code>[0, 1, 0, 0, 0]</code>
<code>[1, 1, 1, 1, 1]</code>	<code>-></code>	<code>[0, 0, 0, 0, 0]</code>

La función `incrementar(s)` calcula la secuencia siguiente de una secuencia dada:

```
def incrementar(s):
    carry = 1
    l = len(s)

    for i in range(l-1, -1, -1):
        if (s[i] == 1 and carry == 1):
            s[i] = 0
            carry = 1
        else:
            s[i] = s[i] + carry
            carry = 0
    return s
```

Ejercicio 5.14: Complejidad de `incrementar()`

Si tomamos `n = len(s)` podemos tratar de medir la complejidad de la función `incrementar()` en términos de la longitud `n` de la secuencia. ¿Te parece que `incrementar()` es una función lineal, cuadrática, logarítmica o exponencial? ¿Por qué?

Ejercicio 5.15: Un ejemplo más complejo

Por último, escribí una función `listar_secuencias(n)` que devuelva una lista con todas las secuencias binarias de longitud `n` comenzando con la primera `([0]*n)` y usando en cada paso la función `incrementar()` definida más arriba. ¿Cuántas listas hay de longitud `n`? ¿Y de longitud `n+1`?

¿Podés correr `listar_secuencias(15)`? ¿Y `listar_secuencias(20)`? ¿Hasta cuánto llegas a correr en un tiempo razonable?

¿Te parece que `listar_secuencias(n)` es una función lineal, cuadrática, logarítmica o exponencial en `n`? ¿Por qué?

5.6 Gráficos de complejidad

Contar la cantidad de operaciones de un algoritmo

La siguiente función realiza una búsqueda secuencial de un elemento en una lista. Devuelve la posición del elemento si lo encuentra y -1 si no lo encuentra.

```
def busqueda_secuencial(lista, x):
    '''Si x está en la lista devuelve el índice de su primera aparición,
    de lo contrario devuelve -1.
    '''
    pos = -1
    for i,z in enumerate(lista):
        if z == x:
            pos = i
            break
    return pos
```

Esta modificación de la función cuenta (y devuelve) además cuántas comparaciones ($z == x$) hace la función. Observá que devuelve un par de datos.

```
def busqueda_secuencial_(lista, x):
    '''Si x está en la lista devuelve el índice de su primer aparición,
    de lo contrario devuelve -1.
    '''
    comps = 0 # inicializo en cero la cantidad de comparaciones
    pos = -1
    for i,z in enumerate(lista):
        comps += 1 # sumo la comparación que estoy por hacer
        if z == x:
            pos = i
            break
    return pos, comps
```

Si querés acceder a la posición podés usar `busqueda_secuencial_(lista, x)[0]` y para acceder a la cantidad de comparaciones que hizo `busqueda_secuencial_(lista, x)[1]`.

Ejercicio 5.16: Contar comparaciones en la búsqueda binaria

Modificá el código de búsqueda binaria (`busqueda_binaria(lista, x)`) introducido en la [Sección 5.4](#), de forma que devuelva (además de la posición del elemento en la lista) la cantidad de comparaciones que realizó el algoritmo para encontrarlo o decidir que no está.

Gráficar la cantidad de comparaciones promedio

La siguiente función `generar_lista(n, m)` devuelve una lista ordenada de n elementos diferentes entre 0 y $m-1$, mientras que `generar_elemento(m)` devuelve un elemento aleatorio en el mismo rango de valores.

```
import random

def generar_lista(n, m):
    l = random.sample(range(m), k = n)
    l.sort()
    return l

def generar_elemento(m):
    return random.randint(0, m-1)
```

Dada una lista ya generada, digamos que un *experimento elemental* es generar un elemento, buscarlo en la lista y contar la cantidad de comparaciones realizadas. Esta cantidad de operaciones es el *resultado* del experimento elemental.

```
m = 10000
n = 100
lista = generar_lista(n, m)

# acá comienza el experimento
x = generar_elemento(m)
comps = busqueda_secuencial_(lista, x) [1]
```

Entonces, el siguiente código da la cantidad de comparaciones *promedio* en k experimentos elementales. Observá que hay muchas variables diferentes dando vueltas: n , m y k .

```
m = 10000
n = 100
k = 1000
lista = generar_lista(n, m)

def experimento_secuencial_promedio(lista, m, k):
    comps_tot = 0
    for i in range(k):
        x = generar_elemento(m)
        comps_tot += busqueda_secuencial_(lista,x) [1]

    comps_prom = comps_tot / k
    return comps_prom
```

Como las listas tienen $n = 100$ elementos y estoy buscando un número cualquiera entre m números diferentes, es casi seguro que no lo voy a encontrar y que voy a tener que recorrer toda la lista para concluir esto (aunque en algún caso puede ser

que esté y lo encuentre antes de recorrerla toda!). Entonces el promedio de comparaciones va a dar cercano al largo n de la lista, quizás un poco menor. Tiene una componente aleatoria, es un *experimento numérico*.

Si decíamos que buscar un elemento era un *experimento elemental* digamos que repetir k experimentos elementales y calcular el promedio de comparaciones es un *experimento de promedios*.

Grafiquemos los resultados de estos *experimentos de promedios* para diferentes listas de largos n entre 1 y 256. Es decir, estaremos graficando la cantidad de comparaciones que hace en promedio el algoritmo de búsqueda secuencial sobre una lista de largo n , para diferentes valores de n .

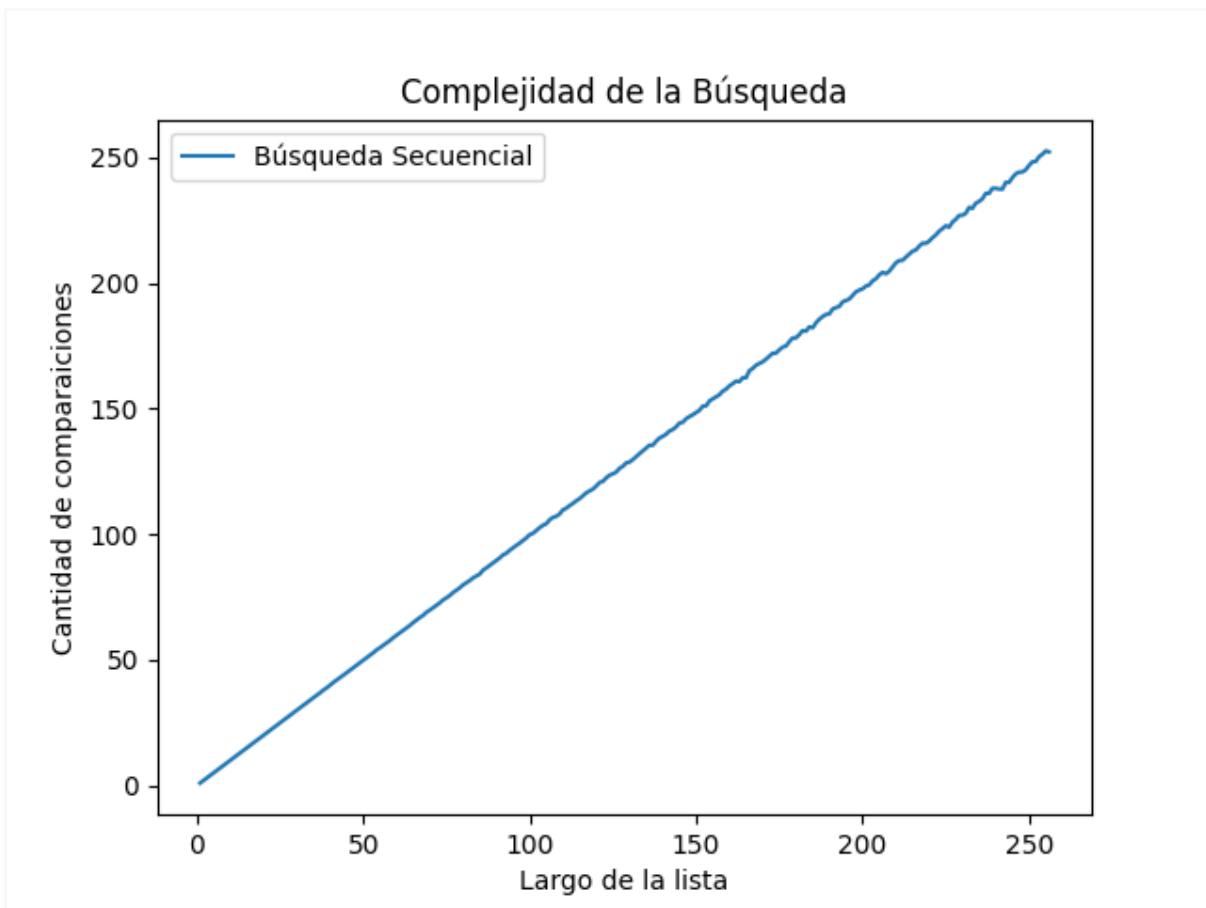
```
import matplotlib.pyplot as plt
import numpy as np

m = 10000
k = 1000

largos = np.arange(256) + 1 # estos son los largos de listas que voy a usar
comps_promedio = np.zeros(256) # aca guardo el promedio de comparaciones
sobre una lista de largo i, para i entre 1 y 256.

for i, n in enumerate(largos):
    lista = generar_lista(n, m) # genero lista de largo n
    comps_promedio[i] = experimento_secuencial_promedio(lista, m, k)

# ahora grafico largos de listas contra operaciones promedio de búsqueda.
plt.plot(largos,comps_promedio,label = 'Búsqueda Secuencial')
plt.xlabel("Largo de la lista")
plt.ylabel("Cantidad de comparaciones")
plt.title("Complejidad de la Búsqueda")
plt.legend()
```



En la próxima clase estudiaremos en detalle la librería `matplotlib` que ya empezamos a usar la clase pasada. Por ahora solo agregamos la función `plot(x, y)` a la que se le pasan dos vectores (o listas) `x` e `y` y realiza una gráfico de líneas uniendo los puntos con esas coordenadas. El parámetro `label` permite ponerle un nombre a la curva que se muestra luego con la función `plt.legend()`.

Este gráfico parece medio sonzo, pero en el próximo ejercicio va a ir tomando color.

Ejercicio 5.17: Búsqueda binaria vs. búsqueda secuencial

En este Ejercicio vamos a rehacer los gráficos del ejemplo anterior, pero primero cambiando el algoritmo de búsqueda y luego comparando ambos algoritmos.

1. Usando `experimento_secuencial_promedio(lista, m, k)` como base, escribí una función `experimento_binario_promedio(lista, m, k)` que cuente la cantidad de comparaciones que realiza en promedio (entre `k` experimentos elementales) la búsqueda binaria sobre la lista pasada como parámetro.
2. Graficá los resultados de estos experimentos para listas de largo entre 1 y 256.

3. Graficá ambas curvas en una misma figura, nombrando adecuadamente las curvas, los ejes y la figura completa. Jugá con `xlim` e `ylim` para visualizar bien las dos curvas, aunque tengas que restringir el rango.
4. ¿Qué observas en estos gráficos? ¿Qué podés decir sobre la complejidad de cada algoritmo? ¿Son similares?

El código de este ejercicio guardalo en `plot_bbin_vs_bsec.py`.

5.7 Cierre de la quinta clase

En esta quinta clase trabajamos con funciones y creamos módulos. También aprendimos algunas nociones de complejidad de algoritmos, estudiamos la búsqueda binaria y comparamos su performance con la de la búsqueda secuencial..

Para cerrar esta clase te pedimos dos cosas:

- Que recopiles las soluciones de los siguientes ejercicios:
 1. El archivo `fileparse.py` del [Ejercicio 5.5](#) o del siguiente.
 2. El archivo `informe_funciones.py` de [Ejercicio 5.8](#).
 3. El archivo `costo_camion.py` del [Ejercicio 5.9](#).
 4. El archivo `bbin.py` del [Ejercicio 5.12](#).
 5. El archivo `plot_bbin_vs_bsec.py` del [Ejercicio 5.17](#).
- Que completes [este formulario](#) 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!

6. 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 mismo unos años más tarde.

Por eso insistimos con algunos temas de estilo, documentación, especificación 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 16/9 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 experiencia pueda tener continuidad en el tiempo.

- [6.1 Control de errores](#)
- [6.2 El módulo principal](#)
- [6.3 Cuestiones de diseño](#)
- [6.4 Contratos: Especificación y Documentación](#)
- [6.5 Estilos de codeo](#)
- [6.6 La biblioteca matplotlib](#)
- [6.7 Cierre de la sexta clase](#)

6.1 Control de errores

Aunque ya hablamos de *excepciones*, en esta sección hablaremos de administración de excepciones y control de errores con mayor detalle.

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](#).

```
ArithmetricError
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

Lancemos excepciones

La función `parse_csv()` que escribiste en el [Ejercicio 5.6](#) admite seleccionar algunas columnas por el usuario, 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 = ['name', '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 situación real, es posible que los archivos CSV de entrada estén "rotos", ausentes, o que su contenido no se acomode 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])
Row 4: No pude convertir ['Mandarina', '', '51.23']
Row 4: Motivo: invalid literal for int() with base 10: ''
Row 7: No pude convertir ['Naranja', '', '70.44']
Row 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}]
>>>
```

Ejercicios:

Trabajá siempre con las últimas versiones de tus archivos. En esta clase vamos a trabajar sobre 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.

Ejercicio 6.1: Errores silenciados

Modificá `parse_csv()` de modo que le usuario 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 silencias los errores. Es mejor informar los problemas y darle al usuario la opción de silenciarlos explícitamente. Un buen diálogo entre el código y el usuario facilita el debugging y el buen uso del programa.

6.2 El módulo principal

En esta sección introducimos el concepto de módulo principal.

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]} {sys.argv[1]} {sys.argv[2]}')
camion = sys.argv[1]
precios = sys.argv[2]
...
```

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 código 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 poder 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 6.2: 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
<hr/>			
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, modifícá 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 6.3: Hacer un script

Finalmente, modifícá 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.

6.3 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](#) del inglés o en español "[Test del pato](#)" 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](#) 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 *línneas*, 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()`.

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 restrinas las opciones innecesariamente. Con mayor flexibilidad suele venir asociada una mayor potencia.

Ejercicio

Ejercicio 6.4: 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 ó 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 = ['name,cajones,precio', 'Lima,100,34.23', 'Naranja,50,91.1',  
'Mburucuya,75,45.1']  
>>> camion = fileparse.parse_csv(lines, types=[str,int,float])  
>>>
```

Y ahora que 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 lio) ...  
>>>
```

Sí, hay que tener cuidado.

Ejercicio 6.5: 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.

6.4 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.

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 mantienen 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), 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 código de 4 dígitos elegido al azar'''
    digitos = ('0','1','2','3','4','5','6','7','8','9')
    código = ""
    for i in range(4):
        candidato = random.choice(digitos)
        # Debemos asegurarnos de no repetir dígitos
        while candidato in código:
            candidato = random.choice(digitos)
        código = código + candidato
    return código
```

¿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 otro 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](#).

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 posición `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.
- La 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.

Aseveraciones

Retomamos acá el concepto de aseveración que introdujimos en la [Sección 4.1](#). 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 6.1](#), es mejor que los errores generen excepciones ruidosamente y no atraparlas si no sabemos exactamente cómo manejarlas.

Veamos otro ejemplo, tal vez más interesante. Consideraremos una función `sumar_enteros(desde, hasta)` que implementa la sumatoria $\sum_{i=desde}^{hasta} i$.

$$\sum_{i=desde}^{hasta} i$$

En este caso, analizando los parámetros que recibirá la función, podemos definir la precondition 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 precondition puede ser necesario exigir que `hasta` sea mayor o igual a `desde`.

La declaración de la función, incluyendo documentación, precondition 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 6.6: 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](#).

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 p^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 *demonstrar* 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 6.7: Invariante en sumas

En el [Ejercicio 6.6](#), 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. Esta 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.

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 6.8: 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
```

6.5 Estilos de codeo

PEP 8 - La guía de estilo para Python

La comunidad de usuarios de Python ha adoptado una guía de estilo que facilita la lectura del código y la consistencia entre programas de distintos usuarios. Esta guía no es de seguimiento obligatorio, pero es altamente recomendable. El documento completo se denomina PEP 8 y está escrito originalmente en [inglés](#), aunque hay alguna traducción al [castellano](#).

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)

# opción 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 aritméticos:

```
# 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 compatibilidad. 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](#) para tener toda la información sobre estilo recomendado en Python.

Zen de Python

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](#)

El Zen de Python es una colección de 20 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!

6.6 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.

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)
```

6.7 Cierre de la sexta clase

En esta sexta 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 6.4](#).
 2. El archivo `informe.py` del [Ejercicio 6.5](#).

3. El archivo `documentacion.py` del [Ejercicio 6.8](#).
4. El archivo `random_walk.py` del [Ejercicio 6.10](#).
 - Que completes [este formulario](#) 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!

6.7 Cierre de la sexta clase

En esta sexta 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 standard.

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 6.4](#).
 2. El archivo `informe.py` del [Ejercicio 6.5](#).
 3. El archivo `documentacion.py` del [Ejercicio 6.8](#).
 4. El archivo `random_walk.py` del [Ejercicio 6.10](#).

- Que completes [este formulario](#) 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!

6.7 Cierre de la sexta clase

En esta sexta 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:

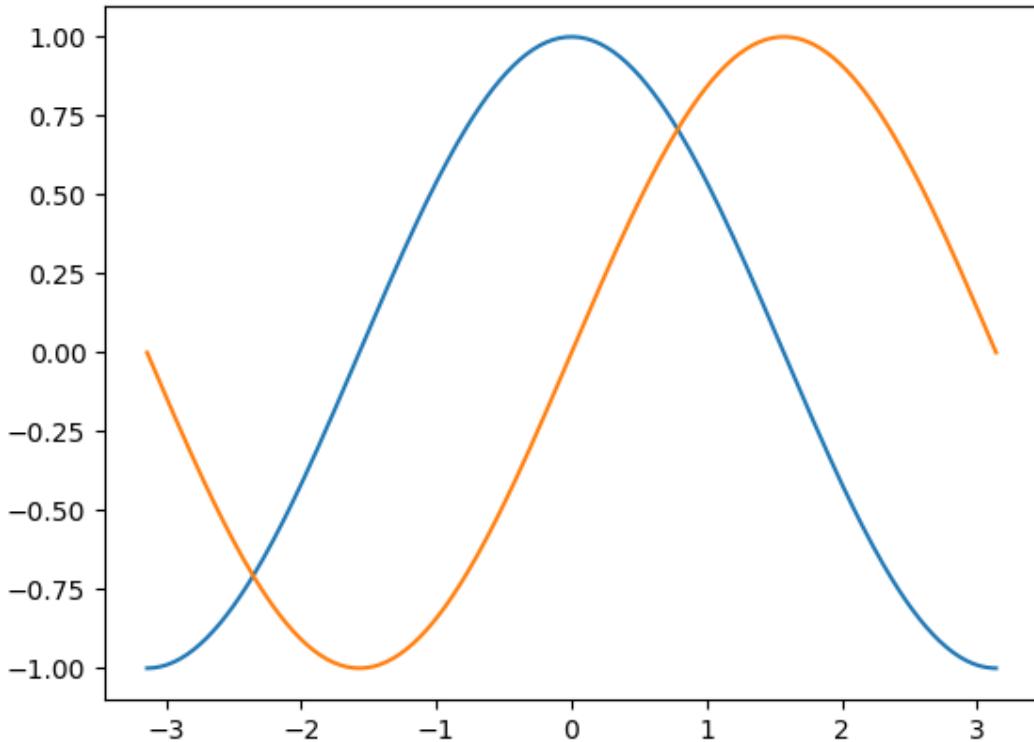
- Que recopiles las soluciones de los siguientes ejercicios:
 1. El archivo `fileparse.py` del [Ejercicio 6.4](#).
 2. El archivo `informe.py` del [Ejercicio 6.5](#).
 3. El archivo `documentacion.py` del [Ejercicio 6.8](#).
 4. El archivo `random_walk.py` del [Ejercicio 6.10](#).

- Que completes [este formulario](#) 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!

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

El ploteo estándar



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 continua de ancho 1 (en pixeles)
plt.plot(X, C, color="blue", linewidth=1.0, linestyle="-")

# Plotea el seno con una línea verde continua 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))

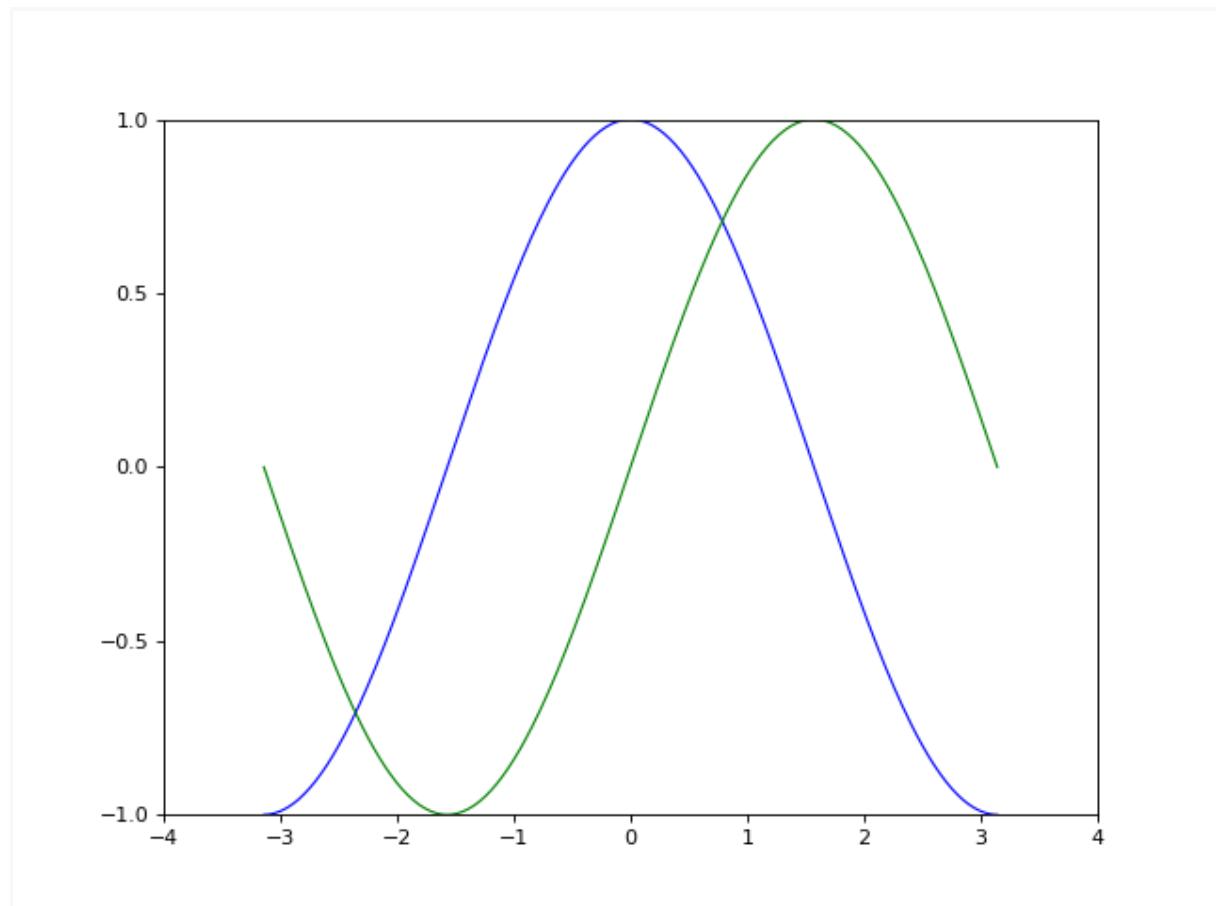
# Rango del eje y
plt.ylim(-1.0, 1.0)

# 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()
```



Los gráficos que genera matplotlib son muy flexibles, te dejamos [un machete](#) resumiendo las variaciones más usuales.

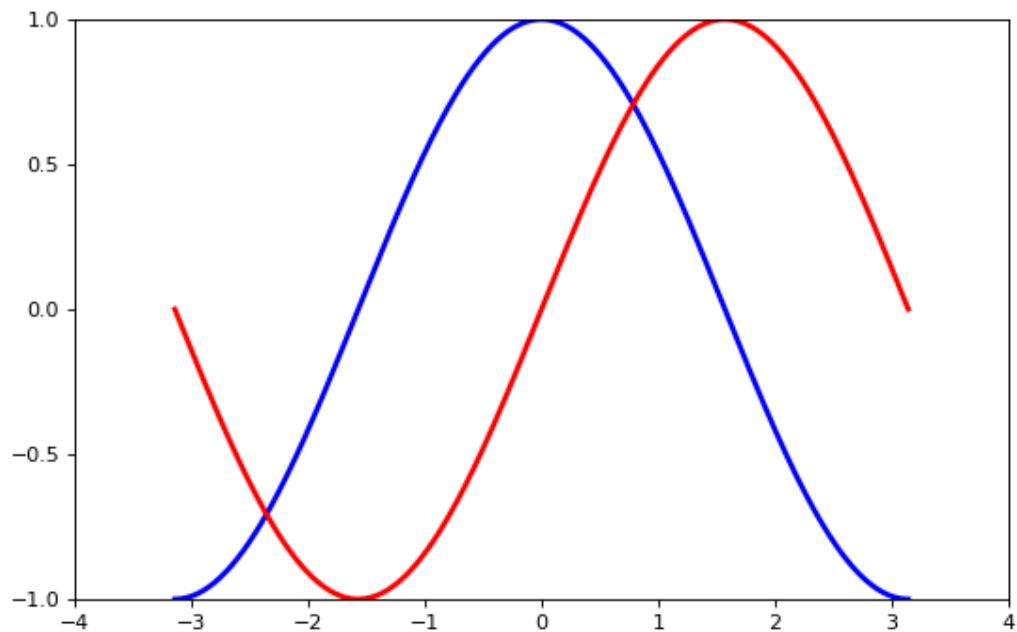
A continuación presentamos detalles técnicos de esta biblioteca tan útil. No hace falta que te los aprendas (igual te los vas a olvidar), ni que pruebes todas las combinaciones. Podés volver a esta página o a la [documentación](#) cuando lo necesites. Iguál mirá los ejercicios al final de esta sección,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

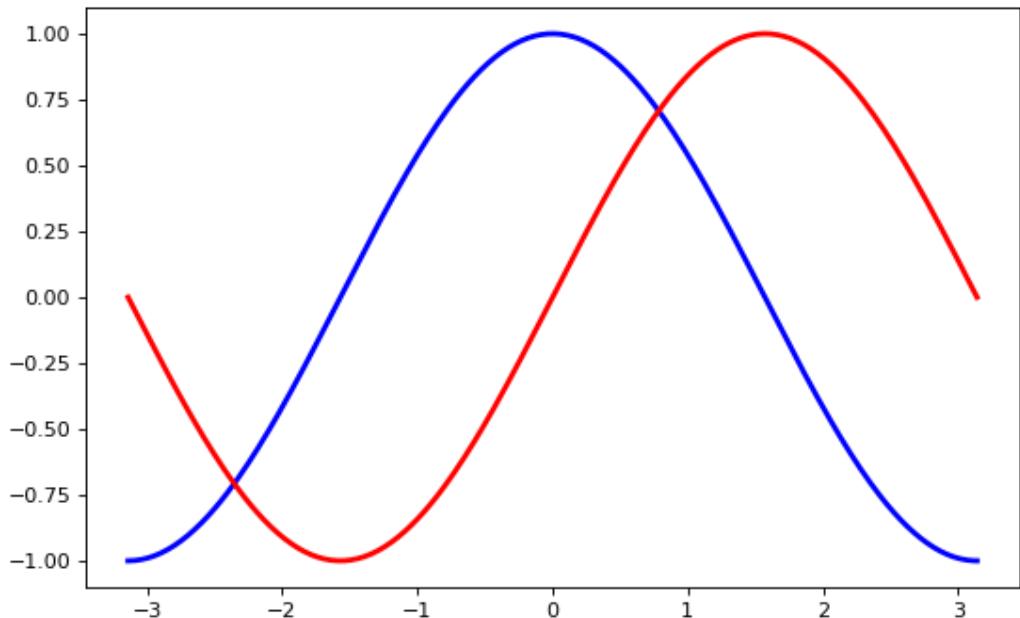
```
...
plt.figure(figsize=(10, 6), dpi=80)
plt.plot(X, C, color="blue", linewidth=2.5, linestyle="--")
plt.plot(X, S, color="red",   linewidth=2.5, linestyle="--")
...
```



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.

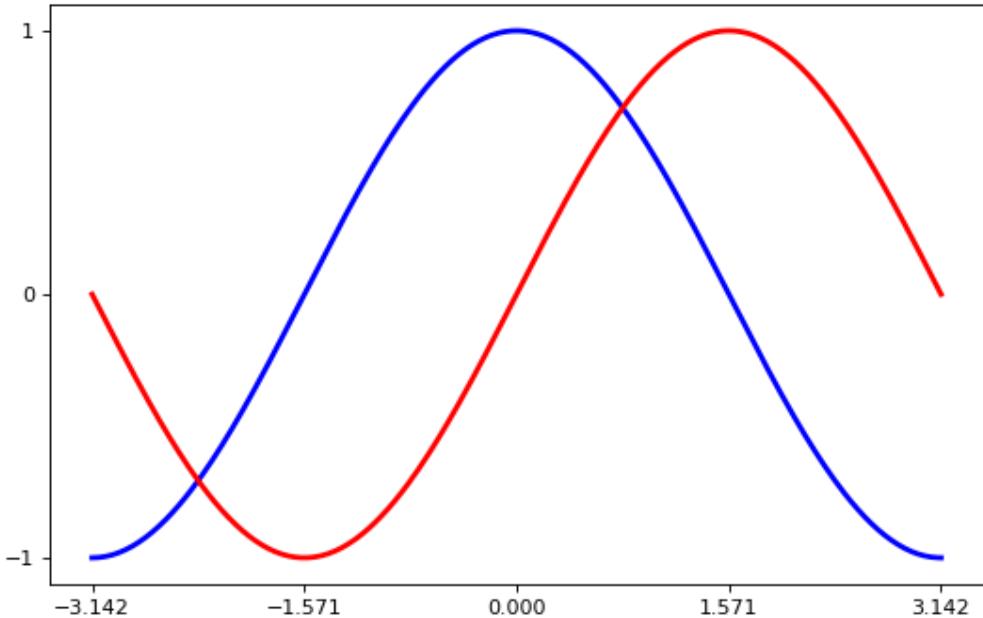
```
...
plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.ylim(C.min() * 1.1, C.max() * 1.1)
...
```



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 ($+/-\pi, +/-\pi/2$). Cambiémoslos para mostrar únicamente esos valores.

```
...
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi])
plt.yticks([-1, 0, +1])
...
```



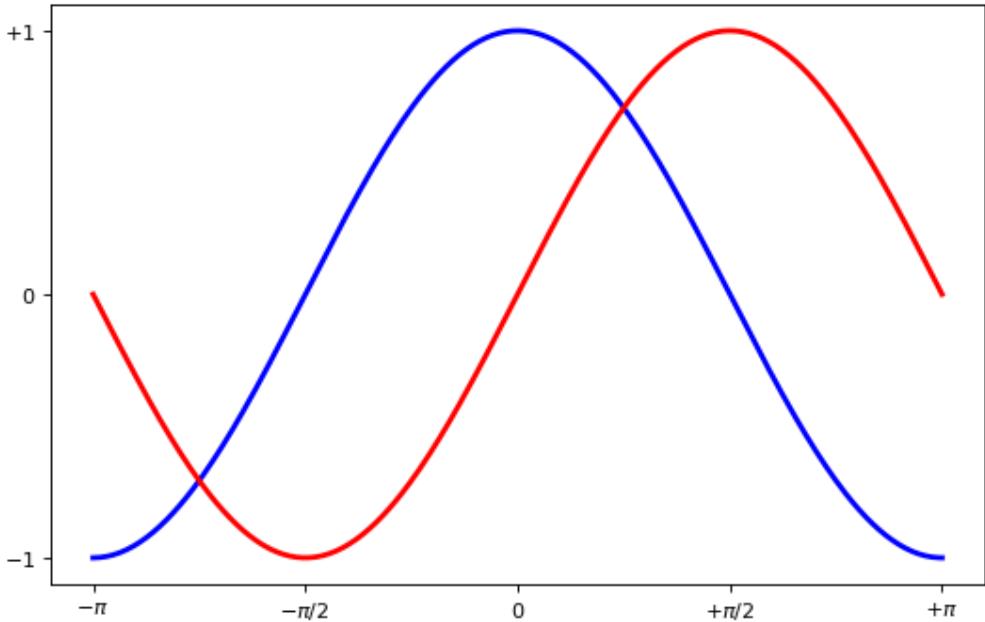
Texto de las marcas en los ejes

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](#) 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'$+\pi$'])

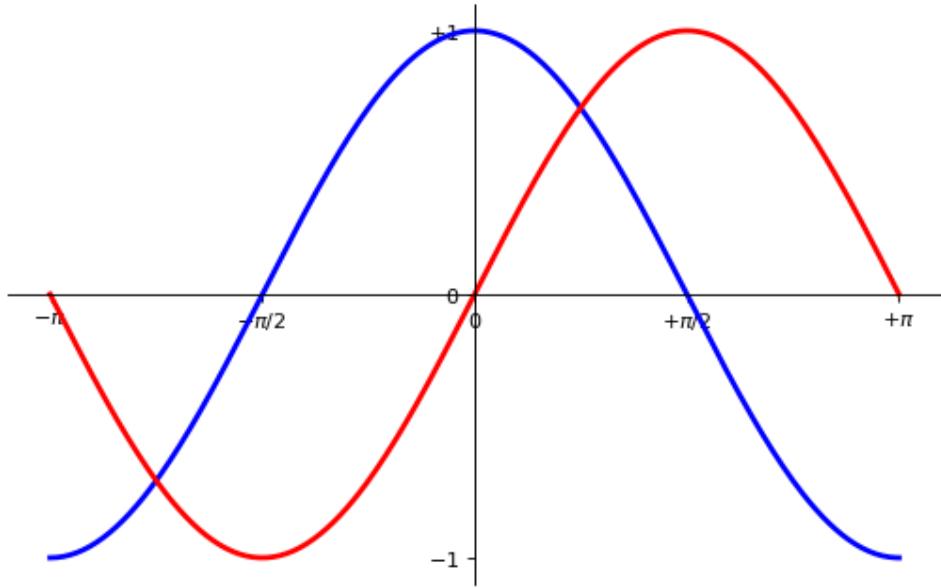
plt.yticks([-1, 0, +1],
           [r'$-1$', r'$0$', r'$+1$'])
...
```



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))
...
```

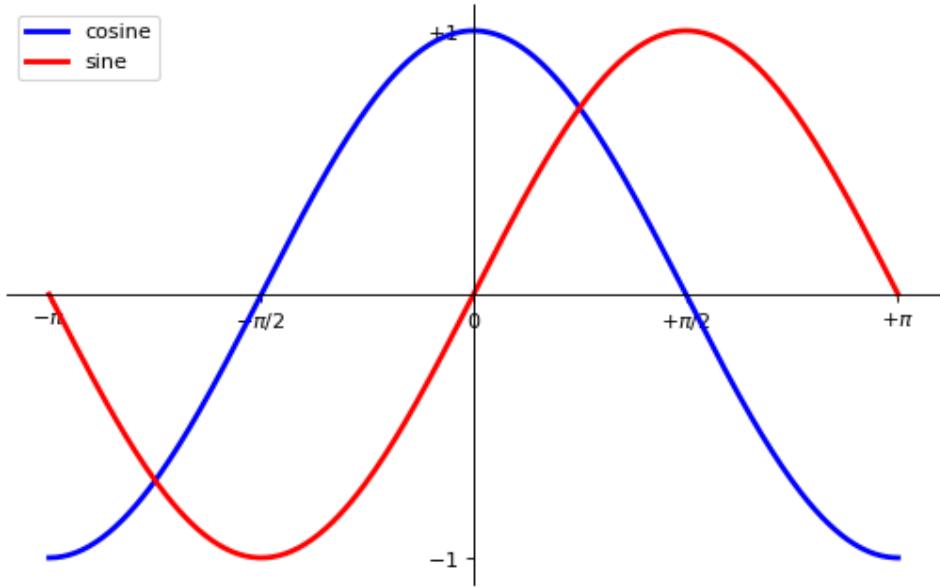


Pongámosle un título

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')
...
```



Algunos puntos interesantes

Vamos a marcar algunos puntos interesantes usando el comando `annotate`. Elegimos el valor $2\pi/3$ 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"))

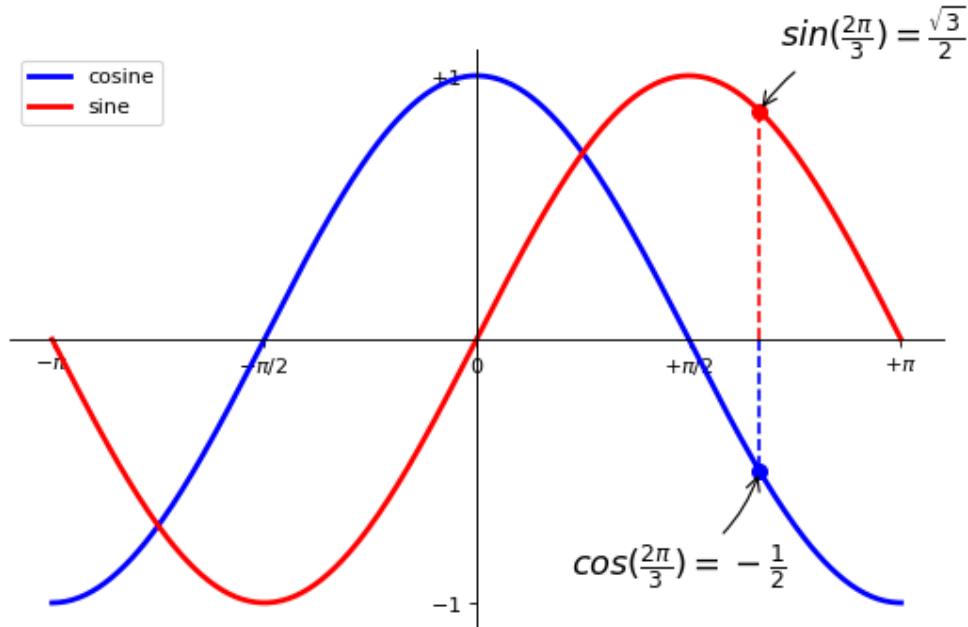
plt.plot([t, t], [0, np.sin(t)], color='red', linewidth=2.5, linestyle="--")
plt.scatter([t, ], [np.sin(t), ], 50, color='red')

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"))
...

```



El diablo está en los detalles

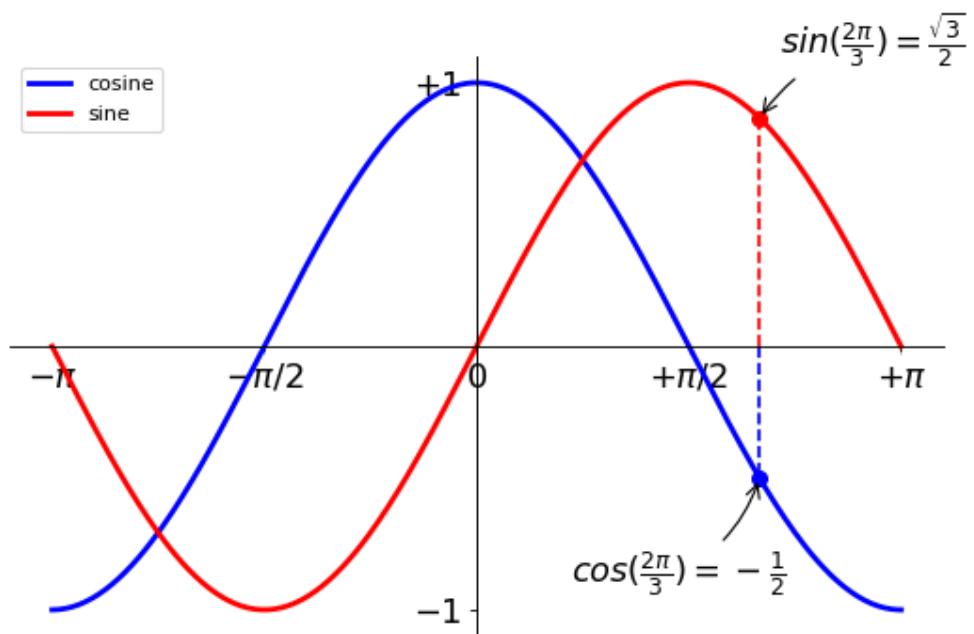
Notá (vas a tener que mirar muy de cerca) que los ejes tapan los trazos de las funciones seno y coseno, y éstas tapan los valores escritos sobre los ejes. Si esto fuera una publicación quedaría feo.

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))
...

```



Figuras, subplots, ejes y marcas (ticks)

En matplotlib el término "figura" se refiere a toda la ventana que conforma la interfase al usuario. 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 explícitamente, 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 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 usuario 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. También 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*:

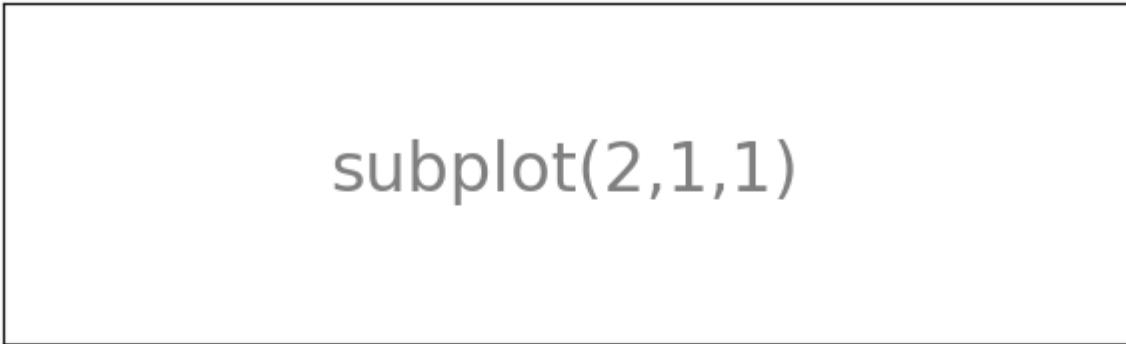
```
def cm2inch(value):
    return value/2.54

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

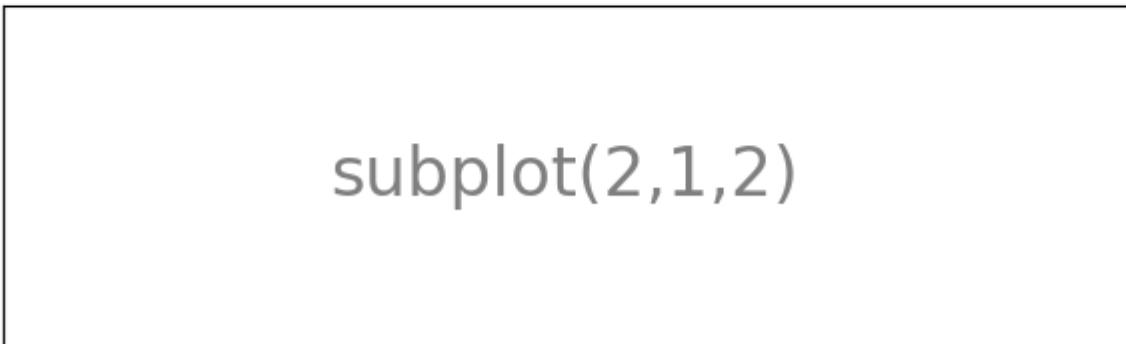
Subplots

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:



subplot(2,1,1)



subplot(2,1,2)

Ejemplo:

subplot(1,2,1)

subplot(1,2,2)

Ejemplo:

subplot(2,2,1)

subplot(2,2,2)

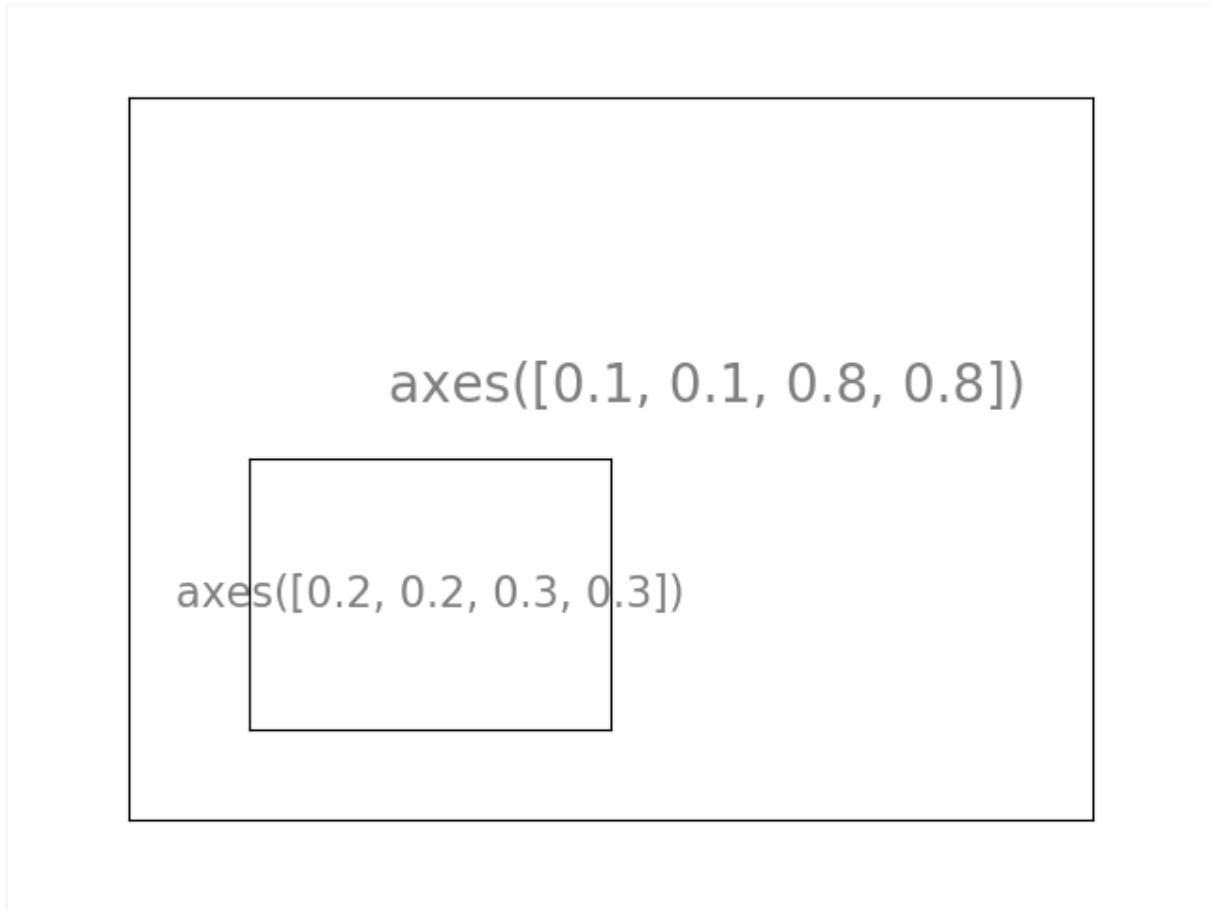
subplot(2,2,3)

subplot(2,2,4)

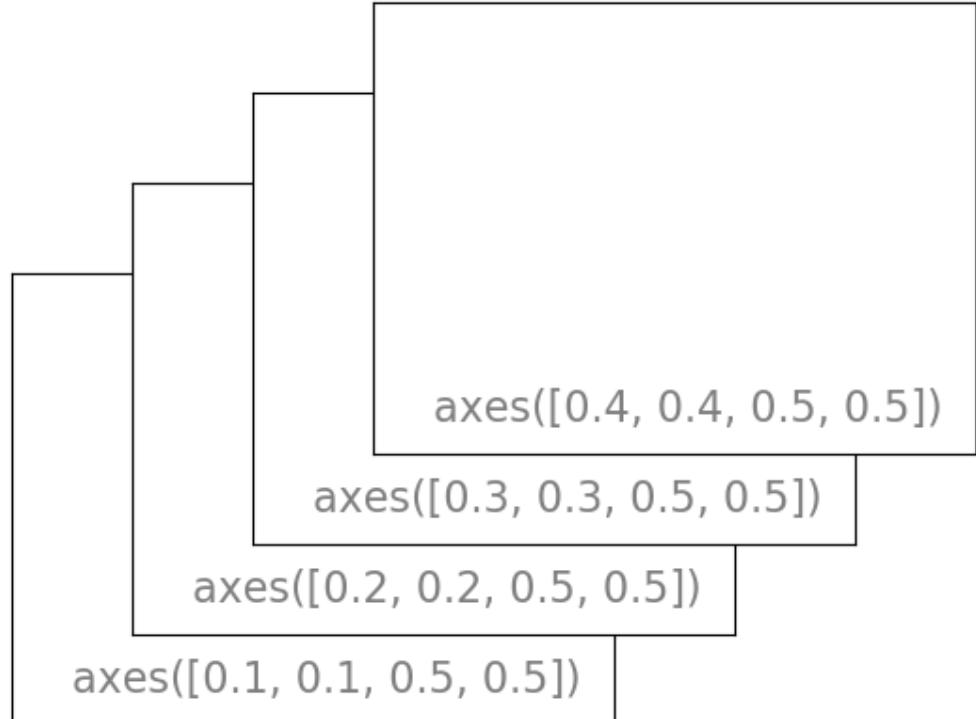
Ejes

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:



Ejemplo:



Ejercicios:

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

Ejercicio 6.9: 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

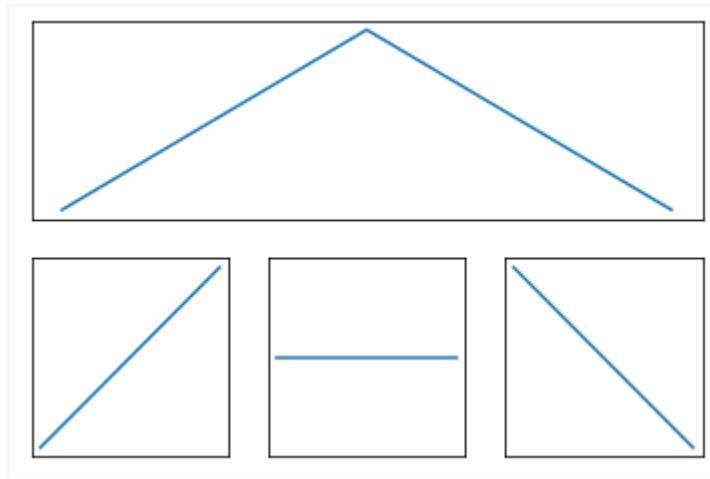
plt.subplot(2, 2, 3) # define la primera de abajo, que sería la tercera si
fuera una grilla regular de 2x2
plt.plot([0,1],[0,1])
plt.xticks([]), plt.yticks([])
```

```

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()

```



Ejercicio 6.10: Caminatas al azar

Una [caminata al azar](#) 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()

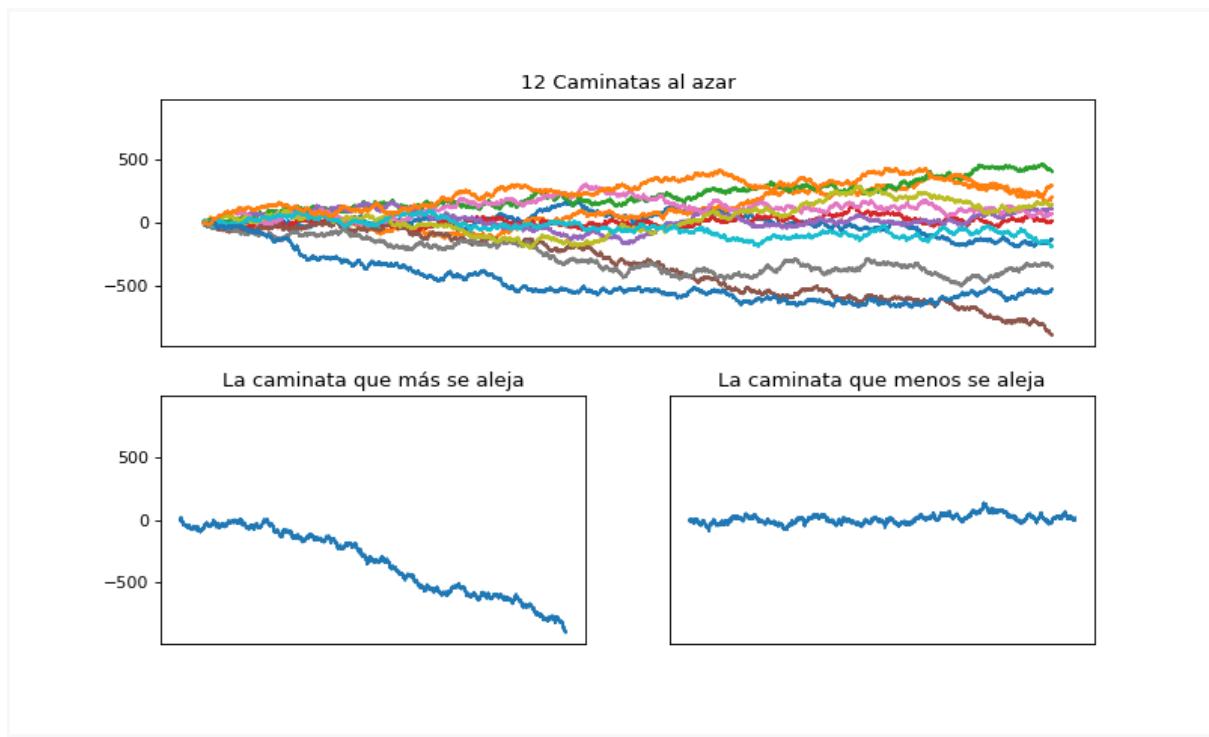
```

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 6.9](#) para graficar tres subplots 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:



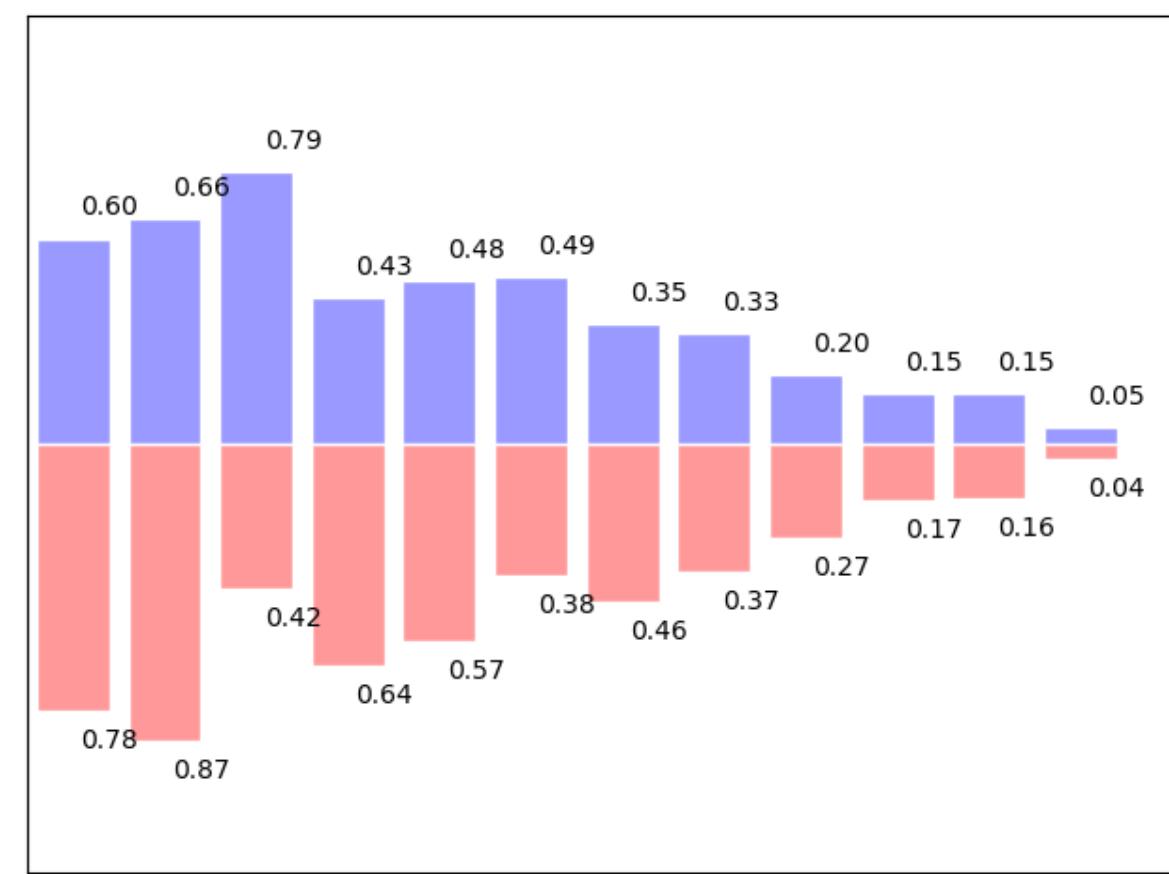
Optativos:

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](#).

Ejercicio 6.11: 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)
```



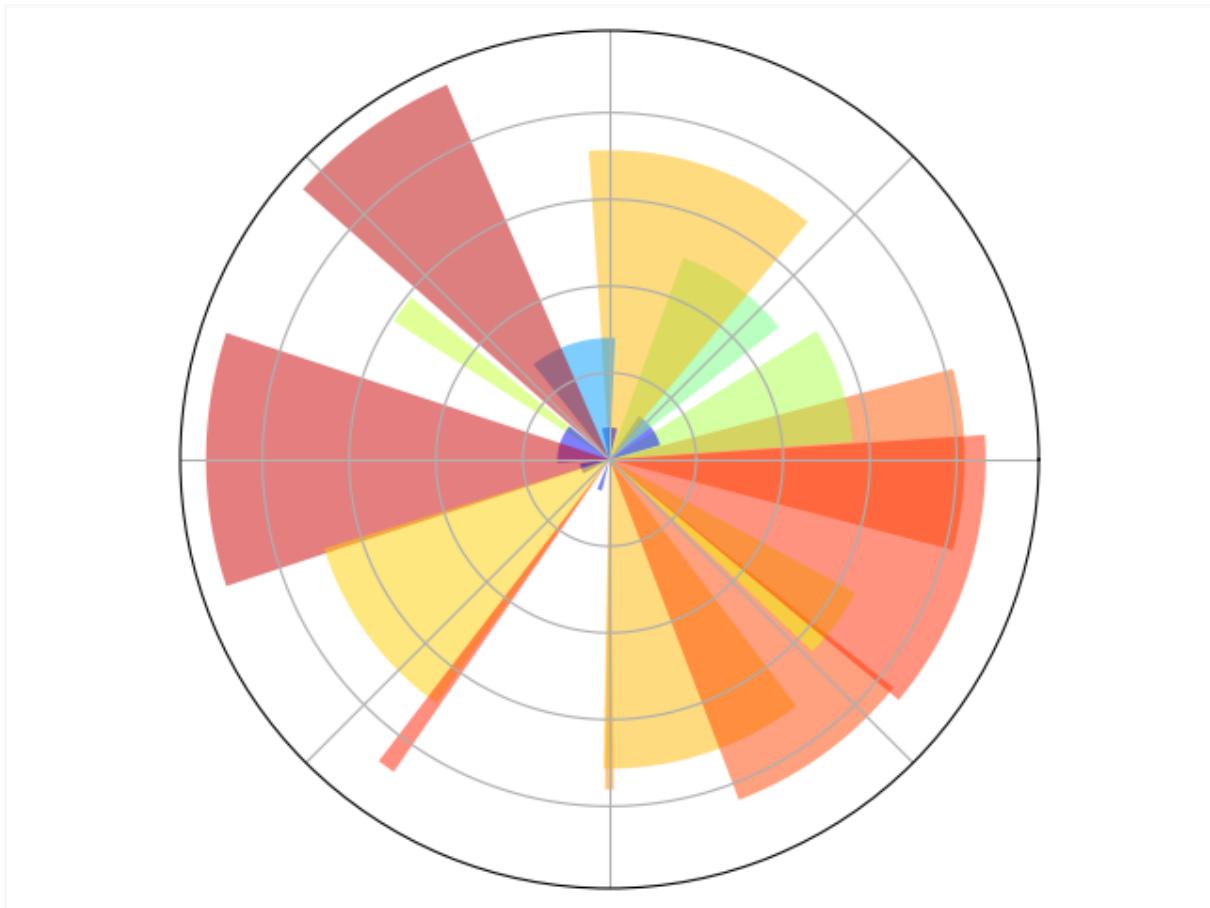
Ejercicio 6.12: Coordenadas polares

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(radis, bars):
    bar.set_facecolor(plt.cm.jet(r / 10.))
    bar.set_alpha(0.5)
```

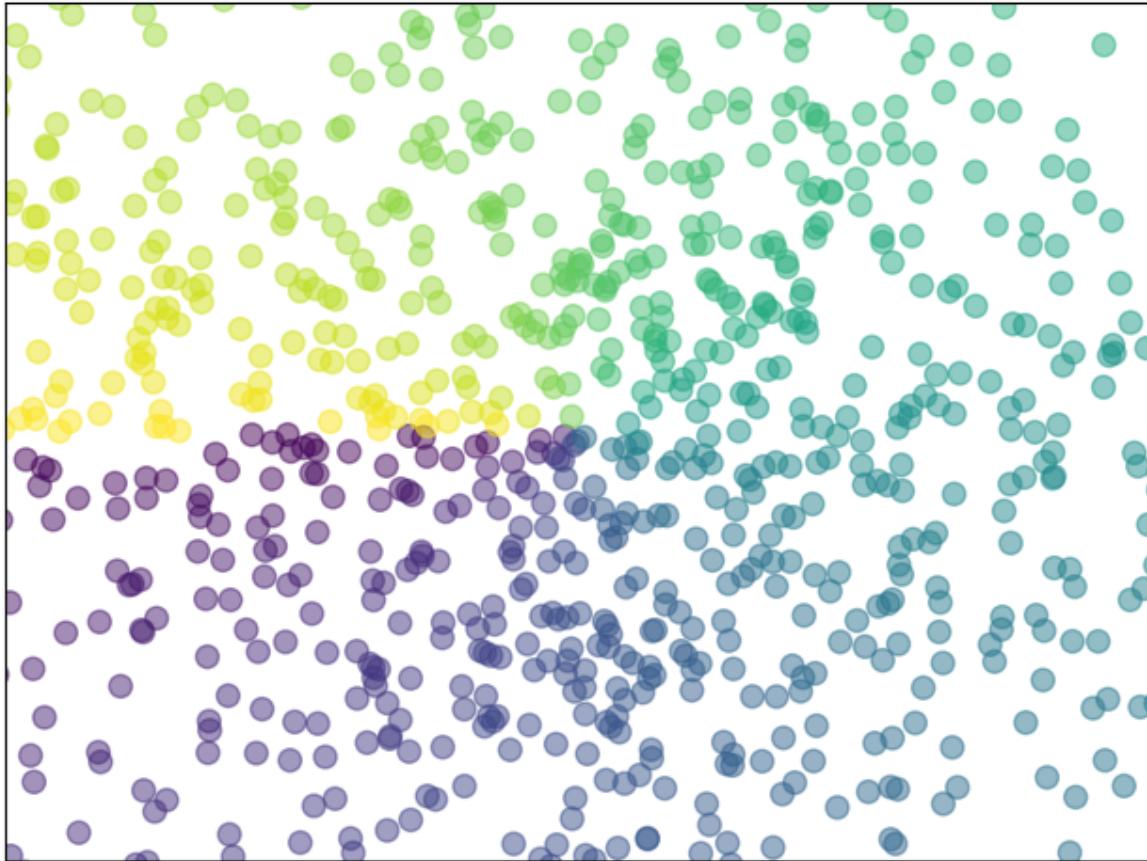


Pista: sólo necesitás modificar los ejes en la primera línea. Fijate que hay un parámetro `polar` que tiene por omisión valor `False`.

Ejercicio 6.13: 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)
```



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

6.7 Cierre de la sexta clase

En esta sexta 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 6.4](#).
 2. El archivo `informe.py` del [Ejercicio 6.5](#).
 3. El archivo `documentacion.py` del [Ejercicio 6.8](#).
 4. El archivo `random_walk.py` del [Ejercicio 6.10](#).
- Que completes [este formulario](#) 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!

7. Fechas, Carpetas y Pandas

En esta clase introducimos el módulo `datetime` para manejar datos relacionados con el tiempo (Sección 1) y un par de funciones del módulo `os` para leer directorios, procesar archivos y realizar algunas tareas relacionadas con el sistema operativo (Sección 2). Luego te proponemos integrar esto para hacer un script que corra desde línea de comandos y te permita ordenar los archivos de cierto tipo (Sección 3).

En la segunda mitad introducimos el módulo Pandas y el tipo `DataFrame` así como algunos de sus métodos elementales. Usamos pandas para analizar dos datasets de Arbolado Porteño graficando algunos de sus datos. En esta parte tenés que comparar características de árboles que crecen en los parques con otros que crecen en las veredas (Sección 4 y ejercicio de revisión por pares).

En la parte, Sección 5, te proponemos analizar ondas de mareas en el Río de la Plata como excusa para introducir el procesamiento de series temporales. Nos metemos un poco en temas específicos con una última parte optativa que incluye un breve análisis de la transformada de Fourier para medir el tiempo que tarda esta onda de marea en trasladarse de un puerto a otro.

- [7.1 Manejo de fechas y horas](#)

- [7.2 Manejo de archivos y carpetas](#)
- [7.3 Ordenar archivos en Python](#)
- [7.4 Introducción a Pandas](#)
- [7.5 Series temporales](#)
- [7.6 Cierre de la séptima](#)

7.1 Manejo de fechas y horas

El módulo datetime

A continuación introducimos el módulo `datetime` que permite trabajar con fechas y horas. Este módulo define un nuevo tipo de objeto: `datetime` (sí, con el mismo nombre del módulo), que permite representar un instante temporal (fecha y hora). También define objetos de tipo `date` para representar sólo una fecha y de tipo `time` para guardar y trabajar con horarios. Finalmente, en esta breve introducción al módulo `datetime` mencionamos el tipo `timedelta` que se usa para representar diferencias entre instantes de tiempos, es decir, duraciones y trabajar con ellas.

Ejemplo: Obtener fecha y hora actuales

```
>>> import datetime

>>> fecha_hora = datetime.datetime.now()
>>> print(fecha_hora)
2020-09-24 10:03:18.636670
```

Lo que hicimos fue importar el módulo `datetime` y usar el método `now()` de la clase `datetime` del módulo (con el mismo nombre) para crear el objeto `fecha_hora` que va a contener la fecha y la hora actuales.

Ejemplo: Obtener fecha actual

Análogamente, podemos obtener solo la fecha:

```
>>> fecha = datetime.date.today()
>>> print(fecha)
2020-09-24
```

Acá usamos el método `today()` de la clase `date` para obtener la fecha actual.

¿Qué hay dentro del módulo datetime?

En Python podemos usar la función `dir()` para obtener una lista de todos los atributos de un módulo.

```
>>> print(dir(datetime))
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__spec__', '_divide_and_round',
 'date', 'datetime', 'datetime_CAPI', 'time', 'timedelta', 'timezone',
 'tzinfo']
```

Nos vamos a concentrar en lo más usado de `datetime`:

- `date`
- `time`
- `datetime`
- `timedelta`

La clase `datetime.date`

Podés generar objetos de tipo fecha con la clase `date`. Un objeto de esta clase representa una fecha (año, mes, día).

Ejemplo: Un objeto para representar una fecha

```
>>> d = datetime.date(2019, 4, 13)
>>> print(d)
2019-04-13
```

El comando `date()` de este ejemplo construye un objeto de tipo `date`. Este constructor toma tres argumentos: año, mes y día.

La variable `d` es un objeto de tipo `date` (es decir, representa una fecha).

También podríamos importar directamente la clase `date` del módulo `datetime`:

```
>>> from datetime import date
>>>
>>> d = date(2019, 4, 13)
>>> print(d)
2019-04-13
```

Ejemplo: Obtener la fecha a partir de un timestamp

En los sistemas operativos derivados de Unix (Mac OS X, Linux, etc.) se toma como medida de tiempo el número de segundos transcurridos desde el primero de enero de 1970 a las 0 horas UTC hasta el momento a representar. Se lo conoce como Unix timestamp. Podés convertir un timestamp a fecha usando el método `fromtimestamp()`.

```
>>> from datetime import date  
>>>  
>>> timestamp = date.fromtimestamp(1326244364)  
>>> print('Fecha =', timestamp)  
Fecha = 2012-01-10
```

Esto es importante porque las fechas de modificación de los archivos usan timestamps por ejemplo.

Ejemplo: Obtener el año, el mes y el día por separado.

Así podés obtener el año, el mes, el día y el día de la semana:

```
from datetime import date  
  
hoy = date.today()  
  
print('Año actual:', hoy.year)  
print('Mes actual:', hoy.month)  
print('Día actual:', hoy.day)  
print('Día de la semana:', hoy.weekday()) # va de 0 a 6 empezando en lunes
```

La clase `datetime.time`

Un objeto de la clase `time` representa la hora local (de como este configurada tu computadora). No nos vamos a meter en esta clase con los [husos horarios](#) (conocido también como timezones), pero si vas a usar datos provistos por otros, es importante que sepas si está expresado en tu hora local, en la hora local de otro lugar o en [UTC](#).

Ejemplo: Representar la hora con un objeto `time`

La clase `time` se usa para representar horarios. A continuación damos algunos ejemplos de constructores de esta clase (un constructor es una forma de construir un objeto de una clase dada, una forma de inicializarlo, digamos).

```
>>> from datetime import time  
>>>  
>>> a = time()          # time(hour = 0, minute = 0, second = 0)
```

```

>>> print('a =', a)
a = 00:00:00

>>> b = time(11, 34, 56)
>>> print('b =', b)
b = 11:34:56

>>> c = time(hour = 11, minute = 34, second = 56)
>>> print('c =', c)
c = 11:34:56

>>> d = time(11, 34, 56, 234566) # time(hour, minute, second, microsecond)
>>> print('d =', d)
d = 11:34:56.234566

```

Ejemplo: Obtener horas, minutos, segundos y micro-segundos

Una vez que creaste un objeto `time`, podés extraer sus atributos así:

```

from datetime import time

a = time(11, 34, 56)

print('hour =', a.hour)
print('minute =', a.minute)
print('second =', a.second)
print('microsecond =', a.microsecond)

```

Como no le pasaste ningún valor para el argumento `microsecond`, éste va a tomar el valor predeterminado, que es 0.

La clase `datetime.datetime`

Como ya mencionamos, el módulo `datetime` tiene una clase con su mismo nombre que permite almacenar información de fecha y hora en un solo objeto.

Ejemplo: Objeto `datetime`

```

>>> from datetime import datetime

>>> # datetime(year, month, day)
>>> a = datetime(2020, 4, 21)
>>> print(a)
2020-04-21 00:00:00

```

```
>>> # datetime(year, month, day, hour, minute, second, microsecond)
>>> b = datetime(2021, 4, 21, 6, 53, 31, 342260)
>>> print(b)
2021-04-21 06:53:31.342260
```

Los primeros tres argumentos, `year`, `month` y `day` del constructor `datetime()` son obligatorios. Los otros tienen a 0 como valor por omisión.

Ejemplo: Obtener año, mes, día, hora, minutos, timestamp de un datetime

El siguiente código genera un objeto `datetime` con valores pasados por parámetro y luego imprime la información.

En particular, muestra cómo convertir una fecha a timestamp. En general los timestamps son enteros y no tienen en cuenta las décimas de segundos.

```
from datetime import datetime

a = datetime(2021, 4, 21, 6, 53, 31, 342260)
print('año =', a.year)
print('mes =', a.month)
print('día =', a.day)
print('hora =', a.hour)
print('minuto =', a.minute)
print('timestamp =', a.timestamp())
```

La clase `datetime.timedelta`

Un objeto `timedelta` representa una duración, es decir, la diferencia entre dos instantes de tiempo.

Ejemplo: Diferencia entre fechas y horarios

```
>>> from datetime import datetime, date

>>> t1 = date(year = 2021, month = 4, day = 21)
>>> t2 = date(year = 2020, month = 8, day = 23)
>>> t3 = t1 - t2
>>> print(t3)
241 days, 0:00:00

>>> t4 = datetime(year = 2020, month = 7, day = 12, hour = 7, minute = 9,
second = 33)
>>> t5 = datetime(year = 2021, month = 6, day = 10, hour = 5, minute = 55,
second = 13)
```

```

>>> t6 = t4 - t5
>>> print(t6)
-333 days, 1:14:20

>>> print('tipo de t3 =', type(t3))
tipo de t3 = <class 'datetime.timedelta'>

>>> print('tipo de t6 =', type(t6))
tipo de t6 = <class 'datetime.timedelta'>

```

Observá que t3 y t6 son de tipo <class 'datetime.timedelta'>.

Ejemplo: Diferencia entre objetos timedelta

```

>>> from datetime import timedelta

>>> t1 = timedelta(weeks = 1, days = 2, hours = 1, seconds = 33)
>>> t2 = timedelta(days = 6, hours = 11, minutes = 4, seconds = 54)
>>> t3 = t1 - t2

>>> print('t3 =', t3)
2 days, 13:55:39

```

t3 también es de tipo <class 'datetime.timedelta'>.

Ejemplo: Imprimir objetos timedelta negativos

```

>>> from datetime import timedelta

>>> t1 = timedelta(seconds = 21)
>>> t2 = timedelta(seconds = 55)
>>> t3 = t1 - t2

>>> print(t3)
-1 day, 23:59:26

>>> print(abs(t3))
0:00:34

```

Ejemplo: Duración en segundos

Podés obtener el tiempo medido en segundos usando el método total_seconds().

```

>>> from datetime import timedelta

>>> t = timedelta(days = 1, hours = 2, seconds = 30, microseconds = 100000)
>>> print('segundos totales =', t.total_seconds())
segundos totales = 93630.1

```

También podés sumar fechas y horarios usando el operador `+`. También podés multiplicar o dividir un objeto `timedelta` por números enteros o floats.

Formato para fechas y horas

Hay diversas formas de representar el tiempo, que varían según el lugar, la organización, etc. Por ejemplo, en Argentina solemos usar `dd/mm/yyyy`, mientras que en las culturas anglosajonas es más común usar `mm/dd/yyyy`.

En Python tenemos los métodos `strftime()` y `strptime()` para manejar esto.

Python `strftime()` - convertir un objeto `datetime` a string

El método `strftime()` está definido en las clases `date`, `datetime` y `time`. Este método crea una cadena con formato a partir estos objetos.

Ejemplo: Formato de fecha usando `strftime()`

```
>>> from datetime import datetime

>>> now = datetime.now()

>>> t = now.strftime('%H:%M:%S')
>>> print('hora:', t)
hora: 14:40:06

>>> s1 = now.strftime('%m/%d/%Y, %H:%M:%S')
>>> # en formato mm/dd/YY H:M:S
>>> print('s1:', s1)
s1: 09/24/2020, 14:40:06

>>> s2 = now.strftime('%d/%m/%Y, %H:%M:%S')
>>> # en formato dd/mm/YY H:M:S
>>> print('s2:', s2)
s2: 24/09/2020, 14:40:06
```

Acá, `%Y`, `%m`, `%d`, `%H` etc. son códigos de formato. El método `strftime()` toma uno o más códigos de formato y devuelve la cadena con formato basado en esos códigos.

En el programa de arriba, `t`, `s1` y `s2` son cadenas. Y los códigos de formato son:

- `%Y` - año [0001,..., 2018, 2019,..., 9999]
- `%m` - mes [01, 02, ..., 11, 12]

- %d - día [01, 02, ..., 30, 31]
- %H - hora [00, 01, ..., 22, 23]
- %M - minuto [00, 01, ..., 58, 59]
- %S - segundo [00, 01, ..., 58, 59]

Para aprender más sobre `strftime()` visitá [la documentación](#).

Python `strptime()` - convertir una cadena a un objeto `datetime`

El método `strptime()` crea un objeto `datetime` a partir de una cadena.

Ejemplo: `strptime()`

```
>>> from datetime import datetime

>>> cadena_con_fecha= '21 September, 2021'
>>> print('date_string =', cadena_con_fecha)
date_string = 21 September, 2021

>>> date_object = datetime.strptime(cadena_con_fecha, '%d %B, %Y')
>>> print('date_object =', date_object)
date_object = 2021-09-21 00:00:00
```

El método `strptime()` toma dos argumentos:

- una cadena que representa una fecha y hora
- un código de formato correspondiente al primer argumento

Los códigos de formato `%d`, `%B`, `%Y` significan `day`, `month (full name)` y `year` respectivamente.

Visitá [la documentación](#) para más detalles.

Ejercicios:

Ejercicio 7.1: Segundos vividos

Escribí una función a la que le pasás tu fecha de nacimiento como cadena en formato 'dd/mm/AAAA' (día, mes, año con 2, 2 y 4 dígitos, separados con barras normales) y te devuelve la cantidad de segundos que viviste (asumiendo que naciste a las 00:00hs de tu fecha de nacimiento).

Guardá este código en el archivo `vida.py`.

Ejercicio 7.2: Cuánto falta

Un conocido canal Argentino tiene por costumbre anunciar la cantidad de días que faltan para la próxima primavera.



Escribí un programa que asista a los técnicos del canal indicándoles, al correr el programa el número que deben poner en la placa.

Ejercicio 7.3: Fecha de reincorporación

Si tenés una licencia por xaternidad que empieza el 26 de septiembre de 2020 y dura 200 días, ¿qué día te reincorporás al trabajo?

Ejercicio 7.4: Días hábiles

Escribí una función `dias_habiles(inicio, fin, feriados)` que calcule los días hábiles entre dos fechas dadas. La función debe tener como argumentos el día inicial, el día final, y una lista con las fechas correspondientes a los feriados que haya en ese lapso, y debe devolver una lista con las fechas de días hábiles del

período, incluyendo la fecha inicial y la fecha final indicadas. Las fechas de entrada y salida deben manejarse en formato de texto.

Consideramos día hábil a un día que no es feriado ni sábado ni domingo.

Probala entre hoy y el 10 de octubre, sabiendo que no hay feriados en el medio.

Probala entre hoy y fin de año considerando los siguientes feriados de Argentina:

```
feriados = ['12/10/2020', '23/11/2020', '7/12/2020', '8/12/2020',
'25/12/2020']
```

7.2 Manejo de archivos y carpetas

Manejo de archivos y directorios

Una carpeta o directorio es una colección de archivos y directorios. Python tiene el módulo `os` que ofrece muchas herramientas útiles para trabajar con directorios y archivos.

En esta sección vas a aprender cómo crear un directorio, renombrarlo, listar todos sus archivos y subdirectorios, etc.

Obtener el directorio actual

Para obtener el directorio de trabajo actual, usamos el función `getcwd()` (*get current working directory*) del módulo `os`.

Esta función te devuelve el directorio actual en forma de cadena.

```
>>> import os
>>> os.getcwd()
'/home/usuario/ejercicios_python'
```

Es importante ver que la salida dependerá del sistema operativo que estés usando.

Por ejemplo, en Windows podrías obtener algo así:
`C:\\usuario\\ejercicios_python.`

Cambiar el directorio de trabajo

Podés cambiar de directorio usando la función `chdir()` (*change directory*). Los directorios pueden ser relativos o absolutos (en sistemas operativos basados en Unix '.' es el directorio actual, '..' es el anterior, '/' es el directorio raíz).

```
>>> os.chdir('./Data')                      # entro en Data, subdirectorio del
actual
>>> print(os.getcwd())
/home/usuario/ejercicios_python/Data
>>> os.chdir('..')                          # subo un nivel
>>> os.chdir('..')                          # subo otro nivel
>>> print(os.getcwd())
/home/usuario/
>>> os.chdir('/home')
>>> print(os.getcwd())
/home
```

Para cambiar de directorio, le pasás el nuevo directorio como cadena a esta función.

En diferentes sistemas operativos las barras de directorio se escriben de diferentes maneras. Es recomendable usar el comando `os.path.join` como en el siguiente ejemplo de manera que tu código funcione independientemente del sistema operativo en el que se lo corra.

```
>>> directorio = os.path.join('/home', 'usuario', 'ejercicios_python')
>>> os.chdir(directorio)
```

En caso de usarlo en Windows, será similar a:

```
>>> directorio = os.path.join('c:\\\\', 'usuario', 'ejercicios_python')
>>> os.chdir(directorio)
```

Usar directorios relativos al actual (que comienzan con '.') y no absolutos (que comienzan con '/') facilita la portabilidad del código de una compu a otra.

Listar directorios y archivos

La función `listdir()` toma un directorio (*path* o camino) y devuelve una lista con todos los archivos y subdirectorios de un directorio. Si no se le pasa ningún parámetro, devuelve los del directorio de trabajo actual.

```
>>> os.getcwd()
'/home/usuario/ejercicios_python'
>>> os.listdir('Data')

['camion2.csv',
 'missing.csv',
 'precios.csv',
```

```
'camion.csv',
'camion.dat',
'temperaturas.npy',
'camion blancos.csv',
'camion.csv.gz',
'dowstocks.csv',
'fecha_camion.csv',
'arbolado-en-espacios-verdes.csv']
```

Crear un nuevo directorio

Podés crear un directorio con la función `mkdir()`.

Esta función toma como argumento el path del nuevo directorio. Si no se especifica el path absoluto, el directorio nuevo se crea en el directorio de trabajo actual.

```
>>> os.mkdir('test')          # creo el directorio test
>>> os.mkdir(os.path.join('test', 'carpeta'))    # creo el subdirectorio
carpeta dentro de test
>>> os.listdir('test')
['carpeta']
```

Renombrar un directorio o un archivo

Para renombrar un directorio o archivo, la función `rename()` toma dos argumentos, el viejo nombre y el nuevo nombre.

```
>>> os.chdir('test')          # entro en el directorio test
>>> os.listdir()
['carpeta']
>>> os.rename('carpeta','carpeta_nueva') # cambio el nombre de carpeta
>>> os.listdir()
['carpeta_nueva']
```

Esto mismo se puede hacer trabajando desde el directorio `Ejercicios`, sin entrar en 'test', marcando el camino hacia la carpeta que queremos renombrar.

```
>>> os.chdir('..')           # subo un nivel
>>> os.listdir('test')        # miro qué hay en test
['carpeta_nueva']
>>>         os.rename(os.path.join('test',
os.path.join('test','carpeta_vieja')), 'carpeta_nueva'),
>>> os.listdir('test')
['carpeta_vieja']
```

La función `rename()` también es útil para mover un archivo o directorio, cambiando el camino (*path*) de acceso al archivo. Probá hacer esto:

```
>>> os.rename(os.path.join('test','carpeta_vieja'), 'carpeta_nueva') #  
cambio el path  
>>> os.listdir('test') # test quedó vacío  
[]
```

La carpeta 'carpeta' ahora se encuentra en 'Ejercicios', y no dentro de 'Ejercicios/test'.

Ojo: `rename()` funciona cuando el archivo (o directorio) no se cambia de disco (o más específicamente de una partición). Si querés mover un archivo del disco a un pendrive, por ejemplo, lo correcto es copiar el archivo al pendrive y luego borrarlo del disco. `rename()` no hace esto: no copia y borra, simplemente cambia el nombre. Para renombrar en caso que se pueda o copiar y borrar si lo primero no es posible, podés usar la función `move()` del módulo `shutil`. Este módulo es de más alto nivel y usa las primitivas de bajo nivel del módulo `os`. Al usar `os` tenemos un control más estricto de las operaciones. Las funciones de `shutil` pueden resultar más cómodas, pero en el camino pueden invocar a diversas funciones de bajo nivel del módulo `os`.

Eliminar un directorio o un archivo

⚠ A continuación usaremos comandos que borran archivos sin pasar por ninguna papelera de reciclaje. Estas acciones no pueden deshacerse: Usar con precaución, *un gran poder conlleva una gran responsabilidad*.

Podés eliminar un archivo usando la función `remove()`. También podés eliminar un directorio vacío usando `rmdir()`.

En el siguiente código trabajaremos en una carpeta que tiene esta estructura:

```
otra_carpeta  
└── archivo.txt  
└── subcarpeta
```

```
>>> os.chdir('otra_carpeta') # entro otra carpeta que tiene  
# una subcarpeta y un archivo de texto  
>>> os.listdir()  
['subcarpeta', 'archivo.txt']  
  
>>> os.remove('archivo.txt') # elimino el archivo  
>>> os.listdir()  
['subcarpeta']  
  
>>> os.rmdir('subcarpeta') # elimino la subcarpeta
```

```
>>> os.listdir()  
[]
```

Ojo: `rmdir()` solamente puede borrar directorios si están vacíos. Para eliminar un directorio no vacío, podés usar `rmtree()` del módulo `shutil`.

```
>>> os.mkdir(os.path.join('test','carpeta')) # creo  
nuevamente una carpeta # dentro de  
test  
>>> os.mkdir(os.path.join('test','carpeta', 'subcarpeta')) # creo una  
subcarpeta en carpeta  
>>> os.chdir('test') # entro en test  
>>> os.rmdir('carpeta') # quiero  
eliminar carpeta  
Traceback (most recent call last):  
  
File "<ipython-input-277-c4255042d84c>", line 1, in <module>  
    os.rmdir('carpeta')  
  
OSError: [Errno 39] Directory not empty: 'carpeta'  
  
>>> import shutil  
  
>>> shutil.rmtree('carpeta')  
>>> os.listdir()  
[]
```

Recorriendo directorios con `os.walk()`

La función `walk()` del módulo `os` genera una lista con los nombres de todos los archivos del árbol de subdirectorios de un directorio dado. Es decir, lista los archivos de un directorio dado y luego entra en cada subdirectorio y hace lo mismo, recursivamente (*top-down*).

La función `walk()` recibe como único parámetro obligatorio el directorio donde comenzar a mirar (la raíz del árbol).

Ejemplo

En este ejemplo se ve cómo se usa `os.walk()`.

```
import os  
for root, dirs, files in os.walk("."):  
    for name in files:  
        print(os.path.join(root, name))  
    for name in dirs:
```

```
    print(os.path.join(root, name))
```

Cambiar atributos de un archivo

Dependiendo del sistema operativo, un archivo puede tener asociadas diferentes fechas (de creación original, de modificación del contenido, de cambio en sus metadatos, de acceso para lectura).

Vamos a cambiar la fecha de modificación de un archivo. Para ello necesitás importar `os` y `datetime`. Después, converís la fecha elegida a `timestamp` y se la asocías al archivo con `utime`, como se muestra acá abajo:

```
import os
import datetime
import time

camino = './rebotes.py'

stats_archivo = os.stat(camino)
print(time.ctime(stats_archivo.st_atime))

fecha_acceso = datetime.datetime(year = 2017, month = 9, day = 21, hour =
19, minute = 51, second = 0)
fecha_modifi = datetime.datetime(year = 2012, month = 9, day = 24, hour =
12, minute = 9, second = 24)

ts_acceso = fecha_acceso.timestamp()
ts_modifi = fecha_modifi.timestamp()
os.utime(camino, (ts_acceso, ts_modifi))

stats_archivo = os.stat(camino)
print(time.ctime(stats_archivo.st_atime))
```

Si mirás la información del archivo `./rebotes.py` desde tu gestor de archivos deberías ver las modificaciones.

7.3 Ordenar archivos en Python

En esta sección vamos a integrar las últimas dos secciones con lo que veníamos viendo antes del parcial. La idea es que descomprimas [este archivo](#) en tu carpeta `Data/` y escribas un script que trabaje con estos archivos.

Esta sección tiene un ejercicio para entregar y luego otro más complejo que es optativo.

Ejercicio 7.5: Recorrer el árbol de archivos

Escribí un programa que dado un directorio, imprima en pantalla los nombres de todos los archivos .png que se encuentren en *algún* subdirectorio del él.

Observación: Al final, tu script debería poder ejecutarse desde la línea de comandos recibiendo como parámetro el directorio a leer original. En la [Sección 6.2](#) dimos un modelo de script que te puede servir.

Guardá el script resultante en un archivo `listar_imgs.py`.

Ejercicio 7.6: Ordenar el árbol de archivos (optativo)

Escribí un programa que te permita ordenar las imágenes PNG de esta carpeta. Guardalo en un archivo `ordenar_imgs.py`.

1. Creá un nuevo directorio `Data/imgs_procesadas/`.
2. Usá `os.walk()` para recorrer los archivos en la carpeta `Data/ordenar/` (y sus subcarpetas).
3. Cuando encuentres archivos con extensión `.png` los vas a *procesar*. En este caso *procesar* significa:
 - Leer la fecha que se encuentra codificada en los últimos 8 caracteres de su nombre en el formato AAAAMMDD (año en 4 dígitos, mes en 2 y día en 2).
 - Usar la fecha obtenida para setear la fecha de última modificación (y de último acceso si no usás Windows).
 - Cambiarle el nombre al archivo para que no tenga más esos dígitos (ni el guión bajo).
 - Mover el archivo a la carpeta `Data/imgs_procesadas/`.
4. Los archivos que no son `.png` no los modifiques.
5. Borrá todas las subcarpetas de `Data/ordenar/` que hayan quedado vacías.

Observación: Al final, tu script debería poder ejecutarse desde la línea de comandos recibiendo como parámetro el directorio a leer original y un directorio destino (que debería ser creado si no existe).

Observación: Este tipo de tareas se repite con mucha frecuencia. Tener la capacidad de automatizarlas mediante un script de Python te puede ahorrar muchísimo tiempo.

Algunos puntos importantes:

- Te recomendamos que modularices el procesamiento de los archivos `png`. Podés, por ejemplo, escribir una función que manipule strings (que tome el nombre de un archivo y devuelva la fecha y el nombre corregido) y otra función que precese cada archivo (que use la función anterior para renombrar, mover y modificar la fecha de cada archivo). La modularización del código es clave para que otras personas lo puedan entender y que sea sencillo de mantener.
- Usá docstrings y comentarios en tu código de manera que sea legible.

7.4 Introducción a Pandas

La biblioteca Pandas es una extensión de NumPy para manipulación y análisis de datos. En particular, ofrece estructuras de datos y operaciones para manipular tablas de datos (numéricos y de otros tipos) y series temporales. Se distribuye como software libre.

Ésta es una breve introducción a [Pandas](#). Para información más completa, te recomendamos consultar [la documentación oficial](#).

Esta biblioteca tiene dos tipos de datos fundamentales: los `DataFrames` que almacenan tablas de datos y las `series` que contienen secuencias de datos.

Lectura de datos

Pandas permite leer diversos formatos de tablas de datos directamente. Probá el siguiente código, para leer un archivo CSV:

```
import pandas as pd
import os

directorio = 'Data'
archivo = 'arbolado-en-espacios-verdes.csv'
fname = os.path.join(directorio, archivo)
df = pd.read_csv(fname)
```

La variable `df` es de tipo `DataFrame` y contiene todos los datos del archivo csv estructurados adecuadamente.

Con `df.head()` podés ver las primeras líneas de datos. Si a `head` le pasás un número como parámetro podés seleccionar cuántas líneas querés ver. Análogamente con `df.tail(n)` verás las últimas `n` líneas de datos.

```
>>> df.head()
```

	long	lat	id_arbol	...	origen	coord_x
coord_y						
0	-58.477564	-34.645015	1	...	Exótico	98692.305719
98253.300738						
1	-58.477559	-34.645047	2	...	Exótico	98692.751564
98249.733979						
2	-58.477551	-34.645091	3	...	Exótico	98693.494639
98244.829684						
3	-58.478129	-34.644567	4	...	Nativo/Autóctono	98640.439091
98302.938142						
4	-58.478121	-34.644598	5	...	Nativo/Autóctono	98641.182166
98299.519997						

Usando `df.columns` pandas te va a devolver un índice con los nombres de las columnas del DataFrame. Recordá que en la [Sección 2.7](#) describimos la base de datos. A su vez, `df.index` te mostrará el índice. En este caso el índice es numérico y se corresponde con el número de la línea leída del archivo. En principio no es muy interesante para analizar cuestiones de árboles, simplemente tenemos las filas numeradas. Veremos otros ejemplos donde el índice puede contener información vital (una categoría, un timestamp, etc).

```
>>> df.columns
Index(['long', 'lat', 'id_arbol', 'altura_tot', 'diametro', 'inclinacio',
       'id_especie', 'nombre_com', 'nombre_cie', 'tipo_folla',
       'espacio_ve',
       'ubicacion', 'nombre_fam', 'nombre_gen', 'origen', 'coord_x',
       'coord_y'],
      dtype='object')
>>> df.index
RangeIndex(start=0, stop=51502, step=1)
```

Otra herramienta útil para inspeccionar los datos recién levantados es `describe()`. Para ver mejor una parte, podemos seleccionar algunas columnas de interés antes de pedirle la descripción.

```
>>> df[['altura_tot', 'diametro', 'inclinacio']].describe()
            altura_tot      diametro      inclinacio
count    51502.000000  51502.000000  51502.000000
mean     12.167100    39.395616    3.472215
std      7.640309    31.171205    7.039495
min      0.000000    1.000000    0.000000
25%     6.000000    18.000000    0.000000
50%    11.000000    32.000000    0.000000
```

```
75%      18.000000    54.000000    5.000000
max      54.000000   500.000000   90.000000
```

Selección

Una de las operaciones primitivas más importantes es la selección de fragmentos de las tablas de datos, ya sean filas, columnas o rangos de filas y columnas.

Por ejemplo con `df['nombre_com']` veremos la columna (que es una serie) de nombres comunes de los árboles en la base. Podemos usar `unique` para ver una vez cada nombre:

```
>>> df['nombre_com'].unique()
array(['Washingtonia (Palmera washingtonia)', 'Ombú', 'Catalpa', 'Ceibo',
       'Brachichiton (Árbol botella, Brachichito)', 'Álamo plateado',
       'Acacia de constantinopla', 'Acacia', 'Roble sedoso (Grevillea)',
       ...
       'Jazmín del Paraguay', 'Plumerillo rojo', 'Árbol fuccia',
       'Canela de venado', 'Boj cepillo', 'Caranday'], dtype=object)
```

Podemos preguntar cuáles se llaman de cierta manera ('Ombú' en este caso), como hacíamos con los ndarrays en numpy:

```
>>> df['nombre_com'] == 'Ombú'
0      False
1      False
2      False
3      True
...
...
```

Observá que esto generó una serie. Podemos sumar los `True` de esta serie para contar la cantidad de Ombús:

```
>>> (df['nombre_com'] == 'Ombú').sum()
590
```

Si queremos hacer lo mismo para otras especies podemos usar `value_counts()`

```
>>> cant_ejemplares = df['nombre_com'].value_counts()
>>> cant_ejemplares.head(10)
Eucalipto          4112
Tipa blanca        4031
Jacarandá           3255
Palo borracho rosado  3150
Casuarina            2719
Fresno americano     2166
```

```

Plátano          1556
Ciprés           1467
Ceibo            1149
Pindó             1068
Name: nombre_com, dtype: int64

```

De esta forma obtenemos, en orden decreciente, los nombres comunes y las cantidades de las especies más frecuentes en la base de datos.

Filtros booleanos

La serie booleana que obtuvimos con `df['nombre_com'] == 'Ombú'` puede usarse para seleccionar esas filas del DataFrame. Probemos con Jacarandá:

```
>>> df_jacarandas = df[df['nombre_com'] == 'Jacarandá']
```

Análogamente, podemos seleccionar algunas columnas de interés y generar vistas (ojo, en estos casos no estamos copiando la información):

```

>>> cols = ['altura_tot', 'diametro', 'inclinacio']
>>> df_jacarandas = df_jacarandas[cols]
>>> df_jacarandas.tail()
    altura_tot  diametro  inclinacio
51104          7         97          4
51172          8         28          8
51180          2         30          0
51207          3         10          0
51375         17         40         20

```

```

>>> df_jacarandas.describe()
    altura_tot  diametro  inclinacio
count  3255.000000  3255.000000  3255.000000
mean   10.369585   28.804301   6.549923
std    5.905744   19.166388   8.459921
min    1.000000   1.000000   0.000000
25%    6.000000   14.000000   0.000000
50%   10.000000   25.000000   4.000000
75%   15.000000   41.000000  10.000000
max   49.000000  159.000000  70.000000

```

Observá que cuando le pedimos los últimos datos de `df_jacarandas` nos mostró los últimos 5 jacarandás de la base de datos, respetando los números de índice de la tabla original (... , 51207, 51375).

Si vas a querer modificar `df_jacarandas` es conveniente crear una copia de los datos de `df` en lugar de simplemente una vista. Esto se puede hacer con el método `copy()` como en el siguiente ejemplo.

```
>>> df_jacarandas = df[df['nombre_com'] == 'Jacarandá'][cols].copy()
```

Scatterplots

Pandas también permite [hacer gráficos bonitos](#). Es realmente sencillo:

```
df_jacarandas.plot.scatter(x = 'diametro', y = 'altura_tot')
```

Hay otro módulo para hacer gráficos que interactúa muy bien con pandas y se llama [Seaborn](#). Está basado en matplotlib, y ofrece una interfaz de alto nivel para realizar gráficos estadísticos atractivos e informativos. En criollo: "usar pandas para manejar los datos y seaborn para visualizarlos, es la posta".

Fijate que seaborn entiende los DataFrames y las columnas y su sintaxis es muy similar a la de pandas:

```
import seaborn as sns

sns.scatterplot(data = df_jacarandas, x = 'diametro', y = 'altura_tot')
```

Filtros por índice y por posición

Como ya mencionamos, el índice de `df` no tiene una semántica interesante. Veamos, en cambio, que la serie que generamos con `cant_ejemplares = df['nombre_com'].value_counts()` sí lo tiene:

```
>>> cant_ejemplares.index
Index(['Eucalipto', 'Tipa blanca', 'Jacarandá', 'Palo borracho rosado',
       'Casuarina', 'Fresno americano', 'Plátano', 'Ciprés', 'Ceibo',
       'Pindó',
       ...
       'Naranjo dulce', 'Peltophorum', 'Ligustrina de California',
       'Afrocarpus', 'Caranday', 'Esterculea', 'Boj cepillo', 'Sesbania',
       'Ligustrum', 'Árbol del humo'],
       dtype='object', length=337)
```

`cant_ejemplares` es una serie (es como un DataFrame de una sola columna). Tiene los nombres de las especies como índice y sus respectivas cantidades como dato asociado.

Podemos acceder a una fila de un DataFarme o una Serie tanto a través de su posición como a través de su índice. Para acceder con el índice usá `loc[]` como en los siguientes ejemplos:

```
>>> df.loc[165]
long                               -58.4684
```

```

lat                               -34.6648
id_arbol                           166
altura_tot                           5
diametro                            10
inclinacio                           0
id_especie                           11
nombre_com                          Jacarandá
nombre_cie                           Jacarandá mimosifolia
tipo_folla                          Árbol Latifoliado Caducifolio
espacio_ve                           INDOAMERICANO
ubicacion                           LACARRA, Av. - ESCALADA, Av. - CASTAÑARES, Av....
nombre_fam                           Bignoniáceas
nombre_gen                           Jacarandá
origen                                Nativo/Autóctono
coord_x                                99534.3
coord_y                                96061.8
Name: 165, dtype: object

```

```

>>> cant_ejemplares.loc['Eucalipto']
4112

```

Para acceder por número de posición usá `iloc`, como se muestra a continuación.

```

>>> df_jacarandas.iloc[0]
altura_tot      5
diametro       10
inclinacio     0
Name: 165, dtype: int64

```

Observá que esto nos devuelve los datos de la primera fila de `df_jacarandas` que corresponde al índice 165 (lo dice en la última línea). También podemos acceder a rebanadas (slices) usando `iloc`:

```

>>> cant_ejemplares.iloc[0:3]
Eucalipto      4112
Tipa blanca    4031
Jacarandá       3255
Name: nombre_com, dtype: int64

```

Por otra parte, podemos seleccionar tanto filas como columnas, si separamos con comas las respectivas selecciones:

```

>>> df_jacarandas.iloc[-5:,2]
51104      4
51172      8
51180      0
51207      0
51375     20

```

```
Name: inclinacio, dtype: int64
```

Esto nos devuelve los datos correspondientes a las últimas 5 filas y a la tercera columna ('inclinacio'). Fijate que siempre vienen acompañados del índice.

Selección de una columna

Si queremos seleccionar una sola columna podemos especificarla por medio de su nombre. Recordemos que al tomar una sola columna obtenemos una serie en lugar de un DataFrame:

```
>>> df_jacarandas_diam = df_jacarandas['diametro']
>>> type(df_jacarandas)
pandas.core.frame.DataFrame
>>> type(df_jacarandas_diam)
pandas.core.series.Series
```

Series temporales en Pandas

Pandas tiene un gran potencial para el manejo de series temporales. Es muy sencillo crear índices con fechas y frecuencias seleccionadas.

```
>>> pd.date_range('20200923', periods = 7)
DatetimeIndex(['2020-09-23', '2020-09-24', '2020-09-25', '2020-09-26',
               '2020-09-27', '2020-09-28', '2020-09-29'],
              dtype='datetime64[ns]', freq='D')

>>> pd.date_range('20200923 14:00', periods = 7)
DatetimeIndex(['2020-09-23 14:00:00', '2020-09-24 14:00:00',
               '2020-09-25 14:00:00', '2020-09-26 14:00:00',
               '2020-09-27 14:00:00', '2020-09-28 14:00:00',
               '2020-09-29 14:00:00'],
              dtype='datetime64[ns]', freq='D')

>>> pd.date_range('20200923 14:00', periods = 6, freq = 'H')
DatetimeIndex(['2020-09-23 14:00:00', '2020-09-23 15:00:00',
               '2020-09-23 16:00:00', '2020-09-23 17:00:00',
               '2020-09-23 18:00:00', '2020-09-23 19:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Luego, podés usar esos índices junto con datos para armar series temporales o DataFrames:

```
>>> pd.Series([1, 2, 3, 4, 5, 6], index = pd.date_range('20200923 14:00',
periods = 6, freq = 'H'))
2020-09-23 14:00:00    1
```

```
2020-09-23 15:00:00    2
2020-09-23 16:00:00    3
2020-09-23 17:00:00    4
2020-09-23 18:00:00    5
2020-09-23 19:00:00    6
Freq: H, dtype: int64
```

Caminatas al azar

Volviendo al tema de las caminatas al azar, podemos hacer una caminata de dos horas dando un paso por minuto a partir del comienzo de esta clase con el siguiente comando:

```
import numpy as np

idx = pd.date_range('20200923 14:00', periods = 120, freq = 'min')
s1 = pd.Series(np.random.randint(-1,2,120), index = idx)
s2 = s1.cumsum()
```

Observá que estamos usando random del módulo numpy, no de random. La función `np.random.randint(-1,2,120)` genera un array de longitud 120 con valores -1, 0, 1 (no incluye extremo derecho del rango de valores).

Podemos ver el gráfico sencillamente:

```
s2.plot()
```

O usar una **media móvil** (rolling mean) para suavizar los datos:

```
w = 5 # ancho en minutos de la ventana
s3 = s2.rolling(w).mean()
s3.plot()
```

Podés ver ambas curvas en un mismo gráfico para ver más claramente el efecto del suavizado:

```
df_series_23 = pd.DataFrame([s2, s3]).T # armo un dataframe con ambas series
df_series_23.plot()
```

Fijate que los datos de la curva suavizada empiezan más tarde, porque al principio no hay datos sobre los cuales hacer promedio. El parámetro `min_periods = 1` del método `rolling` te permite controlar esto. Probalo.

Ejemplo: 12 personas caminando 8 horas

En el siguiente ejemplo creamos un índice que contenga un elemento por minuto a partir del comienzo de la clase y durante 8 horas. Armamos también una lista de nombres.

```
horas = 8
idx = pd.date_range('20200923 14:00', periods = horas*60, freq = 'min')
nombres = ['Pedro', 'Santiago', 'Juan',
'Andrés', 'Bartolomé', 'Tiago', 'Isca', 'Tadeo', 'Mateo', 'Felipe', 'Simón', 'Tomás
']
```

Luego usamos el módulo random de numpy para generar pasos para cada persona a cada minuto. Los acumulamos con `cumsum` y los acomodamos en un DataFrame, usando el índice generado antes y poniéndoles nombres adecuados a cada columna:

```
df_walks =
pd.DataFrame(np.random.randint(-1,2,[horas*60,12])).cumsum(axis=0), index =
idx, columns = nombres)

df_walks.plot()
```

Ahora suavizamos los datos, usando `min_periods` para no perder los datos de los extremos.

```
w = 45
df_walk_suav = df_walks.rolling(w, min_periods = 1).mean() # datos
suavizados
nsuav = ['S_' + n for n in nombres]
df_walk_suav.columns = nsuav # cambio el nombre de las columnas
# para los datos suavizados
df_walk_suav.plot()
```

Guardando datos

Guardar una serie o un DataFrame en el disco es algo realmente sencillo. Probá, por ejemplo, el efecto del comando

```
df_walk_suav.to_csv('caminata_apostolica.csv').
```

Incorporando el Arbolado lineal

Ejercicio 7.7: Lectura y selección

Vamos a trabajar ahora con el archivo 'arbolado-publico-lineal-2017-2018.csv'. Descargalo y guardalo en tu directorio 'Data'.

Levantalo y armá un DataFrame `df_lineal` que tenga solamente las siguientes columnas:

```
cols_sel = ['nombre_cientifico', 'ancho_acera', 'diametro_altura_pecho',
'altura_arbol']
```

Imprimí las diez especies más frecuentes con sus respectivas cantidades.

Trabajaremos con las siguientes especies seleccionadas:

```
especies_seleccionadas = ['Tilia x moltkei', 'Jacaranda mimosifolia',
'Tipuana tipu']
```

Una forma de seleccionarlas es la siguiente:

```
df_lineal_seleccion = df_lineal[df_lineal['nombre_cientifico'].isin(especies_seleccionadas)]
```

Ejercicio 7.8: Boxplots

El siguiente comando realiza un `boxplot` de los diámetros de los árboles agrupados por especie.

```
df_lineal_seleccion.boxplot('diametro_altura_pecho', by = 'nombre_cientifico')
```

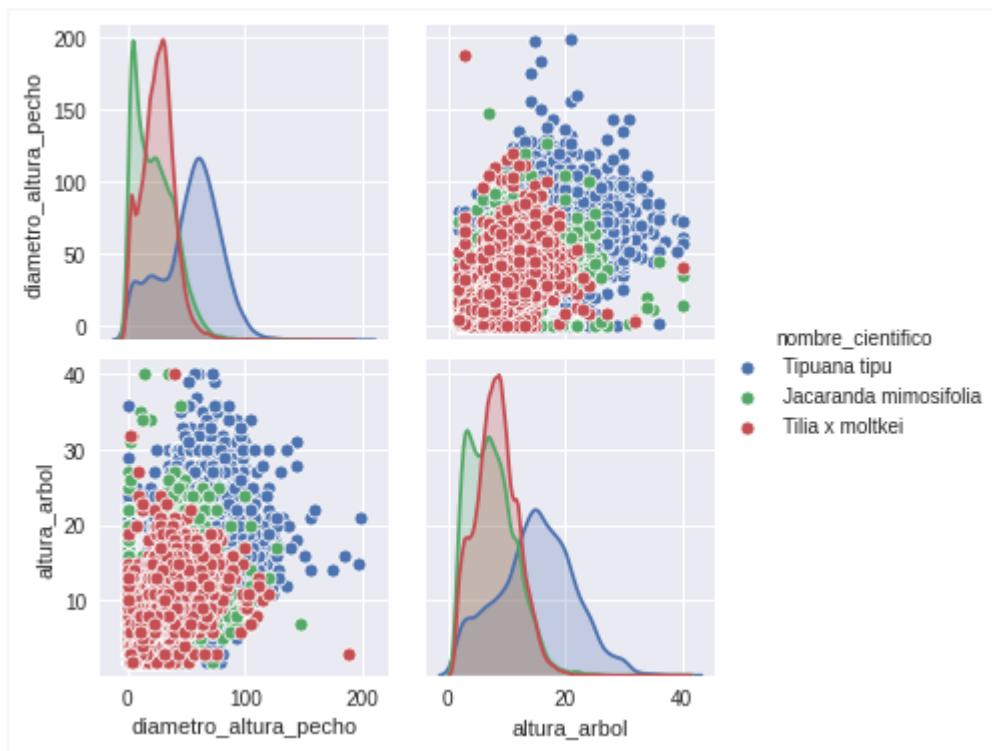
Realizá un gráfico similar pero de los altos en lugar de los diámetros de los árboles.

Ejemplo de pairplot

Otro gráfico interesante que resume muy bien la información es el `pairplot` de seaborn que es una grilla cuadrada de subplots.

Probá el siguiente código:

```
sns.pairplot(data = df_lineal_seleccion[cols_sel], hue = 'nombre_cientifico')
```



El gráfico va a tener una fila (y columna) por cada variable numérica en el DataFrame pasado como `data`. En la diagonal del gráfico, va a haber `kdeplots` (kernel density estimation plots, una versión suavizada de los histogramas) y fuera de la diagonal `scatterplots` combinando todos los pares de variables (cada combinación aparece dos veces, una sobre y otra debajo de la diagonal).

El `hue` selecciona la variable categórica a usar para distinguir subgrupos y asociarles colores. En la diagonal de este ejemplo (y en los scatterplots también) se ve por ejemplo que las Tipas suelen ser más anchas y más altas que los Tilos y los Jacarandás.

Pregunta: ¿Por qué el ancho_acera no tiene lugar en el gráfico?

Te recomendamos pegarle una mirada a [esta página](#) donde vas a poder ver un poco más sobre el potencial de seaborn.

Ejercicio 7.9: Comparando especies en parques y en veredas

Al comienzo de la materia estuvimos trabajando con el dataset de árboles en parques. Ahora estuvimos analizando otro dataset: el de árboles en veredas. Ahora queremos estudiar si hay diferencias entre los ejemplares de una misma especie según si crecen en un sitio o en otro. Queremos hacer un boxplot del diámetro a la altura del pecho para las Tipas (su nombre científico es *tipuana tipu*), que crecen en

ambos tipos de ambiente. Para eso tendremos que juntar datos de dos bases de datos diferentes.

Nos vamos en meter en un lío. El GCBA usa en un dataset 'altura_tot', 'diametro' y 'nombre_cie' para las alturas, diámetros y nombres científicos de los ejemplares, y en el otro dataset usa 'altura_arbol', 'diametro_altura_pecho' y 'nombre_cientifico' para los mismos datos.

Es más, los nombres científicos varían de un dataset al otro. 'Tipuana Tipu' se transforma en 'Tipuana tipu' y 'Jacarandá mimosifolia' en 'Jacaranda mimosifolia'. Obviamente son cambios menores pero suficientes para desalentar al usuarie desprevenide.

En este ejercicio te proponemos los siguientes pasos para comparar los diámetros a la altura del pecho de las tipas en ambos tipos de entornos. Guardá este trabajo en un archivo `arbolado_parques_veredas.py`.

1. Abrí ambos datasets a los que llamaremos `df_parques` y `df_veredas`.
2. Para cada dataset armate otro seleccionando solamente las filas correspondientes a las tipas (llamalos `df_tipas_parques` y `df_tipas_veredas`, respectivamente) y las columnas correspondientes al diámetro a la altura del pecho y alturas. Hacelo como copias (usando `.copy()` como hicimos más arriba) para poder trabajar en estos nuevos dataframes sin modificar los dataframes grandes originales. Renombrá las columnas que muestran la altura y el diámetro a la altura del pecho para que se llamen igual en ambos dataframes, para ello explorá el comando `rename`.
3. Agregale a cada dataframe (`df_tipas_parques` y `df_tipas_veredas`) una columna llamada 'ambiente' que en un caso valga siempre 'parque' y en el otro caso 'vereda'.
4. Juntá ambos datasets con el comando `df_tipas = pd.concat([df_tipas_veredas, df_tipas_parques])`. De esta forma tenemos en un mismo dataframe la información de las tipas distinguidas por ambiente.
5. Creá un boxplot para los diámetros a la altura del pecho de la tipas distinguiendo los ambientes (`boxplot('diametro_altura_pecho', by = 'ambiente')`).
6. Repetí para alturas.
7. ¿Qué tendrías que cambiar para repetir el análisis para otras especies? ¿Convendría definir una función?

7.5 Series temporales

Para esta Sección contamos con el valioso aporte de [Octavio Bruzzone](#). Octavio da dos cursos de posgrado excelentes sobre Series Temporales en Python. Uno se enfoca en los análisis en el dominio del tiempo y el otro en el dominio de las frecuencias. Generosamente nos compartió algunas ideas para este trabajo práctico.

Análisis y visualización de series temporales.

En este práctico vamos a visualizar y analizar datos de mareas en el Río de la Plata. Tiene una primera parte que esperamos que todos hagan y una segunda parte, más larga y compleja, optativa. Trabajá en el archivo `mareas_fft.py`.

Para comenzar, copiate [el archivo](#) con datos de mareas en los puertos de San Fernando y Buenos Aires a tu carpeta `Data`

Lectura de archivos temporales

```
import pandas as pd  
  
df = pd.read_csv('Data/OBS_SHN_SF-BA.csv')
```

Observá los datos:

```
>>> df.head()  
      Time  H_SF  H_BA  
0  2011-01-01 00:00:00    NaN  92.0  
1  2011-01-01 01:00:00    NaN 110.0  
2  2011-01-01 02:00:00    NaN 124.0  
3  2011-01-01 03:00:00    NaN 132.0  
4  2011-01-01 04:00:00    NaN 136.0  
  
>>> df.index  
RangeIndex(start=0, stop=35064, step=1)
```

Este archivo tiene alturas del agua en el puerto de San Fernando (columna `H_SF`) y en el puerto de Buenos Aires (columna `H_BA`) medidas en centímetros. Tiene un dato por hora (columna `Time`) durante cuatro años. En los primeros registros se observa algo muy frecuente con este tipo de archivos: tiene muchos datos faltantes.

El índice de un dataframe nos da información de su estructura. En este caso, está representando el número de línea del archivo que leímos. Pero un índice puede aportarnos más información relevante para nuestro problema, por lo que la

propuesta es que el índice debería ser el instante en le que se tomó cada muestra ('Time').

Para esto tenemos que decirle a la función `read_csv` dos cosas:

- por un lado que use la columna 'Time' como índice (`index_col = ['Time']`) y
- por el otro que la interprete como un timestamp (`parse_dates = True`).

```
df      = pd.read_csv('Data/OBS_SHN_SF-BA.csv',      index_col=['Time'],
parse_dates=True)
```

Observá la diferencia:

```
>>> df.head()
          H_SF    H_BA
Time
2011-01-01 00:00:00  NaN  92.0
2011-01-01 01:00:00  NaN  110.0
2011-01-01 02:00:00  NaN  124.0
2011-01-01 03:00:00  NaN  132.0
2011-01-01 04:00:00  NaN  136.0

>>> df.index
DatetimeIndex(['2011-01-01 00:00:00', '2011-01-01 01:00:00',
                 ...
                 '2014-12-31 22:00:00', '2014-12-31 23:00:00'],
                dtype='datetime64[ns]', name='Time', length=35064, freq=None)
```

Que el índice sea temporal nos da una versatilidad genial para trabajar con estos datos. Probá por ejemplo los siguientes comandos:

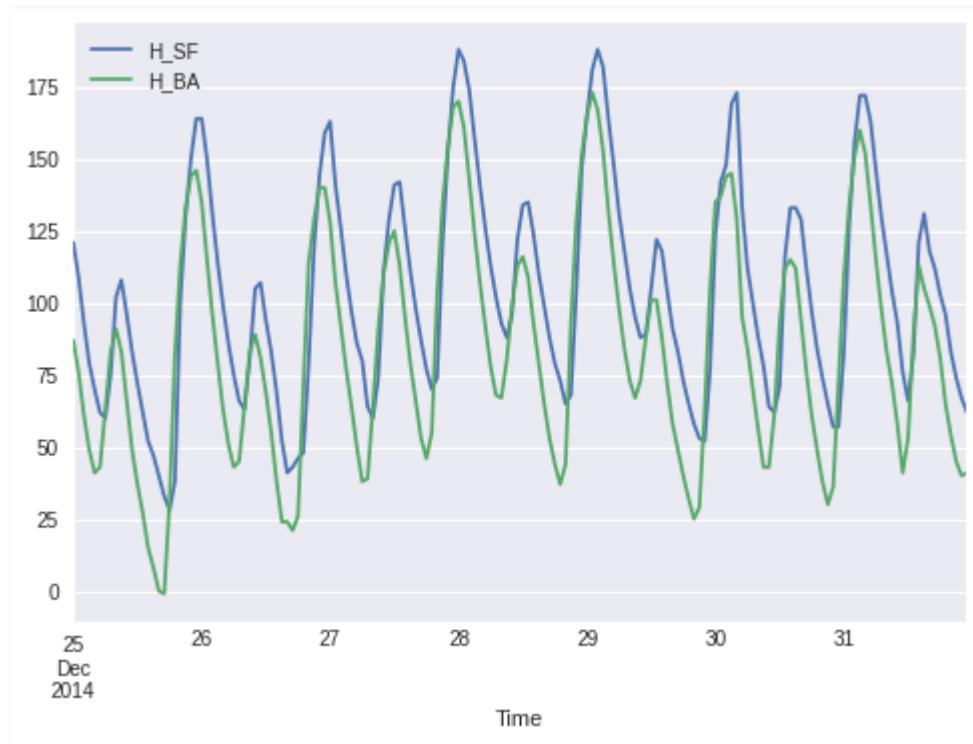
```
>>> df['1-18-2014 9:00':'1-18-2014 18:00']
          H_SF    H_BA
Time
2014-01-18 09:00:00  85.0  67.0
2014-01-18 10:00:00  79.0  60.0
2014-01-18 11:00:00  73.0  49.0
2014-01-18 12:00:00  65.0  43.0
2014-01-18 13:00:00  59.0  36.0
2014-01-18 14:00:00  53.0  29.0
2014-01-18 15:00:00  48.0  22.0
2014-01-18 16:00:00  42.0  18.0
2014-01-18 17:00:00  36.0  33.0
2014-01-18 18:00:00  40.0  67.0
```

Probá también `df['2-19-2014']` (observá que el formato de fechas que se usa es el de EEUU), y `df['12-25-2014':]`.

Ondas de marea en el Río de la Plata

Grafiquemos estos últimos datos:

```
df['12-25-2014':].plot()
```



Acá se ven tres fenómenos interesantes:

- Hay 14 picos en 7 días, esto corresponde a la frecuencia *semidiurna* de las mareas. Cada poco más de 12hs tenemos un ciclo con pleamar y bajamar. Dos ciclos por día.
- Por otra parte, se ve que las mareas en San Fernando están retrasadas respecto a las de Buenos Aires. Esto se debe a que las ondas de marea vienen del océano atlántico y se propagan por el estuario del río de la Plata, pasando primero por Buenos Aires y llegando luego, con retraso, a San Fernando. En ciertas condiciones esta onda de mareas puede llegar a la ciudad de Rosario, aunque se va atenuando en su viaje desde el atlántico.
- Finalmente, hay una marcada diferencia entre la altura registrada en San Fernando y la de Buenos Aires. Esto se debe a que las dos escalas, a partir de las que se registran los datos, tienen ceros que no están nivelados.

En este práctico nos proponemos estudiar la propagación de esta *onda de marea* que es generada por la atracción gravitacional que ejercen la luna y el sol sobre el agua. Vamos a usar una transformada de Fourier que nos permite estudiar las

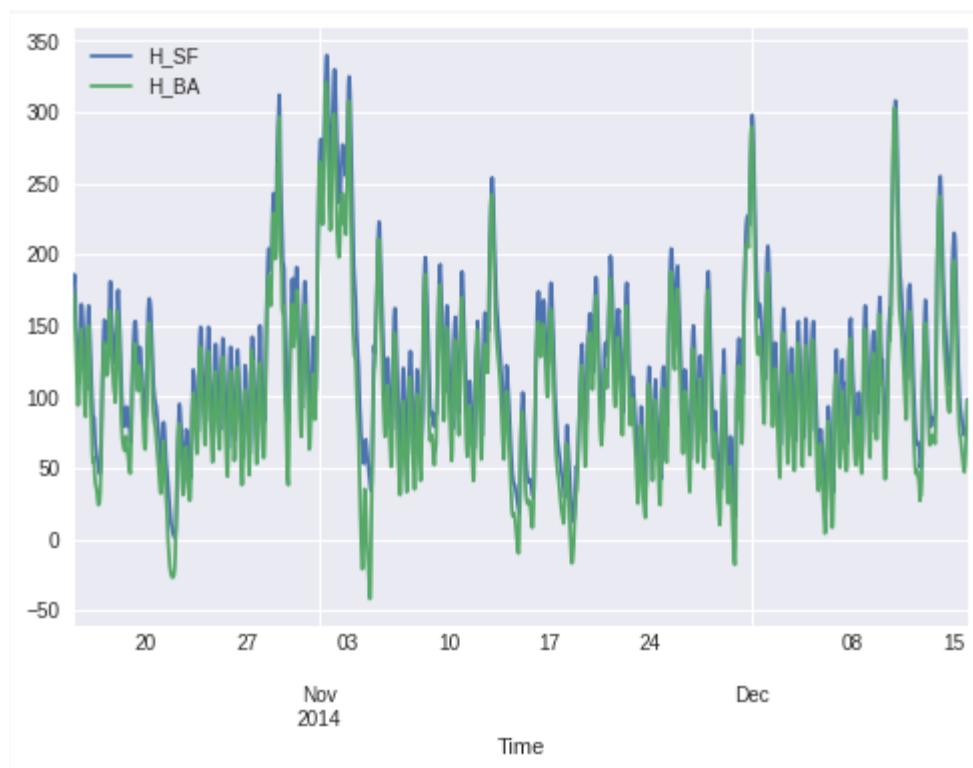
frecuencias predominantes en la serie de alturas. Las mareas se verán claramente porque estos efectos astronómicos son regulares y tienen frecuencias invariantes.

Vientos y ondas de tormenta en el Río de la Plata

Si miramos un gráfico un poco más extendido en el tiempo vamos a ver que las alturas no solo fluctúan con las mareas semidiurnas sino que la componente meteorológica (vientos principalmente, que generan *ondas de tormenta*) modifica las alturas de manera muy considerable.

El siguiente comando genera un gráfico entre el 15 de octubre de 2014 y el 15 de diciembre del mismo año.

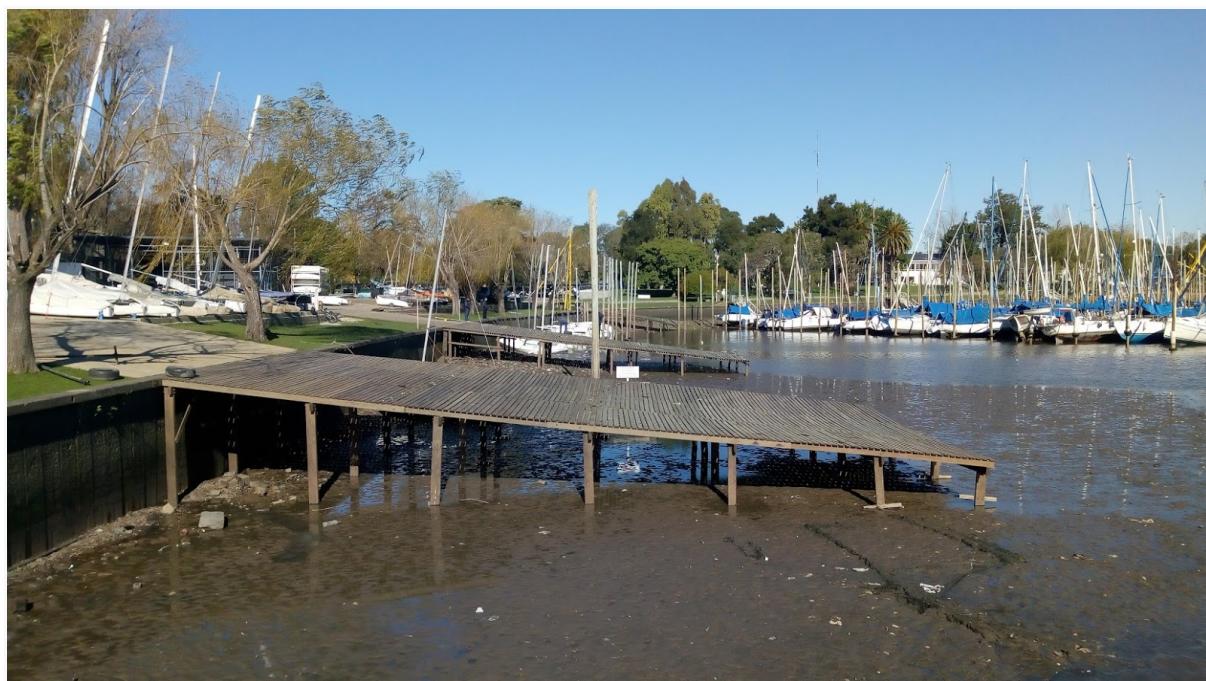
```
df['10-15-2014':'12-15-2014'].plot()
```



En ese gráfico se puede observar cómo una sudestada a principios de noviembre elevó el nivel del estuario más de un metro durante casi tres días. Las dos fotos que siguen son justamente de esa sudestada. Fueron tomadas el primero de noviembre por Gustavo Castaing.



Así como el viento del sudeste empuja el agua del mar hacia adentro del estuario y genera crecidas, los vientos del norte o el oeste también impulsan desplazamientos del agua del estuario, en este caso generando bajantes. En las siguientes dos fotos puede verse una bajante capturada por Juan Pablo Martínez Bigozzi el 19 de junio del 2019.



La transformada de Fourier no resultará muy útil para ver estas *ondas de tormenta*. Como carecen de regularidad, no aparecerán claramente en el espectro de frecuencias.

Ejercicio 7.10:

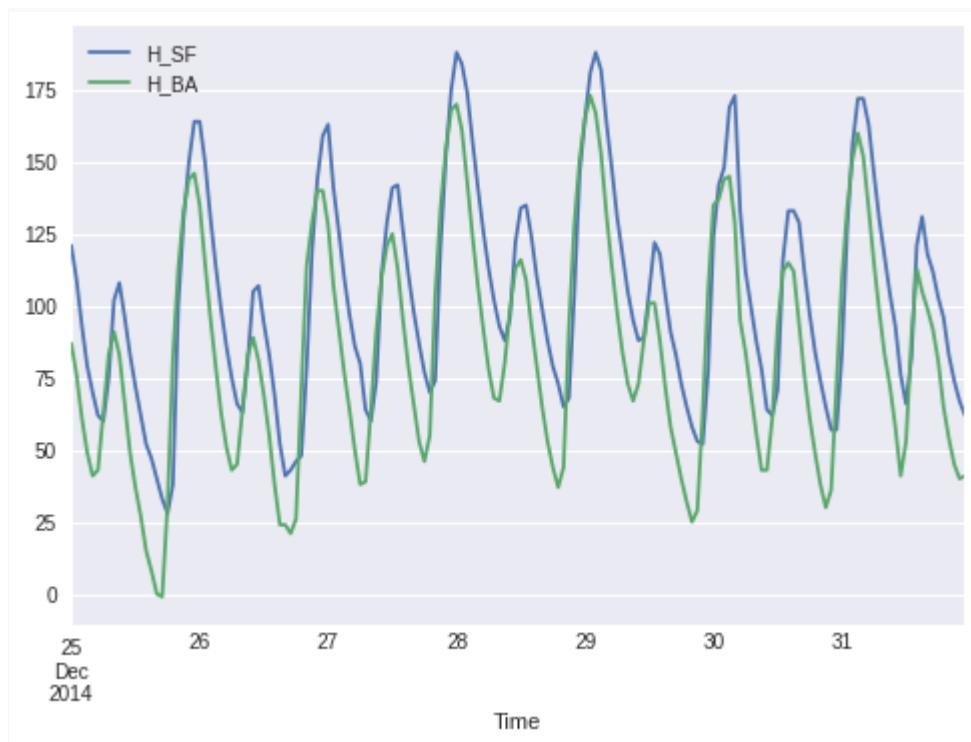
Trabajemos con una copia de este fragmento:

```
dh = df['12-25-2014':].copy()
```

Podemos desplazar (shift en inglés) una Serie de Pandas usando el método `ds.shift(pasos)`. Además, podemos subir o bajar su gráfico sumando una constante a todas las mediciones `ds + cte`.

Finalmente podemos unir dos series en un DataFrame de manera muy simple, para poder graficarlas juntas. Si concatenamos estas operaciones obtenemos algo así:

```
delta_t = 0 # tiempo que tarda la marea entre ambos puertos  
delta_h = 0 # diferencia de los ceros de escala entre ambos puertos  
pd.DataFrame([dh['H_SF'].shift(delta_t) - delta_h, dh['H_BA']]).T.plot()
```



Buscá los valores de `delta_t` (es un número entero, son pasos) y `delta_h` (puede tener decimales, es un float) que hacen que los dos gráficos se vean lo más similares posible.

Guardá tu código en el archivo `mareas_a_mano.py` para entregar.

Parte optativa

En lo que sigue vamos a usar herramientas matemáticas para hacer un análisis similar al que hicimos recién de manera *artesanal*. Para una onda sinusoidal, el desplazamiento horizontal corresponde a una diferencia de fase y el desplazamiento vertical es simplemente una constante aditiva. Vamos a descomponer la serie de alturas observadas del agua por medio de la transformada de Fourier.

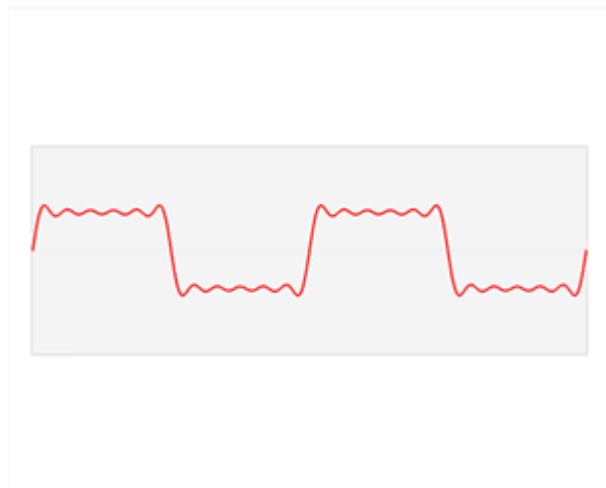
Las ondas de marea se mueven lentamente y tardan cierto tiempo en llegar de un puerto a otro. En lo que sigue vamos a ver cómo calcular el tiempo que le toma a esta onda en desplazarse de Buenos Aires hasta San Fernando.

Lo que sigue es optativo.

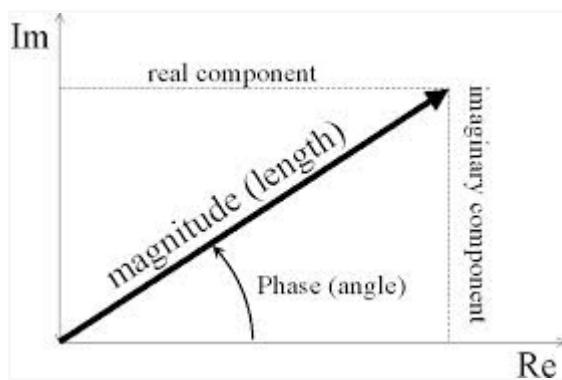
Análisis por medio de transformadas de Fourier

La transformada de Fourier descompone una señal en una suma de senos y cosenos (sinusoides) con diferentes frecuencias y amplitudes.

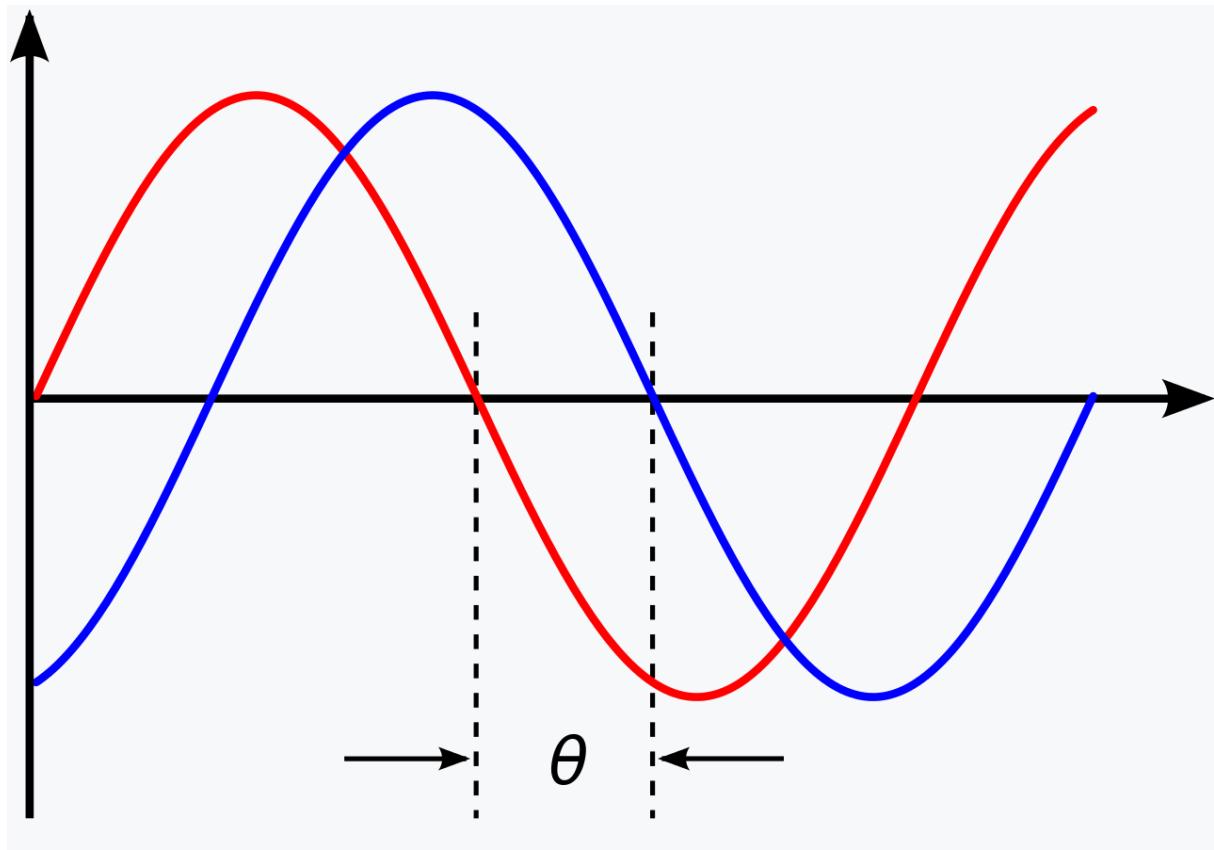
Esta animación ilustra gráficamente el proceso de la transformada de Fourier.



La transformada da, para cada frecuencia, un número complejo $a + bi$ que puede pensarse como un vector (a, b) en el plano. La parte real va a multiplicar un coseno con la frecuencia indicada y la parte imaginaria un seno con la misma frecuencia. La magnitud (o amplitud, o potencia) de la señal en esa frecuencia se corresponde con el largo del vector (a, b) .



La fase (o desplazamiento del máximo respecto del origen de las coordenadas), se corresponde con ángulo que forma este vector (a, b) con el semieje de los reales positivos.



Aquí, la variable theta (θ) representa el desplazamiento de fase de la curva azul (respecto a la roja que tiene desplazamiento nulo). Esta fase suele medirse en radianes, correspondiendo 2π a un ciclo completo de desfasaje.

Vamos a aplicar estas herramientas al análisis de la propagación de la onda de marea por el estuario del plata.

Preparación de módulos y datos

Vamos a usar los siguientes módulos:

```
from scipy import signal # para procesar señales  
import numpy as np  
import matplotlib.pyplot as plt
```

Seleccionemos las dos series como vectores de numpy (con el método `to_numpy()`).

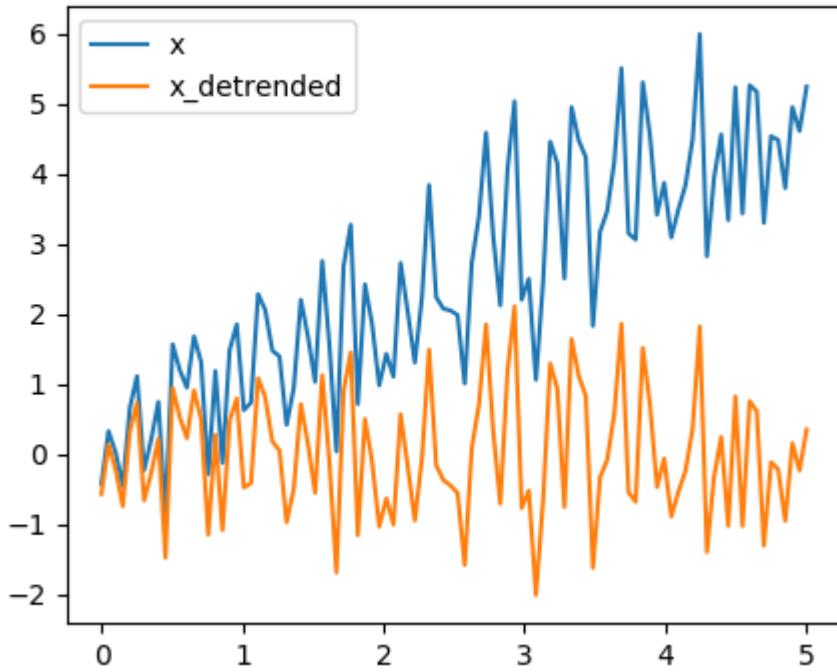
```
inicio = '2014-01'  
fin = '2014-06'
```

```
alturas_sf = df[inicio:fin]['H_SF'].to_numpy()
alturas_ba = df[inicio:fin]['H_BA'].to_numpy()
```

Primero definamos una función que calcule la transformada de Fourier para datos horarios y considerando como unidad de frecuencia los días (descartamos la mitad de los coeficientes de la transformada porque los datos son reales y no complejos). Podés tomarla como una caja negra por ahora...

```
def calcular_fft(y, freq_sampleo = 24.0):
    '''y debe ser un vector con números reales
    representando datos de una serie temporal.
    freq_sampleo está seteado para considerar 24 datos por unidad.
    Devuelve dos vectores, uno de frecuencias
    y otro con la transformada propiamente.
    La transformada contiene los valores complejos
    que se corresponden con respectivas frecuencias.'''
    N = len(y)
    freq = np.fft.freq(N, d = 1/freq_sampleo)[:N//2]
    tran = (np.fft.fft(y)/N)[:N//2]
    return freq, tran
```

Para poder analizar una onda por medio de su transformada de Fourier, es necesario que la onda sea periódica. Puede pasar que no sea el caso y que una onda tenga tendencia lineal, en ese caso podríamos usar la función `scipy.signal.detrend()`.



En nuestro caso supondremos que la marea media se mantuvo estable a lo largo del período de estudio, así que no tenemos que hacerle este procesamiento intermedio.

Espectro de potencia y de ángulos para San Fernando

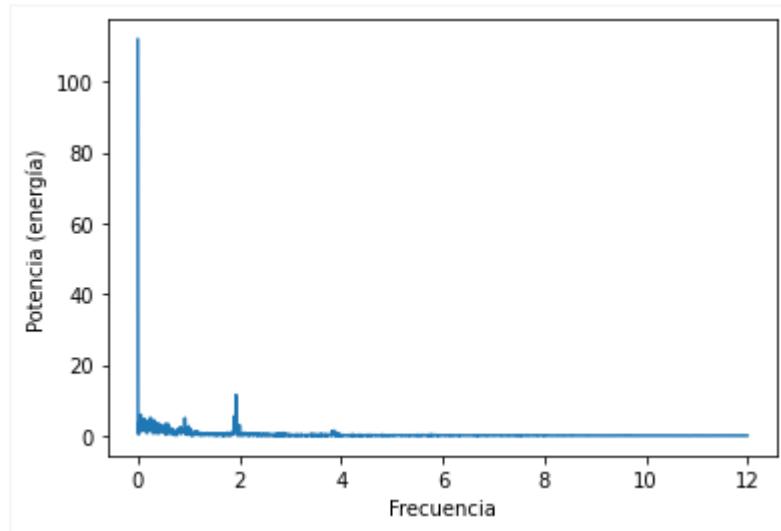
Primero calculamos la transformada de las alturas de San Fernando.

```
freq_sf, fft_sf = calcular_fft(alturas_sf)
```

Si quisiéramos graficar `freq_sf` contra `fft_sf` no podríamos ver mucho ya que `fft_sf` contiene números complejos.

La potencia (o amplitud) para cada frecuencia se calcula como el módulo del número complejo correspondiente (para la frecuencia `freq_sf[i]` y la potencia es `abs(fft_sf[i])`). Al graficar esto podemos ver la amplitud de los sinusoides para cada frecuencia. Este gráfico se llama el *espectro de potencias* de la onda original.

```
plt.plot(freq_sf, np.abs(fft_sf))
plt.xlabel("Frecuencia")
plt.ylabel("Potencia (energía)")
plt.show()
```



A simple vista se observan dos picos, uno en frecuencia 0 (constante relacionada con el cero de escala) y otro pico cercano a la frecuencia 2 (frecuencia semidiurna) que está relacionado con la onda de mareas.

El pico en la primera posición efectivamente se corresponde con la frecuencia 0 y su amplitud es:

```
>>> freq_sf[0]
0.0
>>> np.abs(fft_sf[0])
111.83
```

A partir de esto podemos decir que las alturas del río en San Fernando durante este período oscilan alrededor de los 111.8 cm de altura.

Para analizar precisamente el pico semidiurno podemos usar `find_peaks` que provee `scipy.signal` para evitar hacerlo a ojo. Vamos a pedir aquellos picos que tengan al menos cierta diferencia con su entorno (prominencia), un buen valor para esto es el 8. Podés probar otros valores y observar el resultado.

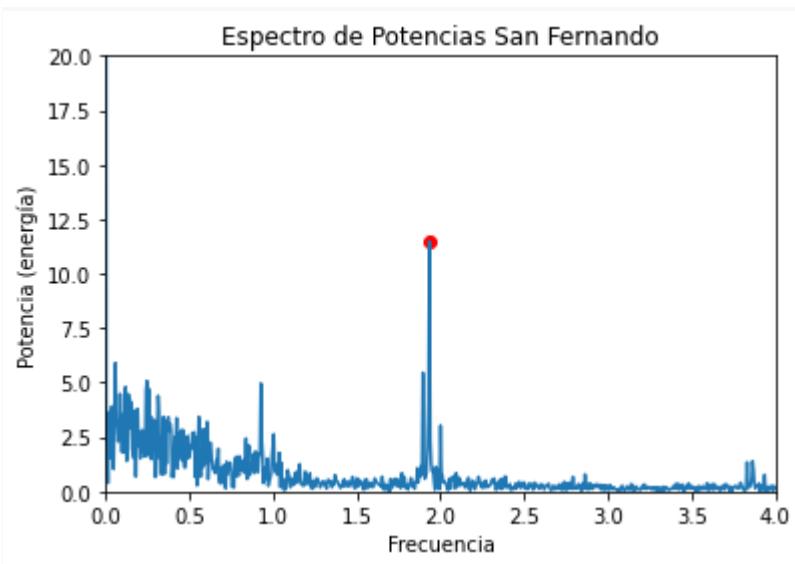
```
>>> print(signal.find_peaks(np.abs(fft_sf), prominence = 8))
(array([350]), {'prominences': array([11.4554514]), 'left_bases':
array([307]), 'right_bases': array([2109])})
```

Esta respuesta nos indica que hay un pico con la prominencia solicitada (al menos 8), que tiene un magnitud de 11.45 y que corresponde a la posición 350 del vector.

```
>>> freq_sf[350]
1.93
```

La frecuencia relacionada con esa posición es cercana a dos, como ya habíamos observado en el gráfico (dos ciclos por día). Podemos distinguir los picos agregando un punto rojo y mirando más de cerca el área de interés:

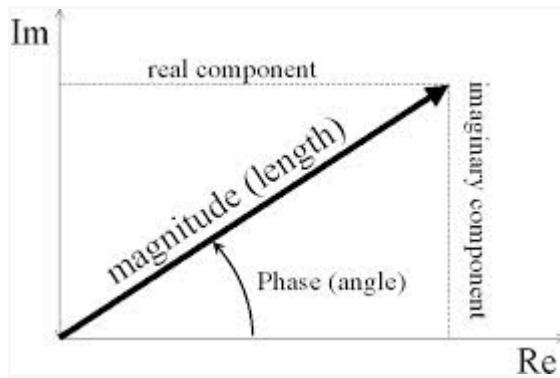
```
plt.plot(freq_sf, np.abs(fft_sf))
plt.xlabel("Frecuencia")
plt.ylabel("Potencia (energía)")
plt.xlim(0,4)
plt.ylim(0,20)
# me quedo solo con el último pico
pico_sf = signal.find_peaks(np.abs(fft_sf), prominence = 8)[0][-1]
# es el pico a analizar, el de la onda de mareas
# marco ese pico con un circulito rojo
plt.scatter(freq_sf[pico_sf], np.abs(fft_sf)[pico_sf], facecolor = 'r')
plt.show()
```



Estos gráficos permiten interpretar que si descomponemos la curva de alturas en San Fernando como suma de sinusoidales, el sinusoide con frecuencia 1.93 tiene una magnitud considerablemente llamativa. No es casualidad que este sea un punto distinguido: se trata de la frecuencia de las mareas lunares.

Ahora viene la parte un poco más sutil: el análisis de las fases. Si conocemos la fase de estas componentes en dos puertos distintos, podremos estimar el tiempo que tarda en desplazarse la marea de uno a otro.

Para calcular la fase (entre $-\pi$ y π) de dicha componente (la que ubicamos en la posición 350) en el puerto de San Fernando, podemos simplemente usar `np.angle()` y pasarle el número complejo en cuestión:



```
>>> ang_sf = np.angle(fft_sf)[pico_sf]
>>> print(ang_sf)
1.4849
```

Obtenemos un valor cercano a $\pi/2$. Recordemos que 2π corresponde a un desfasaje de un ciclo completo de la curva. Como nuestra curva de estudio tiene una frecuencia diaria ligeramente inferior a 2 ($\text{freq_sf}[350] \sim 1.93$), 2π corresponde a $24/1.93$ horas ~ 12.44 horas. Por lo tanto la fase obtenida con $\text{angSF}[350]$ corresponde a un retardo de

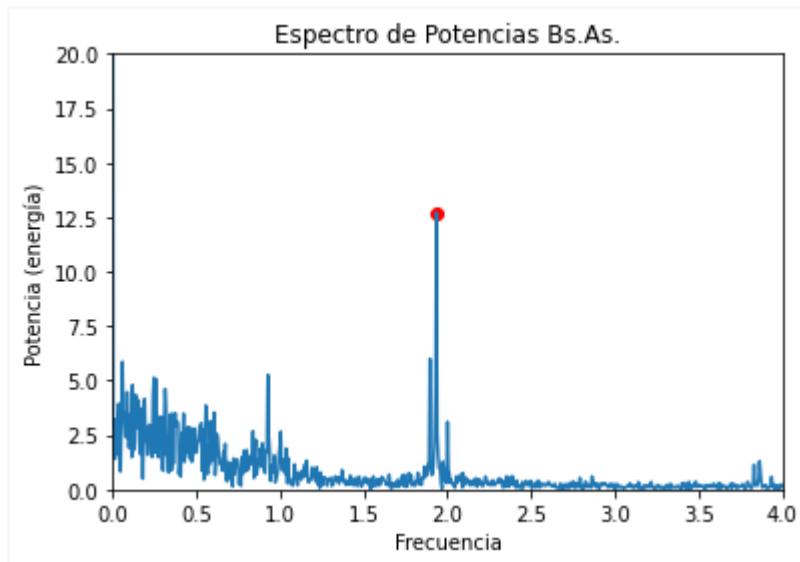
```
>>> ang_sf * 24 / (2 * np.pi * freq_sf[350])
2.93
```

Es decir, este sinusoide está desfasado poco menos de 3hs respecto al seno *neutro*.

Espectro de potencia y de ángulos para Buenos Aires

Repitamos velozmente el procedimiento para el puerto de Buenos Aires y analicemos las diferencias.

```
freq_ba, fft_ba = calcular_fft(alturas_ba)
plt.plot(freq_ba, np.abs(fft_ba))
plt.xlabel("Frecuencia")
plt.ylabel("Potencia (energía)")
plt.xlim(0, 4)
plt.ylim(0, 20)
# me quedo solo con el último pico
pico_ba = signal.find_peaks(np.abs(fft_ba), prominence = 8)[0][-1]
#se grafican los picos como circulitos rojos
plt.scatter(freq_ba[pico_ba], np.abs(fft_ba)[pico_ba], facecolor='r')
plt.title("Espectro de Potencias Bs.As.")
plt.show()
```



Si buscamos la constante alrededor de la que oscilan las mareas según el nivel del puerto de Buenos Aires obtenemos:

```
>>> np.abs(fft_ba[0])
88.21
```

Con este resultado es sencillo obtener una estimación para la diferencia de alturas de los ceros de escala entre ambos puertos.

Pregunta 1: ¿Cuál es la diferencia de altura media entre los puertos obtenida de esta forma?

Pregunta 2: ¿De qué otra forma se puede estimar el valor medio de un puerto? ¿Cuánto da la diferencia con este otro método?

Por otra parte, si observamos que el espectro de potencia vemos que los picos en ambos puertos son súmamente similares.

```
>>> print(signal.find_peaks(np.abs(fft_ba), prominence=8))
(array([350]), {'prominences': array([12.67228046]), 'left_bases':
array([279]), 'right_bases': array([1000])})
```

Las mareas de Buenos Aires tiene una componente de máxima amplitud en la frecuencia 1.93 (misma que San Fernando) y con una magnitud de 12.67 (bastante similar a la magnitud correspondiente en San Fernando). Resta estudiar la fase de la curva de los datos de `df_ba` en esta frecuencia para poder determinar con precisión la diferencia de fase entre ambos puertos para ondas de marea. Primero calculamos el ángulo de la componente correspondiente y luego lo convertimos en horas usando el factor `ang2h`:

```
>>> ang_ba = np.angle(fft_ba)[pico_ba]
>>> print(ang_ba)
1.96
>>> freq = freq_ba[pico_ba]
>>> ang2h = 24 / (2*np.pi*freq)
>>> ang_ba * ang2h
3.8786004708135566
```

Por lo tanto, el retardo de la onda de mareas puede calcularse usando

```
(ang_ba - ang_sf) * ang2h
```

Ejercicio 7.11: Desfasajes

En la Pregunta 1 estimaste el desfasaje vertical entre los ceros de escala de los puertos analizados. Ahora tenés que estimar el desfasaje temporal de las ondas de marea entre ambos puertos. ¿A cuántos minutos corresponde aproximadamente el tiempo que tarda la onda de mareas en llegar del puerto de Buenos Aires al de San Fernando?

Usá estos datos para volver a hacer el gráfico del [Ejercicio 7.10](#) (vas a tener que redondear a horas enteras el delay temporal).

Un poco más avanzados:

Ejercicio 7.12: Otros puertos

Usando el [archivo con datos del Puerto de Zárate](#), estimá el tiempo (expresado en horas y minutos) que le toma a la onda de marea llegar de Buenos Aires a Zárate.

Obviamente la onda llega atenuada a Zárate. ¿Cómo se refleja esta atenuación en la transformada? ¿Podés cuantificar esta atenuación?

Guardá lo que hayas hecho hasta acá en el archivo `mareas_fft.py` para entregar.

Ejercicio 7.13: Otros períodos

El primer análisis se realizó con el primer semestre del 2014 ya que no tiene ni datos faltantes ni outliers. Este ejercicio es una invitación a explorar estos problemas tan frecuentes.

- ¿Se puede comparar Zárate con San Fernando usando todos los datos de Zárate? ¿Cómo se comporta San Fernando en esas fechas?

- ¿Se pueden usar las series completas de BA y SF para calcular el desfasaje de la onda de mareas? ¿Qué son las alturas negativas? ¿Tienen sentido?

La siguiente función completa datos faltantes y corrige pequeños problemas en los índices. Es un poco brutal tratar así un DataFrame: es conveniente mirar los datos antes de completar faltantes. Lo dejamos como puntero a diferentes métodos muy útiles para la limpieza de series.

```
def reparar(df):
    df = df.interpolate()
    df = df.resample('H').mean()
    df = df.fillna(method = 'ffill')
    return df
```

7.6 Cierre de la séptima

Para cerrar esta clase te pedimos dos cosas:

- Que recopiles las soluciones de los siguientes ejercicios:
 - i. El archivo `vida.py` del [Ejercicio 7.1](#).
 - ii. El archivo `listar_imgs.py` del [Ejercicio 7.5](#).
 - iii. El archivo `arbolado_parques_veredas.py` del [Ejercicio 7.9](#).
 - iv. El archivo `mareas_a_mano.py` del [Ejercicio 7.10](#).
- Te proponemos además dos optativos:
 - i. El archivo `ordenar_imgs.py` del [Ejercicio 7.6](#).
 - ii. El archivo `mareas_fft.py` del [Ejercicio 7.12](#).
- Que completes [este formulario](#) usando tu dirección de mail como identificación. Al terminar vas a obtener un link para enviarnos tus ejercicios.

La corrección de pares de esta semana será con el archivo `arbolado_parques_veredas.py` del [Ejercicio 7.9](#).

¡Gracias!

8. Clases y objetos

En esta clase vamos a meternos con la programación orientada a objetos. Vamos a ver los conceptos de clases y objetos. Hasta ahora los programas que escribimos

usaron sólo tipos de datos nativos de Python, con la instrucción `class` vamos a definir nuevas clases. Vamos a ir más allá y a hablar del concepto de herencia, una herramienta comúnmente usada para escribir programas extensibles. Por último, vamos a referirnos a algunos *métodos especiales* de los objetos de Python.

En los ejercicios vamos a ver los conceptos de pilas y colas. En particular, la *pila de llamadas* nos prepara el camino para poder estudiar en un par de clases el concepto de recursión.

Cerramos esta clase con un ejercicio optativo que involucra conceptos de teledetección y de aprendizaje automático.

El ejercicio para la revisión de pares de esta semana es de programación orientada a objetos (modelar una torre de control).

- [8.1 Clases](#)
- [8.2 Herencia](#)
- [8.3 Métodos especiales](#)
- [8.4 Objetos, pilas y colas](#)
- [8.5 Teledetección](#)
- [8.6 Cierre de la octava clase](#)

8.1 Clases

La programación orientada a objetos requiere un pequeño pero importante cambio en la forma de pensar la programación tradicional. Dejá decantar los conceptos nuevos mientras leés esta sección.

En esta sección veremos el concepto de clase, cómo crear nuevos tipos de objetos, su utilidad, y las ventajas de esa forma de organizar los programas.

Programación orientada a objetos (POO)

La programación orientada a objetos es una forma de organizar el código. Así como un algoritmo suele estar asociado a una estructura de datos particular, la programación orientada a objetos "empaqueta" los datos junto con los métodos usados para tratarlos.

Cada uno de esos *objetos* consiste en

- Datos (atributos de los objetos).

- Comportamiento (métodos de los objetos: son funciones que actúan sobre los atributos del objeto).

Ya usaste objetos durante el curso infinidad de veces. Por ejemplo, al manipular una lista.

```
>>> nums = [1, 2, 3]
>>> nums.append(4)      # Esto es un método de la lista
>>> nums.insert(1,10)   # Otro método
>>> nums
[1, 10, 2, 3, 4]       # Estos son los datos modificados por los métodos
>>>
```

Miremos un poco más en detalle este fragmento de código. Sabemos que `nums` es una variable de tipo lista. Equivalentemente, podemos decir que `nums` es una *instancia* de la clase *list*. Cada variable de tipo lista es una instancia de la misma clase. Al hablar de 'instancia' nos referimos a un 'objeto': un objeto es una instancia de una clase.

Un objeto de tipo lista tiene atributos (datos) y métodos. Los métodos, como `append()` o `insert()`, se definen cuando se define la clase, pero se usan para manipular los datos de un objeto concreto (`nums` en este caso).

La instrucción `class`

Para definir un tipo nuevo de objeto, usá la instrucción `class`.

```
class Jugador:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.salud = 100

    def mover(self, dx, dy):
        self.x += dx
        self.y += dy

    def lastimar(self, pts):
        self.salud -= pts
```

Un objeto de tipo `Jugador` tiene como atributos `x`, `y` y `salud`. Sus métodos son `mover()` y `lastimar()`.

Puede decirse que una clase es la definición formal de las relaciones entre los datos y los métodos que los manipulan. Un objeto es una instancia particular de la clase a la cual pertenece, con datos propios pero los mismos métodos que los demás

objetos de esa clase. Este concepto te va a quedar más claro cuando lo veas funcionar y lo uses.

Instancias

Los programas manipulan instancias individuales de las clases. Cada instancia es un objeto, y es en cada objeto que uno puede manipular los datos y llamar a sus métodos.

Podés crear un objeto mediante un llamado a la clase como si fuera una función.

```
>>> a = Jugador(2, 3)      # Clase Jugador definida antes
>>> b = Jugador(10, 20)
>>>
```

a y b son instancias de Jugador definida más arriba. Es decir, a y b son objetos de la clase Jugador.

Importante: La instrucción `class` es solamente la definición de una clase, no hace nada por sí misma. Es similar a la definición de una función.

Datos de una instancia

Cada instancia tiene sus propios datos locales. Acá pedimos ver el atributo `x` de cada instancia:

```
>>> a.x
2
>>> b.x
10
```

Estos datos locales se inicializan, para cada instancia, durante la ejecución del método `__init__()` de la clase.

```
class Jugador:
    def __init__(self, x, y):
        # Todo dato guardado en `self` es propio de esa instancia
        self.x = x
        self.y = y
        self.salud = 100
```

No hay restricciones en la cantidad o el tipo de atributos que puede tener una clase.

Métodos de una instancia.

Los métodos de una instancia son los métodos y las funciones que actúan sobre los datos almacenados en esa instancia.

```
class Jugador:  
    ...  
    # `mover` es un método  
    def mover(self, dx, dy):  
        self.x += dx  
        self.y += dy
```

Siempre se recibe la instancia misma como primer argumento: "self" significa "mismo" como en "mi mismo" ó "en sí misma". Es como decir "yo".

```
>>> a.mover(1, 2)  
  
# `self` refiere a `a`  
# `dx` refiere a `1`  
# `dy` refiere a `2`  
def mover(self, dx, dy):  
    ...
```

Por convención siempre llamamos `self` a la instancia actual, y ésta es siempre pasada como primer argumento a todos los métodos. En realidad el nombre real de la variable no importa, pero es una convención en Python llamar al primer argumento `self`.

Podríamos usar `mismo`, por ejemplo, en lugar de `self` y todo va a funcionar igual, pero no respeta las convenciones de la comunidad:

```
class Jugador:  
    ...  
    # `mover` es un método  
    def mover(mismo, dx, dy):  
        mismo.x += dx  
        mismo.y += dy
```

Visibilidad en clases (Scoping)

Las clases no definen ni limitan (como los módulos) un entorno de visibilidad.

```
class Jugador:  
    ...  
    def mover(self, dx, dy):  
        self.x += dx  
        self.y += dy  
  
    def izquierda(self, dist):  
        mover(-dist, 0)      # NO! Refiere a una función global `mover`.
```

```
self.mover(-dist, 0) # Si. Llama al método `mover` definido antes.
```

Si necesitás referirte a un dato o un método de una clase tenés que hacer una referencia explícita (agregando el `self`), sino te estás refiriendo a otra cosa como en el ejemplo anterior.

Ejercicios

Vamos a comenzar esta serie de ejercicios modificando código que escribiste antes del parcial. En particular retomaremos el código del [Ejercicio 6.5](#). Te dejamos [acá](#) una versión funcionando que podés mirar y/o usar. Tiene cosas interesantes, aunque tengas la tuya funcionando si querés pegale una mirada.

Ejercicio 8.1: Objetos como estructura de datos.

Durante las primeras clases trabajamos con datos en forma de tuplas y diccionarios. Un lote con cajones de frutas, por ejemplo, estaba representado por una tupla, como ésta:

```
s = ('Pera', 100, 490.10)
```

o por un diccionario, de esta otra forma:

```
s = { 'nombre' : 'Pera',
      'cajones' : 100,
      'precio' : 490.10
    }
```

Incluso escribiste funciones para manipular datos almacenados de ese modo:

```
def costo(registro):
    return registro['cajones'] * registro['precio']
```

Otra forma de representar los datos con los que estás trabajando es definir una clase. Creá un archivo llamado `lote.py` y adentro definí una clase llamada `Lote` que represente un lote de cajones de una misma fruta. Definila de modo que cada instancia de la clase `Lote` (es decir, cada objeto `lote`) tenga los atributos `nombre`, `cajones`, y `precio`. Éste es un ejemplo del comportamiento buscado:

```
>>> import lote
>>> a = lote.Lote('Pera', 100, 490.10)
>>> a.nombre
'Pera'
>>> a.cajones
100
>>> a.precio
```

```

490.1
>>>
Vamos a crear más objetos de tipo Lote para manipularlos. Por ejemplo:
>>> b = lote.Lote('Manzana', 50, 122.34)
>>> c = lote.Lote('Naranja', 75, 91.75)
>>> b.cajones * b.precio
6117.0
>>> c.cajones * c.precio
6881.25
>>> lotes = [a, b, c]
>>> lotes
[<lote.Lote object at 0x37d0b0>, <lote.Lote object at 0x37d110>, <lote.Lote
object at 0x37d050>]
>>> for c in lotes:
    print(f'{c.nombre:>10s} {c.cajones:>10d} {c.precio:>10.2f}' )

... mirá el resultado ...
>>>

```

Fijate que la clase `Lote` funciona como una "fábrica" para crear objetos que son instancias de esa clase. Vos la llamás como si fuera una función y te crea una nueva instancia de sí misma. Más aún, cada instancia es única y tiene sus propios datos que son independientes de las demás instancias de la misma clase.

Una instancia definida por una clase puede tener cierta similitud con un diccionario, pero usa una sintaxis algo diferente. Por ejemplo, en lugar de escribir `c['nombre']` ó `c['precio']` en objetos escribís `c.nombre` ó `c.precio`.

Ejercicio 8.2: Agregá algunos métodos

Al definir una clase podés agregar funciones a los objetos que definís. Las funciones específicas de objetos se llaman *métodos* y operan sobre los datos guardados en cada instancia.

Agregá los métodos `costo()` y `vender()` a tu objeto `Lote`. Deberían dar este comportamiento:

```

>>> import lote
>>> s = lote.Lote('Pera', 100, 490.10)
>>> s.costo()
49010.0
>>> s.cajones
100
>>> s.vender(25)
>>> s.cajones
75
>>> s.costo()

```

```
36757.5
>>>
```

Ejercicio 8.3: Lista de instancias

Seguí estos pasos para crear una lista de las instancias de `Lote` (una lista de objetos `Lote`) a partir de una lista de diccionarios. Luego calculá el precio total de todas esas instancias.

```
>>> import fileparse
>>> with open('Data/camion.csv') as lineas:
...     camion_dicts = fileparse.parse_csv(lineas, select = ['nombre',
'cajones', 'precio'], types = [str, int, float])
...
>>> camion = [ lote.Lote(d['nombre'], d['cajones'], d['precio']) for d in
camion_dicts]
>>> camion
[<lote.Lote object at 0x10c9e2128>, <lote.Lote object at 0x10c9e2048>,
<lote.Lote object at 0x10c9e2080>,
 <lote.Lote object at 0x10c9e25f8>, <lote.Lote object at 0x10c9e2630>,
<lote.Lote object at 0x10ca6f748>,
 <lote.Lote object at 0x10ca6f7b8>]
>>> sum([c.costo() for c in camion])
47671.15
>>>
```

Ejercicio 8.4: Usá tu clase

Modificá la función `leer_camion()` en el programa `informe.py` de modo que lea un archivo con el contenido de un camion y devuelva una lista de instancias de `Lote` como mostramos recién en el [Ejercicio 8.3](#).

Cuando hayas hecho esto, cambiá un poco el código en `informe.py` y en `costo_camion.py` de modo que funcionen con objetos `Lote` (instancias de la clase `Lote`) en lugar de diccionarios.

Ayuda: No deberían ser cambios importantes. Las referencias a diccionarios ahora tienen que hacer referencia a objetos (`c['cajones']` cambia a `c.cajones`).

Hecho esto, deberías poder ejecutar tus funciones como antes:

```
>>> import costo_camion
>>> costo_camion.costo_camion('Data/camion.csv')
47671.15
>>> import informe
>>> informe.informe_camion('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

8.2 Herencia

La herencia entre clases es una herramienta muy usada para escribir programas extensibles. Exploraremos esta idea a continuación.

Introducción

Se usa herencia para crear objetos más especializados a partir de objetos existentes.

```
class Padre:
    ...
class Hijo(Padre):
    ...
```

Se dice que `Hijo` es una clase derivada o subclase. La clase `Padre` es conocida como la clase base, o superclase. La expresión `class Hijo(Padre) :` significa que estamos creando una clase llamada `Hijo` que es derivada de la clase `Padre`.

Extensiones

Al usar herencia podés tomar una clase existente y ...

- Agregarle métodos
- Redefinir métodos existentes
- Agregar nuevos atributos

Podés verlo como una forma de extender de tu código existente. Darle nuevos comportamientos, abarcar un abanico más amplio de posibilidades ó aumentar su compatibilidad.

Ejemplo

Suponé que partís de la siguiente clase:

```
class Lote:  
    def __init__(self, nombre, cajones, precio):  
        self.nombre = nombre  
        self.cajones = cajones  
        self.precio = precio  
  
    def costo(self):  
        return self.cajones * self.precio  
  
    def vender(self, ncajones):  
        self.cajones -= ncajones
```

Podés modificar lo que necesites mediante herencia.

Agregar un método nuevo

```
class MiLote(Lote):  
    def rematar(self):  
        self.vender(self.cajones)
```

Se puede usar así:

```
>>> c = MiLote('Pera', 100, 490.1)  
>>> c.vender(25)  
>>> c.cajones  
75  
>>> c.rematar()  
>>> c.cajones  
0  
>>>
```

Esta clase heredó los atributos y métodos de `Lote` y la extendió con un nuevo método (`rematar()`).

Redefinir un método existente

```
class MiLote(Lote):  
    def costo(self):  
        return 1.25 * self.cajones * self.precio
```

Un ejemplo de uso:

```
>>> c = MiLote('Pera', 100, 490.1)  
>>> c.costo()  
61262.5  
>>>
```

El método nuevo simplemente reemplaza al definido en la clase base. Los demás métodos y atributos no son afectados. ¿No es buenísimo?

Utilizar un método prevalente

Hay veces en que una clase extiende el método de la superclase a la que pertenece, pero necesita ejecutar el método original como parte de la redefinición del método nuevo. Para referirte a la superclase, usá `super()`:

```
class Lote:
    ...
    def costo(self):
        return self.cajones * self.precio
    ...

class MiLote(Lote):
    def precio(self):
        # Fijate cómo usamos `super`
        costo_orig = super().costo()
        return 1.25 * costo_orig
```

Usá `super()` para llamar al método de la clase base (del la cual ésta es heredera).

El método `__init__` y herencia.

Al crear cada instancia se ejecuta `__init__`. Ahí reside el código importante para la creación de una instancia nueva. Si redefinís `__init__` siempre incluí un llamado al método `__init__` de la clase base para inicializarla también.

```
class Lote:
    def __init__(self, nombre, cajones, precio):
        self.nombre = nombre
        self.cajones = cajones
        self.precio = precio
    ...

class MiLote(Lote):
    def __init__(self, nombre, cajones, precio, factor):
        # Fijate como es el llamado a `super().__init__()`
        super().__init__(nombre, cajones, precio)
        self.factor = factor

    def costo(self):
        return self.factor * super().costo()
```

Es necesario llamar al método `__init__()` en la clase base. Es una forma de ejecutar la versión previa del método que estamos redefiniendo, como mostramos recién.

Usos de herencia

Uno de los usos de definir una clase como heredera de otra es organizar jerárquicamente objetos que están relacionados.

Un ejemplo: Las figuras geométricas pueden tener ciertos métodos y atributos que luego son refinados en casos concretos como círculos o rectángulos.

```
class FiguraGeom:  
    ...  
  
class Circulo(FiguraGeom):  
    ...  
  
class Rectangulo(FiguraGeom):  
    ...
```

Imaginate por ejemplo su uso en una jerarquía lógica, o taxonómica, en la que las clases tienen una relación natural tal que hace intuitivo derivar una de otra.

Una aplicación más común, y tal vez más práctica, consiste en escribir código que es reutilizable y/o extensible. Podríamos definir una clase base para una interfaz de transferencia de datos y permitir que cada fabricante de equipo de adquisición de datos implemente los detalles de comunicación con cada interfaz en particular.

```
class Procesador_de_datos(TCPServer):  
    def procesar_pedido(self):  
        ...  
        # Procesamiento de datos
```

La clase base contiene código de administración no específico. Cada clase hereda ese código y modifica las partes necesarias.

Relación "isinstance"

La herencia establece una relación de clases.

```
class FiguraGeom:  
    ...  
  
class Circulo(FiguraGeom):
```

Preguntamos si un objeto es una instancia de cierta clase:

```
>>> f = Circulo(4.0)
>>> isinstance(f, FiguraGeom)
True
>>>
```

Importante: Idealmente, todo código que funcione con instancias de una clase base debería también funcionar con instancias de las clases derivadas de ella.

La clase base `object`

Si una clase no tiene superclase, a veces se escribe `object` como clase base.

```
class Figura_geom(object):
    ...
object es la superclase de todo objeto en Python.
```

Herencia múltiple.

Podés heredar de varias clases simultáneamente si los especificás en la definición de clase.

```
class Madre:
    ...

class Padre:
    ...

class Hijo(Madre, Padre):
    ...
```

La clase `Hijo` hereda características de ambos padres. Algunos detalles son un poco delicados y no vamos a usar esa forma de heredar clases en este curso, aunque vas a encontrar un poco más de información en la próxima sección.

Ejercicios

El concepto de herencia es especialmente útil cuando estás escribiendo código que va a ser extendido o adaptado, ya sea en bibliotecas o grandes sistemas configurables, pero también en pequeños paquetes de procesamiento de datos que pueden adquirir datos de diversas fuentes. Como ya dijimos antes, uno puede

escribir las relaciones y comportamientos fundamentales y dejar los detalles de implementación de cada interfaz para cuando sean necesarios.

Para verlo mejor volvamos a la función `imprimir_informe()` del [Ejercicio 5.1](#), que figura en el programa `informe.py`. Tenía más o menos este aspecto:

```
def imprimir_informe(data_informe):
    """
    Imprime una tabla prolífica desde una lista de tuplas con (nombre,
    cajones, precio, cambio)
    """
    headers = ('Nombre', 'Cajones', 'Precio', 'Cambio')
    print('%10s %10s %10s %10s' % headers)
    print(len(headers) * '-' * 10 + ' ')
    for row in data_informe:
        print('%10s %10d %10.2f %10.2f' % row)
```

Al ejecutar tu programa `informe` la salida es algo parecido a esto:

```
>>> import informe
>>> informe.informe_camion('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
```

A continuación vamos a trabajar con herencias relacionadas con este código.

Ejercicio 8.5: Un problema de extensibilidad

Imaginá que necesitás que la función `imprimir_informe()` pueda exportar el informe en una variedad de formatos: texto plano, HTML, CSV ó XML. Podrías escribir una función enorme que resuelva todos los casos, pero resultaría en código repetido, y difícil de mantener. Esta es una oportunidad perfecta para usar herencia de objetos.

Vamos a enfocarnos en los pasos necesarios para crear una tabla.

Al principio de la tabla tenemos los encabezados de las columnas. Después de eso, los datos de la tabla ordenados en una fila por ítem. Pongamos cada uno de esos pasos en una clase distinta. Creá un archivo llamado `formato_tabla.py` y definí la siguiente clase:

```
# formato_tabla.py

class FormatoTabla:
    def encabezado(self, headers):
        """
        Crea el encabezado de la tabla.
        """
        raise NotImplementedError()

    def fila(self, rowdata):
        """
        Crea una única fila de datos de la tabla.
        """
        raise NotImplementedError()
```

Por ahora la clase no hace nada, pero sirve como una especie de especificación de diseño para otras clases que vamos a definir. Una clase como ésta es a menudo llamada "clase base abstracta".

Ahora es necesario modificar la función `imprimir_informe()` para que acepte como fuente de datos un objeto `FormatoTabla` e invoque los métodos de este objeto para producir la tabla de salida. Algo así:

```
# informe.py
import formato_tabla

...

def imprimir_informe(data_informe, formateador):
    """
    Imprime una tabla prolíja desde una lista de tuplas
    con (nombre, cajones, precio, diferencia)
    """
    formateador.encabezado(['Nombre', 'Cantidad', 'Precio', 'Cambio'])
    for nombre, cajones, precio, cambio in data_informe:
        rowdata = [nombre, str(cajones), f'{precio:0.2f}', f'{cambio:0.2f}']
        formateador.fila(rowdata)
```

Como agregaste un argumento a `imprimir_informe()`, hay que modificar también `informe_camion()`. Cambialo para que cree un objeto `formateador` de este modo:

```
# informe.py

import formato_tabla

...

def informe_camion(archivo_camion, archivo_precios):
    """
```

```

Crea un informe a partir de un archivo de camión
y otro de precios de venta.

'''

# Leer archivos con datos
camion = leer_camion(archivo_camion)
precios = leer_precios(archivo_precios)

# Crear los datos para el informe
data_informe = hacer_informe(camion, precios)

# Imprimir el informe
formateador = formateo_tabla.FormatoTabla()
imprimir_informe(data_informe, formateador)

```

Ejecutá este código:

```

>>> ====== RESTART ======
>>> import informe
>>> informe.informe_camion('Data/camion.csv', 'Data/precios.csv')
... crashes ...

```

Debería dar inmediatamente una excepción de tipo `NotImplementedError`. No es nada maravilloso, pero es exactamente lo que esperábamos que sucediera, ¿no? Sigamos...

Ejercicio 8.6: Usemos herencia para cambiar la salida

La clase `FormatoTabla` que definiste en la primera parte es sólo la base de un sistema extensible. Éste es el momento de extenderla. Definí una clase `FormatoTablaTXT` como sigue:

```

# formato_tabla.py

...

class FormatoTablaTXT(FormatoTabla):
    '''
    Generar una tabla en formato TXT
    '''

    def encabezado(self, headers):
        for h in headers:
            print(f'{h:>10s}', end=' ')
        print()
        print(( '-' * 10 + ' ') * len(headers))

    def fila(self, data_fila):
        for d in data_fila:
            print(f'{d:>10s}', end=' ')

```

```
    print()
```

Modificá la función `informe_camion()` y probala:

```
# informe.py

...
def informe_camion(archivo_camion, archivo_precios):
    """
    Crea un informe por camion a partir de archivos camion y precio.
    """
    # Leer archivos con datos
    camion = leer_camion(archivo_camion)
    precios = leer_precios(archivo_precios)

    # Obtener los datos para un informe
    data_informe = hacer_informe(camion, precios)

    # Imprimir
    formateador = formato_tabla.FormatoTablaTXT()
    imprimir_informe(data_informe, formateador)
```

Este código debería dar la misma salida que antes:

```
>>> =====REINICIAR INTERPRETE=====
>>> import informe
>>> informe.informe_camion('Data/camion.csv', 'Data/precios.csv')
  Nombre      Cantidad      Precio      Cambio
  -----      -----
  Lima          100      32.20       8.02
  Naranja        50      91.10      15.18
  Caqui         150     103.44       2.02
  Mandarina      200      51.23      29.66
  Durazno        95      40.37      33.11
  Mandarina       50      65.10      15.79
  Naranja        100      70.44      35.84
>>>
```

Ahora probemos otras variantes. Definí, para empezar, una nueva clase llamada `FormatoTablaCSV` que genere la salida en formato CSV:

```
# formato_tabla.py
...
class FormatoTablaCSV(FormatoTabla):
    """
    Generar una tabla en formato CSV
    """
    def encabezado(self, headers):
```

```

    print(', '.join(headers))

def fila(self, data_fila):
    print(', '.join(data_fila))

```

Modificá tu programa `informe.py` de este modo:

```

def informe_camion(archivo_camion, archivo_precios):
    """
    Crea un informe por camion a partir de archivos camion y precio.
    """
    # Leer archivos con datos
    camion = leer_camion(archivo_camion)
    precios = leer_precios(archivo_precios)

    # Obtener los datos para un informe
    data_informe = hacer_informe(camion, precios)

    # Imprimir
    formateador = formato_tabla.FormatoTablaCSV()
    imprimir_informe(data_informe, formateador)

```

Ahora la salida debería tener este aspecto:

```

>>> =====REINICIAR INTERPRETE=====
>>> import informe
>>> informe.informe_camion('Data/camion.csv', 'Data/precios.csv')
Nombre,Cantidad,Precio,Cambio
Lima,100,32.20,8.02
Naranja,50,91.10,15.18
Caqui,150,103.44,2.02
Mandarina,200,51.23,29.66
Durazno,95,40.37,33.11
Mandarina,50,65.10,15.79
Naranja,100,70.44,35.84

```

Usando las mismas ideas creá la clase `FormatoTablaHTML` que produzca un tabla de la siguiente forma:

```

<tr><th>Nombre</th><th>Cajones</th><th>Precio</th><th>Cambio</th></tr>
<tr><td>Lima</td><td>100</td><td>32.20</td><td>8.02</td></tr>
<tr><td>Naranja</td><td>50</td><td>91.10</td><td>15.18</td></tr>
<tr><td>Caqui</td><td>150</td><td>103.44</td><td>2.02</td></tr>
<tr><td>Mandarina</td><td>200</td><td>51.23</td><td>29.66</td></tr>
<tr><td>Durazno</td><td>95</td><td>40.37</td><td>33.11</td></tr>
<tr><td>Mandarina</td><td>50</td><td>65.10</td><td>15.79</td></tr>
<tr><td>Naranja</td><td>100</td><td>70.44</td><td>35.84</td></tr>

```

Para testear tu código, modifícá el programa principal de modo que use un objeto de la clase `FormatoTablaHTML` en lugar de uno de la clase `FormatoTablaCSV` para darle formato a la tabla de salida. Fijate lo fácil que es cambiar el comportamiento de un programa cuando tenés objetos que son compatibles entre sí.

Ejercicio 8.7: Polimorfismo en acción

Una de las grandes ventajas de la programación orientada a objetos es que podés cambiar un objeto por otro compatible y tu programa va a funcionar sin necesidad de adaptar el código que usa esos objetos.

Si escribiste un programa diseñado para usar un objeto de la clase `FormatoTabla`, va a funcionar sin importar qué objeto de esa clase uses. A este comportamiento particular se lo llama polimorfismo. Está relacionado con la capacidad de usar la misma interfaz con diferentes objetos de la misma clase, haciendo que el programa como un todo se porte distinto.

Ahora bien, un potencial problema es cómo diseñar tu programa de manera que el usuario final pueda elegir el formato. Usar los nombres de las clases de formateadores no resultaría cómodo. Una solución posible es considerar un condicional:

```
def informe_camion(archivo_camion, archivo_precios, fmt = 'txt'):
    """
    Crea un informe con la carga de un camión
    a partir de archivos camion y precio.
    El formato predeterminado de la salida es txt
    Alternativas: csv o html
    """
    # Leer archivos con datos
    camion = leer_camion(archivo_camion)
    precios = leer_precios(archivo_precios)

    # Obtener los datos para un informe
    data_informe = hacer_informe(camion, precios)

    # Elige formato
    if fmt == 'txt':
        formateador = formato_tabla.FormatoTablaTXT()
    elif fmt == 'csv':
        formateador = formato_tabla.FormatoTablaCSV()
    elif fmt == 'html':
        formateador = formato_tabla.FormatoTablaHTML()
    else:
        raise RuntimeError(f'Unknown format {fmt}')
    imprimir_informe(informe, formateador)
```

En este código, el usuario especifica un nombre simplificado como `txt` o `csv` para elegir el formato. Pero bancá. ¿Es una buena idea poner un gran bloque `if` en la función `informe_camion()`? ¿O quizás sería mejor ponerla directamente en una función de propósito general en otro lado?

En el archivo `formato_tabla.py`, agregá la función `crear_formateador(nombre)` que permita crear un objeto formateador dado un tipo de salida como `txt`, `csv`, o `html`. Modificá `informe_camion()` para que se vea así:

```
def informe_camion(archivo_camion, archivo_precios, fmt = 'txt'):  
    """  
    Crea un informe con la carga de un camión  
    a partir de archivos camion y precio.  
    El formato predeterminado de la salida es .txt  
    Alternativas: .csv o .html  
    """  
  
    # Lee archivos de datos  
    camion = leer_camion(archivo_camion)  
    precios = leer_precios(archivo_precios)  
  
    # Crea la data del informe  
    data_informe = hacer_informe(camion, precios)  
  
    # Imprime el informe  
    formateador = formato_tabla.crear_formateador(fmt)  
    imprimir_informe(data_informe, formateador)
```

Acordate de testear todas las ramas posibles del código para asegurarte de que está funcionando. Llamalo y pedile crear salidas en todos los formatos (podés ver el HTML con tu browser).

Ejercicio 8.8: Volvamos a armar todo

Modificá tu programa `informe.py` de modo que la función `informe_camion()` acepte un parámetro opcional que especifique el formato de salida deseado. Por ejemplo:

```
>>> informe.informe_camion('Data/camion.csv', 'Data/precios.csv', fmt = 'txt')
```

Nombre	Cajones	Precio	Cambio
Lima	100	32.20	8.02
Naranja	50	91.10	15.18
Caqui	150	103.44	2.02
Mandarina	200	51.23	29.66
Durazno	95	40.37	33.11
Mandarina	50	65.10	15.79
Naranja	100	70.44	35.84

>>>

Modificá el programa principal y usá `sys.argv()` para poder definir un formato particular directamente desde la línea de comandos. En el siguiente ejemplo se ve un caso de uso. Idealmente, ese parámetro debería ser opcional y, si no se lo pasás, debería andar como antes.

```
bash $ python3 informe.py Data/camion.csv Data/precios.csv csv
Nombre,Cajones,Precio,Cambio
Lima,100,32.20,8.02
Naranja,50,91.10,15.18
Caqui,150,103.44,2.02
Mandarina,200,51.23,29.66
Durazno,95,40.37,33.11
Mandarina,50,65.10,15.79
Naranja,100,70.44,35.84
```

Esta versión de `informe.py` preparala para entregarla.

Discusión

El caso que vimos es un ejemplo de uno de los usos más comunes de herencia en programación orientada a objetos: escribir programas extensibles. Un sistema puede definir una interfaz mediante una superclase base, y pedirte que escribas tus propias implementaciones derivadas de esa clase. Si escribís los métodos específicos para tu caso particular podés adaptar la función de un sistema general para resolver tu problema.

Otro concepto, un poco más interesante, es el de crear tus propias abstracciones. En los ejercicios de esta parte definimos *nuestra propia clase* para crear variaciones en el formato de un informe. Tal vez estés pensando "¡Debería usar una biblioteca para crear formatos ya escrita por otre!". Bueno, no. Está bueno que puedas *tanto* crear tu propia clase *como* usar una biblioteca ya escrita. El hecho de usar tu propia clase te da flexibilidad.

Siempre que tu programa adhiera a la interfaz de objetos definida por tu clase, podés cambiar la implementación interna en los objetos que escribas para que funcionen del modo que elijas. Podés escribir todo el código vos o usar bibliotecas ya escritas, no importa. Cuando encuentres algo mejor, cambiás tu implementación para que llame al nuevo código. Si la interfaz que hiciste está bien escrita, no vas a necesitar modificar el programa que usa las diferentes implementaciones. Simplemente funcionan si cumplen los contratos de la interfaz. Es algo muy útil y es

uno de los motivos por los que usar herencia puede resolverte los problemas de extensibilidad y diversidad a futuro.

Dicho esto, es cierto que diseñar un programa en el paradigma orientado a objetos puede resultar algo muy difícil. Si vas a encarar proyectos grandes con esta herramienta, consultá libros sobre patrones de diseño en POO. De todos modos, haber entendido lo que acabamos de hacer te permite llegar bastante lejos.

8.3 Métodos especiales

Podemos modificar muchos comportamientos de Python definiendo lo que se conoce como "métodos especiales". Acá vamos a ver cómo usar estos métodos y a discutir brevemente otras herramientas relacionadas.

Introducción

Una clase puede tener definidos métodos especiales. Estos métodos tienen un significado particular para el intérprete de Python. Sus nombres empiezan y terminan en __ (doble guión bajo). Por ejemplo `__init__`.

```
class Lote(object):
    def __init__(self):
        ...
    def __repr__(self):
        ...
```

Hay decenas de métodos especiales pero sólo vamos a tratar algunos ejemplos específicos acá.

Métodos especiales para convertir a strings

Los objetos tienen dos representaciones de tipo cadena.

```
>>> from datetime import date
>>> d = date(2020, 12, 21)
>>> print(d)
2020-12-21
>>> d
datetime.date(2020, 12, 21)
>>>
```

La función `str()` se usa para crear una representación agradable de ver:

```
>>> str(d)
'2020-12-21'
>>>
```

Pero para crear una representación más informativa para programadores, se usa la función `repr()`.

```
>>> repr(d)
'datetime.date(2020, 12, 21)'
>>>
```

Las funciones `str()` y `repr()` llaman a métodos especiales de la clase para generar la cadena de caracteres que se va a mostrar.

```
class Date(object):
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    # Con `str()`
    def __str__(self):
        return f'{self.year}-{self.month}-{self.day}'

    # Con `repr()`
    def __repr__(self):
        return f'Date({self.year}, {self.month}, {self.day})'
```

Nota: Hay una convención para `__repr__()` que indica que debe devolver un string que, cuando sea pasado a `eval()` vuelva a crear el objeto subyacente. Analizá el ejemplo de `datetime.date(2020, 12, 21)`. Si no es posible crear un string que haga eso, la convención es generar una representación que sea fácil de interpretar para una persona.

```
class Punto():
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f'({self.x}, {self.y})'

    # Used with `repr()`
    def __repr__(self):
        return f'Punto({self.x}, {self.y})'
```

Métodos matemáticos especiales

Las operaciones matemáticas sobre los objetos involucran llamados a los siguientes métodos.

a + b	a. <u>__add__</u> (b)
a - b	a. <u>__sub__</u> (b)
a * b	a. <u>__mul__</u> (b)
a / b	a. <u>__truediv__</u> (b)
a // b	a. <u>__floordiv__</u> (b)
a % b	a. <u>__mod__</u> (b)
a << b	a. <u>__lshift__</u> (b)
a >> b	a. <u>__rshift__</u> (b)
a & b	a. <u>__and__</u> (b)
a b	a. <u>__or__</u> (b)
a ^ b	a. <u>__xor__</u> (b)
a ** b	a. <u>__pow__</u> (b)
-a	a. <u>__neg__</u> ()
~a	a. <u>__invert__</u> ()
abs(a)	a. <u>__abs__</u> ()

Así, al definir un método `__add__(b)` en la clase `Punto`, por ejemplo, nos permitirá sumar dos instancias de esta clase usando el operador `+`.

```
class Punto():
    ...
    ...
    def __add__(self, b):
        return Punto(self.x + b.x, self.y + b.y)
```

Como en el siguiente ejemplo:

```
>>> a = Punto(1,2)
>>> b = Punto(3,4)
>>> repr(a + b)
'Punto(4, 6)'
```

Métodos especiales para acceder a elementos

Los siguientes métodos se usan para implementar contenedores:

len(x)	x. <u>__len__</u> ()
x[a]	x. <u>__getitem__</u> (a)
x[a] = v	x. <u>__setitem__</u> (a, v)
del x[a]	x. <u>__delitem__</u> (a)

Los podés implementar en tus clases.

```
class Secuencia:  
    def __len__(self):  
        ...  
    def __getitem__(self, a):  
        ...  
    def __setitem__(self, a, v):  
        ...  
    def __delitem__(self, a):  
        ...
```

Invocar métodos

El proceso de invocar un método puede dividirse en dos partes:

1. Búsqueda: Se usa el operador `.`
2. Llamado: Se usan `()`

```
>>> m = Lote('Pera', 100, 490.10)  
>>> c = m.costo # Búsqueda  
>>> c  
<bound method Lote.costo of <Lote object at 0x590d0>>  
>>> c() # Llamado  
49010.0  
>>>
```

Nota: la respuesta al pedido de representación de `c` es algo así como <Método Lote.costo asociado al <objeto Lote en 0x590d0>>*

Métodos ligados

Un método que aún no ha sido llamado por el operador de llamado a funciones `()` se conoce como *método ligado* y opera dentro de la instancia en la que fue originado.

```
>>> m = Lote('Pera', 100, 490.10)  
>>> m  
<Lote object at 0x590d0>  
>>> c = m.costo  
>>> c  
<bound method Lote.costo of <Lote object at 0x590d0>>  
>>> c()  
49010.0  
>>>
```

Estos métodos ligados pueden ser el origen de errores por despropósito, que no son nada obvios. Por ejemplo:

```
>>> m = Lote('Pera', 100, 490.10)
>>> print('Costo : %0.2f' % m.costo)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: float argument required
>>>
```

O una fuente de comportamiento extraño que es difícil de debuggear.

```
f = open(filename, 'w')
...
f.close    # EPA! No hicimos nada. `f` sigue abierto.
```

En ambos casos, el error está causado por omitir los paréntesis en el (intento de) llamado a la función. En estos casos debería haber sido: `m.costo()` o `f.close()`. Sin los paréntesis, no estamos llamando a la función sino refiriéndonos al método.

Acceso a atributos

Existe una forma alternativa de acceder, manipular, y administrar los atributos de un objeto.

```
getattr(obj, 'name')          # Equivale a obj.name
setattr(obj, 'name', value)    # Equivale a obj.name = value
delattr(obj, 'name')          # Equivale a del obj.name
hasattr(obj, 'name')          # Mira si la propiedad existe
```

Ejemplo:

```
if hasattr(obj, 'x'):
    x = getattr(obj, 'x'):
else:
    x = None
```

Nota: si `getattr()` no encuentra el atributo buscado (`x` en este ejemplo), devuelve el argumento opcional `arg` (`None` en este caso)

```
x = getattr(obj, 'x', None)
```

Ejercicios

Ejercicio 8.9: Mejor salida para objetos

Modificá el objeto `Lote` que definiste en `lote.py` (del [Ejercicio 8.1](#)) de modo que el método `__repr__()` genere una salida más agradable. Por ejemplo queremos un comportamiento como éste:

```
>>> peras = Lote('Pera', 100, 490.1)
>>> peras
Lote('Pera', 100, 490.1)
>>>
```

Fijate lo que ocurre cuando leés un camión de frutas y mirás la salida resultante después de hacer estos cambios. Un ejemplo:

```
>>> import informe
>>> camion = informe.leer_camion('Data/camion.csv')
>>> camion
... fijate cuál es la salida ...
>>>
```

Guardá el archivo `lote.py` para entregar.

Ejercicio 8.10: Ejemplo de `getattr()`

`getattr()` es un mecanismo alternativo de leer atributos. Puede usarse para escribir código sumamente versátil. Probá este ejemplo, para empezar:

```
>>> import lote
>>> c = lote.Lote('Peras', 100, 490.1)
>>> columnas = ['nombre', 'cajones']
>>> for colname in columnas:
    print(colname, '=', getattr(c, colname))

nombre = Peras
cajones = 100
>>>
```

Queremos que observes algo interesante: los datos de salida están completamente especificados por los nombres de los atributos listados en la variable `columnas`. Estamos usando el contenido de una variable ('`nombre`' y '`cajones`') como nombres de otras variables, o de atributos de un objeto. No es usual.

Si te dan ganas, en el archivo `formato_tabla.py` usá esta idea pero extendela, y creá una función `imprimir_tabla()` que imprima una tabla mostrando, de una lista de objetos de tipo arbitrario, una lista de atributos especificados por el usuario.

Tal como antes hicimos con la función `imprimir_informe()` del Ejercicio 5.1 `imprimir_tabla()` también debería aceptar cualquier instancia de la clase `FormatoTabla` para definir el formato de la salida. La idea es que funcione más o menos así:

```
>>> import informe
>>> camion = informe.leer_camion('Data/camion.csv')
>>> from formato_tabla import crear_formateador, imprimir_tabla
>>> formateador = crear_formateador('txt')
>>> imprimir_tabla(camion, ['nombre', 'cajones'], formateador)
  nombre      cajones
  ----- 
    Lima        100
  Naranja       50
    Caqui       150
Mandarina     200
  Durazno       95
Mandarina      50
  Naranja      100

>>> imprimir_tabla(camion, ['nombre', 'cajones', 'precio'], formateador)
  nombre      cajones      precio
  -----  -----
    Lima        100      32.2
  Naranja       50      91.1
    Caqui       150     103.44
Mandarina     200      51.23
  Durazno       95      40.37
Mandarina      50      65.1
  Naranja      100      70.44
>>>
```

8.4 Objetos, pilas y colas

En esta sección tendrás que resolver algunos ejercicios definiendo clases y objetos.

Un ejercicio geométrico

Creá una clase llamada `Rectangulo` que va a estar definido por dos puntos. Para esos dos puntos, usá la clase `Punto` de la Sección anterior. El rectángulo es paralelo a los ejes, los puntos representan dos esquinas opuestas cualesquier. La clase debe tener un método constructor para crear el rectángulo a partir de dos puntos y los siguientes métodos:

- `base()` que dé la medida de la base del rectángulo.
- `altura()` que dé la medida de la altura del rectángulo.
- `area()` que dé la medida del área del rectángulo.
- Creá métodos especiales `__str__` y `__repr__`.
- `desplazar(desplazamiento)` que dado un desplazamiento (de tipo Punto) desplace el rectángulo en ambas coordenadas usando el método `add` de la clase Punto.
- `rotar()` que rote el rectángulo sobre su esquina inferior derecha 90 grados a la derecha.

Probá tu código:

```
>>> ul = Punto(0,2)
>>> lr = Punto(1,0)
>>> ll = Punto(0,0)
>>> ur = Punto(1,2)
>>> rect1 = Rectangulo(ul,lr)
>>> rect2 = Rectangulo(ll,ur)
>>> rect1.base()
1
>>> rect1.base()
1
>>> rect2.altura()
2
>>> rect2.altura()
2
>>> rect1.rotar()
>>> rect2.rotar()
>>> rect1.base()
2
>>> rect2.base()
2
>>> rect1.altura()
1
>>> rect2.altura()
1
```

Ejercicio 8.11: Canguros buenos y canguros malos

Este ejercicio está relacionado con un error muy común en Python. Escribí una definición de una clase `Canguro` que tenga:

- Un método `__init__` que inicializa un atributo llamado `contenido_marsupio` como una lista vacía.
- Un método llamado `meter_en_marsupio` que, dado un objeto cualquiera, lo agregue a la lista `contenido_marsupio`.

- Un método `__str__` que devuelve una representación como cadena del objeto `Canguro` y de los contenidos de su marsupio.

Probá tu código creando dos objetos, `madre_canguro` y `cangurito` y guardá en el marsupio de la madre algunos objetos y al cangurito.

Luego, mirá el ejemplo `canguro_malo.py` copiado a continuación. Este ejemplo tiene un bug. Analizalo, corregilo. Entregá como respuesta un archivo `canguros_buenos.py` conteniendo, primero la clase definida por vos y luego una corrección de la clase definida en el ejemplo, junto con un comentario indicando dónde estaba el error y en qué constía.

```
# canguro_malo.py
"""Este código continene un
bug importante y dificil de ver
"""

class Canguro:
    """Un Canguro es un marsupial."""

    def __init__(self, nombre, contenido=[]):
        """Inicializar los contenidos del marsupio.

        nombre: string
        contenido: contenido inicial del marsupio, lista.
        """
        self.nombre = nombre
        self.contenido_marsupio = contenido

    def __str__(self):
        """devuelve una representación como cadena de este Canguro.
        """
        t = [ self.nombre + ' tiene en su marsupio: ' ]
        for obj in self.contenido_marsupio:
            s = '    ' + object.__str__(obj)
            t.append(s)
        return '\n'.join(t)

    def meter_en_marsupio(self, item):
        """Agrega un nuevo item al marsupio.

        item: objeto a ser agregado
        """
        self.contenido_marsupio.append(item)

#%%
madre_canguro = Canguro('Madre')
cangurito = Canguro('gurito')
madre_canguro.meter_en_marsupio('billetera')
madre_canguro.meter_en_marsupio('llaves del auto')
```

```

madre_canguro.meter_en_marsupio(cangurito)

print(madre_canguro)

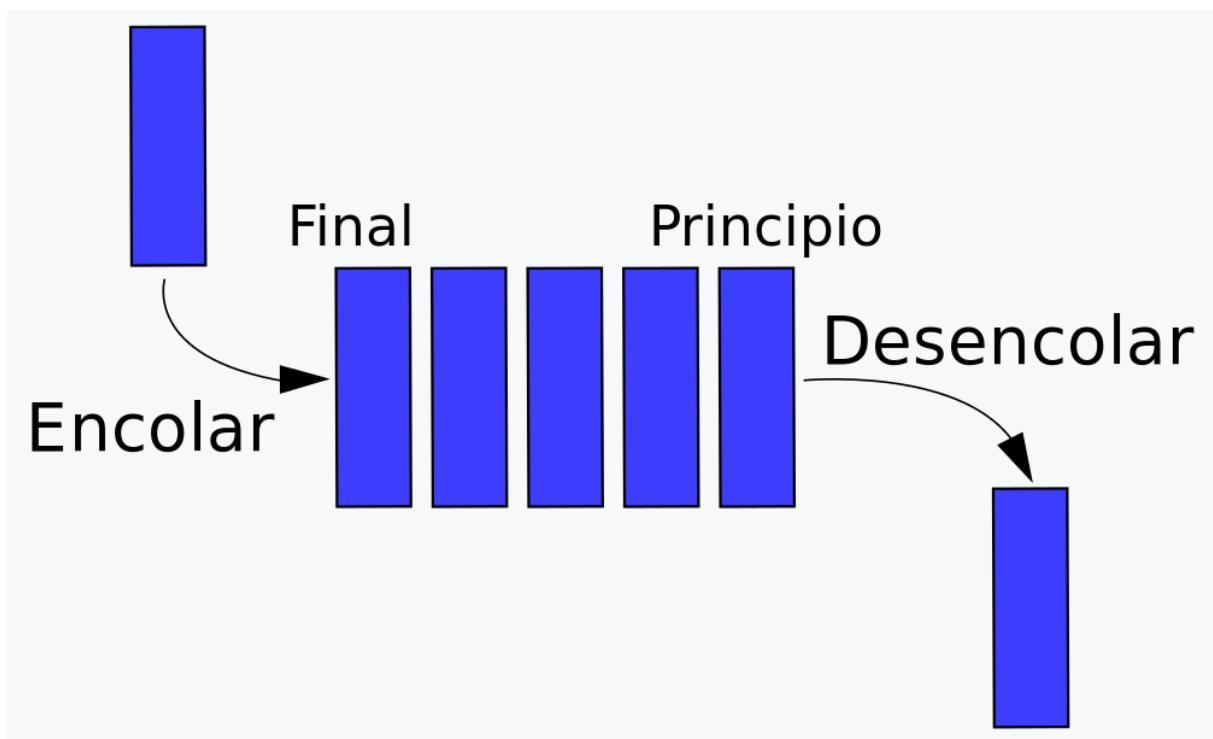
# Al ejecutar este código todo parece funcionar correctamente.
# Para ver el problema, imprimí el contenido de cangurito.

```

Colas

Una cola es una estructura de datos. Se caracteriza por contener una secuencia de elementos y dos operaciones: encolar y desencolar. La primera, encolar, agrega un elemento al final de la secuencia que contiene la cola. Desencolar, por su parte, devuelve el primer elemento de la secuencia y lo elimina de la misma.

Las colas también se llaman estructuras FIFO (del inglés First In First Out), debido a que el primer elemento en entrar a la cola será también el primero en salir. El nombre cola se le da por su analogía con las colas que hacemos (o hacíamos cuando podíamos salir de casa) para entrar al cine, por ejemplo.



Esta es una posible implementación de la clase `Cola`:

```

class Cola:
    '''Representa a una cola, con operaciones de encolar y desencolar.
    El primero en ser encolado es tambien el primero en ser desencolado.'''
    ...

    def __init__(self):

```

```

    '''Crea una cola vacia.'''
    self.items = []

    def encolar(self, x):
        '''Encola el elemento x.'''
        self.items.append(x)

    def desencolar(self):
        '''Elimina el primer elemento de la cola
        y devuelve su valor.
        Si la cola esta vacia, levanta ValueError.'''
        if self.esta_vacia():
            raise ValueError('La cola esta vacia')
        return self.items.pop(0)

    def esta_vacia(self):
        '''Devuelve
        True si la cola esta vacia,
        False si no.'''
        return len(self.items) == 0

```

Ejercicio 8.12: Torre de Control

Usando un par de objetos de la clase `Cola`, escribí una nueva clase llamada `TorreDeControl` que modele el trabajo de una torre de control de un aeropuerto con una pista de aterrizaje. Los aviones que están esperando para aterrizar tienen prioridad sobre los que están esperando para despegar. La clase debe funcionar conforme al siguiente ejemplo:

```

>>> torre = TorreDeControl()
>>> torre.nuevo_arribo('AR156')
>>> torre.nueva_partida('KLM1267')
>>> torre.nuevo_arribo('AR32')
>>> torre.ver_estado()
Vuelos esperando para aterrizar: AR156, AR32
Vuelos esperando para despegar: KLM1267
>>> torre.asignar_pista()
El vuelo AR156 aterrizó con éxito.
>>> torre.asignar_pista()
El vuelo AR32 aterrizó con éxito.
>>> torre.asignar_pista()
El vuelo KLM1267 despegó con éxito.
>>> torre.asignar_pista()
No hay vuelos en espera.

```

Guardá tu solución (conteniendo también la definición de la clase `Cola`) en `torre_control.py` para entregar al final de la clase.

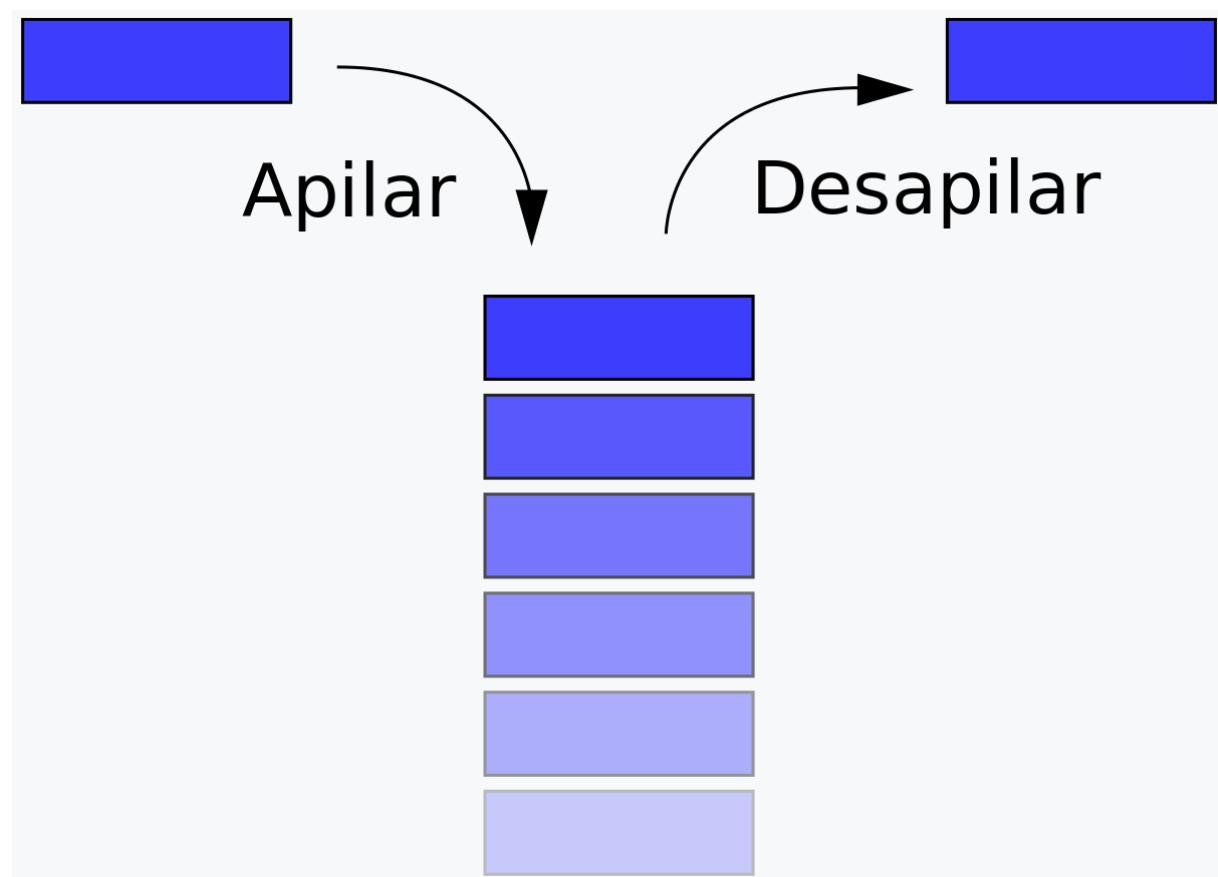
Pilas

Una pila (*stack* en inglés) es una estructura de datos. Se trata de una lista ordenada que permite almacenar y recuperar datos, con un modo de acceso de tipo LIFO (del inglés Last In, First Out, «último en entrar, primero en salir»). Funcionan de manera opuesta que las colas que mencionamos antes.

Las pilas y colas son estructuras de datos que se aplican en multitud de contextos debido a su simplicidad y capacidad de modelar diferentes procesos.

Las operaciones (métodos) elementales de las pilas son *apilar* (coloca un objeto en la pila) y *desapilar* (retira el último elemento apilado). En inglés se llaman *push* y *pop* y son análogos al *encolar* y el *desencolar* de las colas.

En cada momento solamente se tiene acceso a la parte superior de la pila, es decir, al último objeto apilado. La operación *desapilar* justamente permite la obtención de este elemento, que es retirado de la pila.



La pila de llamadas (en inglés *call stack*) de un lenguaje (por ejemplo Python), es una pila manejada por el intérprete que almacena la información sobre las subrutinas activas en cada instante. También se la conoce como pila de ejecución o

pila de control y se usa para llevar registro de las funciones que se fueron llamando y el de las variables definidas en cada contexto.

Por ejemplo, si definimos las siguientes funciones:

```
def f():
    x = 50
    a = 20
    print("En f, x vale", x)

def g():
    x = 10
    b = 45
    print("En g, antes de llamar a f, x vale", x)
    f()
    print("En g, después de llamar a f, x vale", x)
```

la ejecución de `g()` resulta en:

```
>>> g()
En g, antes de llamar a f, x vale 10
En f, x vale 50
En g, después de llamar a f, x vale 10.
```

Para poder volver a recuperar el valor 10 para `x` en `g()` luego de llamar a `f()` se manejó adecuadamente la pila de llamadas. Podemos pensar que en la ejecución de `g()`, justo antes de llamar a `f()` había un *estado* que podría ser resumido en `estado = {función: 'g', próxima_línea_a_ejecutar: 4, variables: {x: 10, b: 45}}`. Luego se ejecuta la cuarta línea de código. El intérprete incrementa `próxima_línea_a_ejecutar` y, antes de llamar a `f()`, apila el `estado` en la pila de llamadas. Al llamar a `f()`, el nuevo estado pasa a ser `estado = {función: 'f', próxima_línea_a_ejecutar: 1, variables: {}}`. El intérprete ejecuta las tres líneas de código de `f`, incrementando la variable `próxima_línea_a_ejecutar` en cada paso, y agregando `x:50` y luego `a:20` el estado de las variables. Por lo tanto, termina la ejecución de `f` en el `estado = {función: 'f', próxima_línea_a_ejecutar: 4, variables: {x: 50, a: 20}}`. Como ya no hay más código que ejecutar de `f()` el intérprete desapila un estado y continúa con la ejecución usando `estado = {función: 'g', próxima_línea_a_ejecutar: 5, variables: {x: 10, b: 45}}`, y por lo tanto imprime:

En g, después de llamar a f, x vale 10.

Estos conceptos son importantes para la clase próxima donde estudiaremos funciones que se llaman a sí mismas *recursivamente*. Si no fuera por la pila de

llamadas, los valores de las variables de las diferentes instancias de una función recursiva correrían el riesgo de mezclarse y confundirse.

Ejercicio 8.13: implementar el TAD pila

Implementá en una clase `Pila` el TAD descripto anteriormente con los métodos `apilar()`, `desapilar()` y `esta_vacia()`.

Usala para reproducir el siguiente código:

```
def mostrar_x_del_estado(estado):
    print(f"Ejecutando {estado['función']}(), x vale {estado['variables']['x']}")
```



```
pila_de_llamadas = Pila()
#la ejecución está en la línea 3 de g(). El estado tiene x=10.
estado = {'función': 'g', 'próxima_línea_a_ejecutar': 3, 'variables': {'x': 10, 'b': 45}}
mostrar_x_del_estado(estado)
#sigo ejecutando, toca llamar a f(): incremento y apilo el estado.
estado['próxima_línea_a_ejecutar'] = 5
pila_de_llamadas.apilar(estado)
#llamo a f y ejecuto primeras líneas
estado = {'función': 'f', 'próxima_línea_a_ejecutar': 3, 'variables': {'x': 50, 'a': 20}}
mostrar_x_del_estado(estado)
#termina ejecución de f: se desapila el estado:
estado = pila_de_llamadas.desapilar()
mostrar_x_del_estado(estado)
```

Su ejecución debería dar:

```
Ejecutando g(), x vale 10
Ejecutando f(), x vale 50
Ejecutando g(), x vale 10
```

8.5 Teledetección

En este ejercicio vamos a trabajar con una imagen satelital obtenida por sensores a bordo del satélite Landsat8. Es un ejercicio optativo para entregar. Si querés, hacelo y guardalo en el archivo `NDVI.py`.

Ejercicio 8.14: Optativo de teledetección

Autora: [Mariela Rajngewerc](#)

La imagen original fue bajada de la página del [earthexplorer](#). En esa página se pueden bajar imágenes con distinto nivel de pre-procesamiento. Para este ejercicio bajamos una imagen de nivel de procesamiento 2, esto quiere decir que los valores de los pixeles representan la reflectancia en superficie en distintas longitudes de onda. [Acá](#) pueden encontrar el manual de estas imágenes donde les detallan la descripción tanto de los nombres de los archivos como de los preprocesamiento que tienen realizados.

Para este ejercicio hemos realizado un clip de cada una de las bandas originales de la imagen y ya multiplicamos a cada una de las bandas por el factor de escala indicado en el manual (0,0001).

Las longitudes de onda y la resoluciones de cada banda de la imagen se describen a continuación:

Banda	Longitud de onda (nanómetros)	Resolución espacial (metros)
Banda 1 - Aerosoles	430 - 450	30
Banda 2 - Azul	450 - 510	30
Banda 3 - Verde	530 - 590	30
Banda 4 - Rojo	640 - 670	30
Banda 5 - Infrarrojo cercano	850 - 880	30
Banda 5 - Infrarrojo medio 1	1570 - 1650	30
Banda 7 - Infrarrojo medio 2	2110 - 2290	30

Si desean abrir los datos de la imagen original en Python deberán bajar algunas librerías específicas para la manipulación de datos satelitales, por ejemplo: gdal. [Acá](#) hay un tutorial de los primeros pasos.

En la carpeta [clip](#) encontrarán los datos que vamos a usar en los ejercicios. Cada banda del clip se encuentra en formato .npy

Ejercicios:

Ejercicio 8.15: Ver una banda

a) Usá [numpy](#) para levantar cada una de las bandas y `plt.imshow(banda)` para verla. ¿Se ve correctamente? Podés ajustar el rango de visualización de colores usando los parámetros `vmin` y `vmax`.

Sugerencia: Con `plt.hist(banda.flatten(), bins=100)` vas a ver un histograma de los valores en la matriz `banda`. Podés usarlo para guiarte en la búsqueda del rango que tiene sentido usar como `vmin` y `vmax`.

b) Probá usando percentiles para fijar el rango. Algo como

```
vmin = np.percentile(data.flatten(), q)
vmax = np.percentile(data.flatten(), 100-q)
```

c) Escribí una función `crear_img_png(carpeta, banda)` que, dada una carpeta y un número de banda, muestre la imagen de dicha banda y la guarde en formato .png. Asegurate de incorporar un `colorbar` al lado de la imagen.

Tené en cuenta lo que hiciste en los puntos anteriores para que se vea adecuadamente.

Ejercicio 8.16: Histogramas

Escribí ahora otra función, llamada `crear_hist_png(carpeta, banda, bins)` que, dada una carpeta, un número de banda y una cantidad de bins, muestre el histograma (con la cantidad de bins seleccionados) de los valores de dicha banda y la guarde en formato .png.

Ejercicio 8.17: Máscaras binarias

a) Usá las funciones `crear_img_png` y `crear_hist_png` que hiciste en los puntos anteriores para generar las imágenes e histogramas de cada banda.

b) ¿Qué banda o bandas parecieran tener histogramas bimodales, mostrando diferentes tipos de pixels? Elegí una de esas bandas y, observando el histograma, seleccioná un umbral que te permita distinguir los dos tipos de pixels. Por ejemplo, podés crear una matriz del mismo tamaño de la banda donde a cada pixel le corresponda un 1 o un 0, 1 si está por arriba del umbral y 0 si no.

Graficá la imagen binaria así obtenida. ¿A qué corresponden los dos tipos de píxeles que pudiste distinguir tan fácilmente?

Ejercicio 8.18: Clasificación manual

En este ejercicio vamos a trabajar con un índice: el Índice de Vegetación de Diferencia Normalizada, también conocido como **NDVI** por sus siglas en inglés. Este índice, basado en la intensidad de la radiación de dos bandas del espectro electromagnético que interactúan particularmente con la vegetación, aporta información sobre la cantidad, estado y desarrollo de la misma.

Para calcular el NDVI se utilizan las bandas espectrales Roja e Infrarroja y el cálculo se hace mediante la siguiente fórmula:

$$(\text{INFRARROJO_CERCANO} - \text{ROJO}) / (\text{INFRARROJO_CERCANO} + \text{ROJO})$$

a) Calcular el NDVI en una nueva matriz.

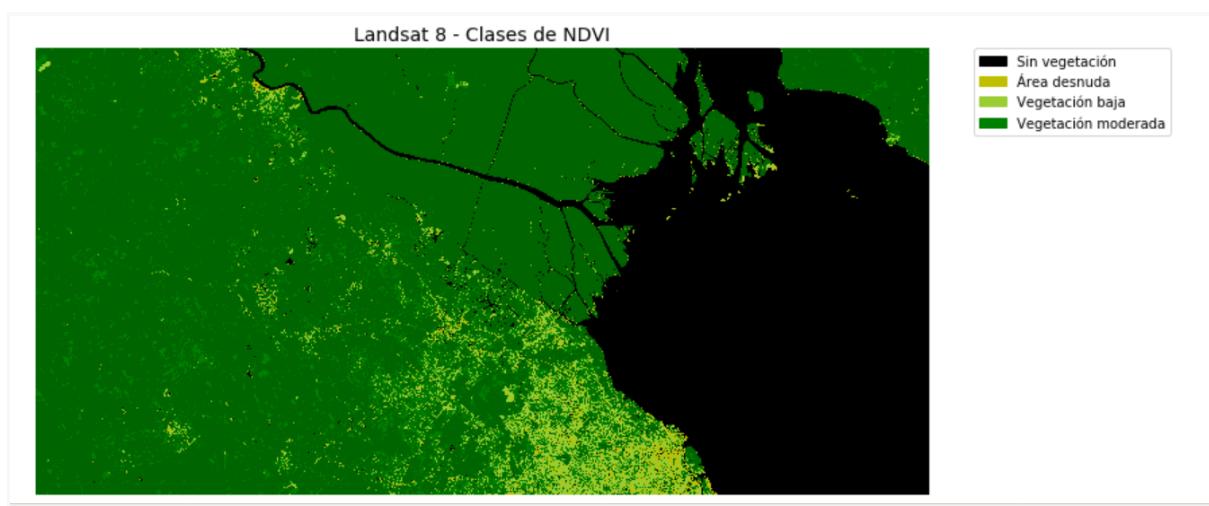
b) Categorizá los valores obtenidos en cada píxel de acuerdo a clases que nos sean más útiles y fáciles de interpretar. La tabla a continuación muestra una propuesta de categorías que podés considerar:

Valor de NDVI	Nombre de la clase	Identificador de Clase	color
< 0	No vegetada	0	black
entre 0 y 0.1	Área desnuda	1	y
entre 0.1 y 0.25	Vegetación baja	2	yellowgreen
entre 0.25 y 0.4	Vegetación moderada	3	g

>0.4	Vegetación densa	4	darkgreen
------	------------------	---	-----------

Creá un `np.array` que le asigne a cada píxel el número dado por el *identificador de categoría* correspondiente según la tabla. Llamá `clases_ndvi` a la matriz así obtenida.

- c) Generá un gráfico con `matplotlib` mostrando las clases obtenidas.
- d) Crear un `colorMap` para lograr asignarle a cada clase el color sugerido en la tabla. Para esto podés usar la función `ListedColormap` incluída en `matplotlib.colors` y crear un `colorMap` (`cmap`).
- e) Ponele una leyenda que indique el nombre de cada clase con el color asignado, para eso te sugerimos usar la función `mpatches` que se encuentra en `matplotlib.patches`. Para que puedas orientarte, te mostramos a continuación un ejemplo de resultado esperado:



Si llegaste hasta acá, no te olvides de guardar tu trabajo en el archivo `NDVI.py` y entregarlo. A continuación, un ejercicio que usa herramientas un poco más avanzadas de aprendizaje automático.

Ejercicio 8.19: Clasificación automática

En el ejercicio anterior definimos a mano los umbrales que distinguen las clases. Es posible hacer esto de forma automática. Para eso se usan técnicas de clustering. El siguiente código muestra un ejemplo con un clasificador muy sencillo: `kmeans`. Este clasificador está ya implementado en la biblioteca `sklearn` que es una biblioteca dedicada al aprendizaje automático en python (probablemente la más usada para esto).

```
# filtro datos ruidosos o que puedan traer problemas.
```

```

# el NDVI debe estar entre -1 y 1.
ndvi[ndvi>1]=1
ndvi[ndvi<-1]=-1

#importo el clasificador y defino una instancia para clasificar con dos
etiquetas
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=2)

#le saco la estructura bidimensional a la matriz NDVI y la llamo datos
datos = ndvi.reshape(-1,1) #datos es un vector con un dato de NDVI por
pixel.

#entreno o ajusto el el clasificador con los datos (demora!)
kmeans.fit(datos) #ajusta el modelo
#usa el modelo ajustado para poner etiquetas
etiquetas = kmeans.predict(ndvi.reshape(-1,1))

#visualizo los resultados recuperando la estructura bidimensional de la
matriz
plt.imshow(etiquetas.reshape(ndvi.shape))

```

Probá ajustando el número de clusters (`n_clusters=5`, por ejemplo) y corriendo nuevamente el modelo. Ponele colores diferentes a las diferentes clases obtenidas.

Si tarda mucho podés trabajar con un pedazo de la imagen. Por ejemplo si hacés `ndvi_clip = ndvi[1000:2000,2000:3000]` te quedás con un cuadradito que es un octavo de la imagen original y podés usarlo para probar cosas rápido. Si te convencen los resultados podés correr tu algoritmo sobre la imagen completa.

8.6 Cierre de la octava clase

Cierre de clase

En esta clase vimos las ventajas de estructurar un programa con las ideas Programación Orientada a Objetos. Vimos cómo podés definir tus propias clases y te mostramos cómo este paradigma puede aportar tanto a la organización de un programa "estático" como también para preparar programas para que sean fácilmente extensibles.

Una de las formas de extender el comportamiento de un programa es definir una interfase de interacción entre objetos de modo que un comportamiento nuevo pueda programarse sin tocar (casi) el código preexistente.

Otra forma muy interesante es definir clases base, abstractas, que van a ser implementadas luego, por herencia, en sus versiones definitivas.

La experiencia te va a permitir decidir (quizás tardíamente) cuándo es más conveniente una arquitectura o la otra.

En fin, para cerrar esta clase, entregá:

- El archivo `informe.py` del [Ejercicio 8.8](#).
- El archivo `lote.py` del [Ejercicio 8.9](#).
- El archivo `torre_control.py` del [Ejercicio 8.12](#).
- El archivo `canguros_buenos.py` del [Ejercicio 8.11](#).
- El archivo `NDVI.py` del [Ejercicio 8.14](#) (optativo).

Además te pedimos que completes [este formulario](#) usando tu dirección de mail como identificación. Al terminar vas a obtener un link para enviarnos tus ejercicios.

¡Nos vemos!

9. Generadores e iteradores

Programar es básicamente escribir condicionales, ciclos y asignaciones de variables. Aunque no de cualquier forma.

Los ciclos, ya sean ciclos `while` o iteraciones `for` son una de las estructuras más ubicuas en cualquier lenguaje. Los programas hacen muchas iteraciones para procesar listas, leer archivos, buscar en bases de datos y demás.

Una de las características más poderosas de Python es la capacidad de redefinir la iteración mediante las llamadas "funciones generadoras". En esta sección veremos de qué se trata ésto. Hacia el final vas a escribir programas que procesan datos en tiempo real, a medida que son generados.

Terminamos la clase con un ejercicio optativo que combina dos temas importantes: objetos y simulaciones. El ejercicio optativo propone simular en el espacio y tiempo la dinámica predador-presa utilizando para esto programación orientada a objetos.

- [9.1 El protocolo de iteración](#)
- [9.2 Iteración a medida](#)
- [9.3 Productores, consumidores y cañerías.](#)
- [9.4 Más sobre generadores](#)
- [9.5 Predador Presa](#)
- [9.6 Cierre de la novena clase](#)

9.1 El protocolo de iteración

En esta sección vemos lo que realmente sucede en Python durante una proceso de iteración.

Iteraciones por doquier

Podemos iterar sobre una gran diversidad de objetos.

```
a = 'hola a todos'  
for c in a: # Iterar las letras en a  
    ...  
  
b = { 'nombre': 'Elsa', 'password':'foo'}  
for k in b: # Iterar para cada clave de diccionario  
    ...  
  
c = [1,2,3,4]  
for i in c: # Iterar sobre los items en una lista ó tupla  
    ...  
  
f = open('foo.txt')  
for x in f: # Iterar sobre las líneas de un archivo ASCII  
    ...
```

Podemos iterar sobre todos estos objetos porque cumplen con un *protocolo* que permite, justamente, iterar. Veamos algo sobre este protocolo:

El protocolo de iteración

Analicemos la instrucción `for`.

```
for x in obj:
```

```
# instrucciones
```

¿Cómo funciona realmente ésto? Mediante un protocolo de iteración que puede resumirse así:

```
_iter = obj.__iter__()          # Buscar el objeto iterador
while True:
    try:
        x = _iter.__next__()    # Dame el siguiente item
    except StopIteration:      # No hay más items
        break
    # instrucciones ...
```

Todo objeto compatible con un ciclo `for` implementa, a bajo nivel, este protocolo de iteración.

Un ejemplo: Iteración manual sobre una lista.

```
>>> x = [1,2,3]
>>> it = x.__iter__()
>>> it
<listiterator object at 0x590b0>
>>> it.__next__()
1
>>> it.__next__()
2
>>> it.__next__()
3
>>> it.__next__()
Traceback (most recent call last):
File "<stdin>", line 1, in ?
StopIteration
>>>
```

Fijate que al agotar los elementos, el `traceback` acusa una excepción de tipo `StopIteration`. Fijate también que en la tercera línea, al preguntar por `it`, python responde <es un objeto de tipo listiterator (iterador de listas) alojado en 590b0 hexadecimal> .

Iterable

Es necesario que entiendas los mecanismos de iteradores si querés permitir iteración sobre objetos que vos definas, es decir, hacerlos *iterables*. El siguiente ejemplo construye un contenedor iterable, simplemente basado en una lista:

```
class Camion:
    def __init__(self):
        self.lotes = []
```

```
def __iter__(self):
    return self.lotes.__iter__()
...
camion = Camion()
for c in camion:
    ...
```

Ejercicios

Ejercicio 9.1: Iteradores, un ejemplo

Construí la siguiente lista:

```
a = [1, 9, 4, 25, 16]
```

Y ahora iterá sobre esa lista *a mano*: Llamá al método `__iter__()` para obtener un objeto iterador y llama al método `__next__()` para obtener sucesivamente cada uno de los elementos.

```
>>> i = a.__iter__()
>>> i
<listiterator object at 0x64c10>
>>> i.__next__()
1
>>> i.__next__()
9
>>> i.__next__()
4
>>> i.__next__()
25
>>> i.__next__()
16
>>> i.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

La función nativa de Python `next()` es un "atajo" al método `__next__()` de un iterador. Probá usarlo a mano sobre un archivo:

```
>>> f = open('Data/camion.csv')
>>> f.__iter__()      # Notar que esto apunta al método...
                           # ...que accede al archivo mismo.
<_io.TextIOWrapper name='Data/portfolio.csv' mode='r' encoding='UTF-8'>
```

```

>>> next(f)
'nombre,cajones,precio\n'
>>> next(f)
'Lima,100,32.20\n'
>>> next(f)
'Naranja,50,91.10\n'
>>>

```

Llamá a `next(f)` hasta que llegues al final del archivo, y fijate qué sucede.

Ejercicio 9.2: Iteración sobre objetos

Como decíamos en la sección [Sección 9.1](#), cuando definás tus propios objetos, es posible que quieras que se pueda iterar sobre ellos (especialmente si estos objetos son "envoltorios" (wrappers) para listas u otros iterables). Hagamos esto: en un nuevo archivo llamado `camion.py`, definí la siguiente clase:

```

# camion.py

class Camion:

    def __init__(self, lotes):
        self._lotes = lotes

    def precio_total(self):
        return sum([l.costo() for l in self._lotes])

    def contar_cajones(self):
        from collections import Counter
        cantidad_total = Counter()
        for l in self._lotes:
            cantidad_total[l.nombre] += l.cantidad
        return cantidad_total

```

La intención es crear un envoltorio para una lista, y de paso agregarle algunos métodos, como la propiedad de calcular el costo total del camión. Vamos a usar lo que hiciste en el [Ejercicio 8.1](#). Modificá la función `leer_camion()` en `informe.py` de modo que cree una instancia de `Camion`, como se muestra:

```

# informe.py
...

import fileparse
from lote import Lote
from camion import Camion

def leer_camion(filename):
    """
    Lee un archivo con el contenido de un camión

```

```

y lo devuelve como un objeto Camion.
...
with open(filename) as file:
    camiondicts = fileparse.parse_csv(file,
select=['nombre','cajones','precio'],
                                types=[str,int,float])

    camion = [ Lote(d['nombre'], d['cajones'], d['precio']) for d in
camiondicts ]
return Camion(camion)
...

```

Ahora intentá correr el programa `informe.py`. No hay forma. `informe.py` intenta iterar sobre las instancias de `Camion` pero éstas no son iterables y el programa no funciona.

```

>>> import informe
>>> informe.informe_camion('Data/camion.csv', 'Data/precios.csv')
... muere ...

```

La forma de arreglar este programa roto es modificar la clase `Camion` y hacerla iterable.

```

class Camion:

    def __init__(self, lotes):
        self._lotes = lotes

    def __iter__(self):
        return self._lotes.__iter__()

    def precio_total(self):
        return sum([l.costo() for l in self._lotes])

    def contar_cajones(self):
        from collections import Counter
        cantidad_total = Counter()
        for lote in self._lotes:
            cantidad_total[lote.nombre] += lote.cajones
        return cantidad_total

```

Después de haber hecho este cambio, tu `informe.py` debería estar funcionando de nuevo. Guardá esta versión de `informe.py` para entregar al final de la clase (en el próximo ejercicio te pediremos también `camion.py`).

Y ya que estás, cambiá el programa `costo_camion.py` para que use objetos que sean instancias de la clase `Camion`, por ejemplo así:

```
# costo_camion.py

import informe

def costo_camion(filename):
    """
    Computa el precio total (cantidad * precio) de un archivo camion
    """
    camion = informe.leer_camion(filename)
    return camion.precio_total()
...
```

Testealo, testealo, y testealo para asegurarte que funciona:

```
>>> import costo_camion
>>> costo_camion.costo_camion('Data/camion.csv')
47671.15
>>>
```

Ejercicio 9.3: Un iterador adecuado

Cuando hagas clases que sean recipientes ó contenedores de estructuras de datos vas a necesitar que hagan algo más que simplemente iterar. Probá modificar la clase `Camion` de modo que tenga algunos de los "métodos mágicos" que mencionamos en la [Sección 8.3](#). Aquí hay algunos:

```
class Camion:
    def __init__(self, lotes):
        self._lotes = lotes

    def __iter__(self):
        return self._lotes.__iter__()

    def __len__(self):
        return len(self._lotes)

    def __getitem__(self, index):
        return self._lotes[index]

    def __contains__(self, nombre):
        return any([lote.nombre == nombre for lote in self._lotes])

    def precio_total(self):
        return sum([l.costo() for l in self._lotes])
```

```

def contar_cajones(self):
    from collections import Counter
    cantidad_total = Counter()
    for l in self._lotes:
        cantidad_total[l.nombre] += l.cajones
    return cantidad_total

```

Por último, probemos esta nueva estructura:

```

>>> import informe
>>> camion = informe.leer_camion('Data/camion.csv')
>>> len(camion)
7
>>> camion[0]
Lote('Lima', 100, 32.2)
>>> camion[1]
Lote('Naranja', 50, 91.1)
>>> camion[0:3]
[Lote('Lima', 100, 32.2), Lote('Naranja', 50, 91.1), Lote('Caqui', 150,
83.44)]
>>> 'Naranja' in camion
True
>>> 'Manzana' in camion
False
>>>

```

Guardá tu versión de `camion.py` con estos cambios para entregar y para la revisión de pares.

Un comentario importante sobre todo esto: Se considera *de buen estilo Python* al código que comparte ciertas normas de interacción con el resto del mundo Python. Este concepto aplicado a objetos contenedores significa que estos cumplen con las buenas costumbres de ser iterables, indexables y que admiten otras operaciones que naturalmente se esperan *a priori* que vayan a cumplir, justamente por el simple hecho de ser objetos contenedores

9.2 Iteración a medida

En esta sección introducimos el concepto de función generadora. Estas funciones te permiten obtener el iterador que necesites.

Un problema de iteración

Suponé que querés crear una secuencia particular de iteración: una cuenta regresiva, por decir algo.

```
>>> for x in regresiva(10):
...     print(x, end=' ')
...
10 9 8 7 6 5 4 3 2 1
>>>
```

Existe una forma fácil de hacer esto.

Generadores

Un generador es una función que define un patrón de iteración.

```
def regresiva(n):
    while n > 0:
        yield n
        n -= 1
```

Nota: "yield" se traduce como "rendir" ó "entregar"

Por ejemplo:

```
>>> for x in regresiva(10):
...     print(x, end=' ')
...
10 9 8 7 6 5 4 3 2 1
>>>
```

Un generador es *cualquier* función que usa el comando `yield`.

El comportamiento de los generadores es algo diferente al del resto de las funciones.

Al llamar a un generador creás un objeto generador, pero su función no se ejecuta de inmediato.

```
def regresiva(n):
    # Agreguemos este print para ver qué pasa...
    print('Cuenta regresiva desde', n)
    while n > 0:
        yield n
        n -= 1
>>> x = regresiva(10)
# No se ejecuta ningún PRINT !
>>> x
```

```
# sin embargo x es un objeto generador
<generator object at 0x58490>
>>>
```

La función sólo se ejecuta ante un llamado al método `__next__()`

```
>>> x = regresiva(10)
>>> x
<generator object at 0x58490>
>>> x.__next__()
Cuenta regresiva desde 10
10
>>>
```

Lo que hace `yield` es notable: produce un valor, y luego suspende la ejecución de la función. La ejecución continúa al volver a llamar a `__next__()`.

```
>>> x.__next__()
9
>>> x.__next__()
8
```

Cuando finalmente se llega al final de la función, la iteración da un error.

```
>>> x.__next__()
1
>>> x.__next__()
Traceback (most recent call last):
File "<stdin>", line 1, in ?
StopIteration
>>>
```

Observación: Una función generadora implementa el mismo protocolo de bajo nivel que los `for` usan sobre listas, tuplas, diccionarios, archivos, etc. ¡Por eso es tan sencillo usar los generadores para iterar!

Ejercicios

Ejercicio 9.4: Un generador simple

Si te encontrás con la necesidad de obtener una iteración particular, pensá en usar funciones generadoras. Son fáciles de escribir: simplemente hacé una función que implemente la lógica de iteración deseada y use `yield` para entregar valores.

Por ejemplo, probá este generador que busca un archivo y entrega las líneas que incluyen cierto substring.

```
>>> def filereturn(filename, substr):
    with open(filename, 'r') as f:
        for line in f:
            if substr in line:
                yield line

>>> for line in open('Data/camion.csv'):
    print(line, end='')

nombre,cantidad,precio
"Lima",100,32.20
"Naranja",50,91.10
"Limón",150,83.44
"Mandarina",200,51.23
"Durazno",95,40.37
"Mandarina",50,65.10
"Naranja",100,70.44
>>> for line in filereturn('Data/camion.csv', 'Naranja'):
    print(line, end='')

"Naranja",50,91.10
"Naranja",100,70.44
>>>
```

Esta idea es muy interesante: podés armar una función que encapsule todo el procesamiento de datos y después recorrerla con un ciclo `for` para que te entregue los datos uno a uno.

El próximo ejemplo es de un caso aún más especial.

Ejercicio 9.5: Monitoreo de datos en tiempo real.

Un generador puede ser una forma interesante de vigilar datos a medida que son producidos. En esta sección vamos a probar esa idea. Para empezar, hacé lo siguiente.

Vas a necesitar dos archivos del repositorio de la materia: [sim_mercado.py](#) y [mccentral.csv](#). El programa `sim_mercado.py` es un generador de datos de precios que toma como referencia a `Data/mcentral.csv`. Al ejecutarlo, el programa escribe datos (con una componente aleatoria) en un archivo llamado `Data/mercadolog.csv` continuamente hasta que es detenido. Se ejecuta indefinidamente: una vez que inicies su ejecución podés dejarlo correr y olvidarte de él. Abrí una consola del

sistema operativo nueva y ejecutá el programa. Si estás en Windows, dale un doble click al ícono de `sim_mercado.py`, ó desde unix:

```
bash % python3 sim_mercado.py
```

Después, olvidate de él. Dejalo ahí, corriendo.

Usando otra consola, mirá el contenido de `Data/mercadolog.csv`. Vas a ver que cada tanto se agrega una nueva línea al archivo.

Ahora que el programa generador de datos está en ejecución, escribamos un programa que abra el archivo, vaya al final, y espere nuevos datos. Para esto creá un programa llamado `vigilante.py` (es uno de los ejercicios a entregar) que contenga el siguiente código.

```
# vigilante.py
import os
import time

f = open('Data/mercadolog.csv')
f.seek(0, os.SEEK_END)      # Mover el índice 0 posiciones desde el EOF

while True:
    line = f.readline()
    if line == '':
        time.sleep(0.5)    # Esperar un rato y
        continue           # vuelve al comienzo del while
    fields = line.split(',')
    nombre = fields[0].strip('"')
    precio = float(fields[1])
    volumen = int(fields[2])
    if volumen > 1000:
        print(f'{nombre:>10s} {precio:>10.2f} {volumen:>10d}')
```

Nota: *EOF = End Of File (fin de archivo)*

Cuando ejecutes el programa vas a ver un indicador de precios en el mercado en tiempo real, con indicación de qué producto se trata, cuál es su precio, y cuál es el volumen de la operación (en cantidad de cajones).

Observación: La forma en que usamos el método `readline()` en este ejemplo es un poco rara, no es la forma en que se suele usar (dentro de un ciclo `for` para recorrer el contenido de un archivo). En este caso la estamos usando para mirar constantemente el fin de archivo para obtener los últimos datos que se hayan agregado (`readline()` devuelve ó bien el último dato o bien una cadena vacía).

Ejercicio 9.6: Uso de un generador para producir datos

Si analizás un poco el código del [Ejercicio 9.5](#) vas a notar que la primera parte del programa "produce" datos (los obtiene del archivo) y la segunda los procesa y los imprime, es decir "consume" datos. Una característica importante de las funciones generadoras es que podés mover todo el código a una función reutilizable. Probalo vos.

Modificá el código del [Ejercicio 9.5](#) de modo que la lectura del archivo esté resuelta por una única función generadora `vigilar()` a la que se le pasa un nombre de archivo como parámetro. Hacelo de modo que el siguiente código funcione:

```
>>> for line in vigilar('Data/mercadolog.csv'):
    print(line)

... Acá deberías ver las líneas impresas ...
```

Modificá el programa `vigilante.py` de modo que tenga esta pinta:

```
if __name__ == '__main__':
    for line in vigilar('Data/mercadolog.csv'):
        fields = line.split(',')
        nombre = fields[0].strip('\'')
        precio = float(fields[1])
        volumen = int(fields[2])
        if volumen > 1000:
            print(f'{nombre:>10s} {precio:>10.2f} {volumen:>10d}')
```

Guradá esta versión de `vigilante.py` para entregar al final de la clase.

Ejercicio 9.7: Cambios de precio de un camión

Modificá el programa `vigilante.py` para que sólo informe las líneas que tienen precios de lotes incluídos en un camión, e ignore el resto de los productos. Por ejemplo:

```
if __name__ == '__main__':
    import informe

    camion = informe.leer_camion ('Data/camion.csv')

    for line in vigilar('Data/mercadolog.csv'):
        fields = line.split(',')
        nombre = fields[0].strip('\'')
        precio = float(fields[1])
        volumen = int(fields[2])
```

```
if nombre in camion:  
    print(f'{nombre:>10s} {precio:>10.2f} {volumen:>10d}')
```

Observación: para que esto funcione, tu clase `Camion` tiene que haber implementado el operador `in`. Controlá que tu solución al ejercicio [Ejercicio 9.3](#) implemente el operador `__contains__()`.

Discusión

Lo que acaba de suceder es algo con mucho potencial: moviste tu patrón de iteración (el que toma las últimas líneas de un archivo) y lo pusiste en su propia función. La función `vigilar()` ahora es una función de uso amplio, que podés usar en cualquier programa. Por ejemplo, la podrías usar para mirar el log (historial) de un servidor ó de un debugger, o de otras fuentes continuas de datos. ¿No está bueno?

9.3 Productores, consumidores y cañerías.

Los generadores son una herramienta muy útil para configurar pipelines (cañerías). Este concepto requiere una breve *aclaración*: Un pipeline tradicional en computación consta de una serie de programas y archivos asociados que constituyen una estructura de procesamiento de datos, donde cada programa se ejecuta independientemente de los demás, pero juntos resultan en un flujo conveniente de datos a través de los archivos asociados desde un "productor" (una cámara, un sensor, un lector de código de barras) hasta un "consumidor" (un graficador, un interruptor eléctrico, un log de una página web). Construiste un pequeño pipeline en la sección anterior, usando `vigilante.py`.

En esta sección hablaremos de cómo implementar estas estructuras de productores y consumidores de datos con generadores en Python.

Sistemas productor-consumidor

El concepto de generadores está íntimamente asociado a problemas de tipo productor-consumidor en sus varias formas. Fíjate esta estructura, que es típica de muchos programas:

```
# Productor
def vigilar(f):
    ...
    while True:
        ...
        yield linea      # Produce/obtiene valores para "linea"
        ...

# Consumidor
for linea in vigilar(f):    # Consume líneas del `yield`
    ...
```

Los `yield` generan los datos que los `for` consumen.

Pipelines con generadores

Podés usar esta característica de los generadores para construir *pipelines* que procesen tus datos, un concepto que es muy usado en Unix (pipes) pero en Windows se usa menos.

productor → procesamiento → procesamiento → consumidor

Los *pipelines* de procesamiento de datos tienen un productor al comienzo, una cadena de etapas de procesamiento y un consumidor al final.

productor → procesamiento → procesamiento → consumidor

```
def productor():
    ...
    yield item
    ...
```

El productor es en general un generador, aunque también podría ser una lista o cualquier otra secuencia iterable.

El `yield` alimenta al pipeline de datos.

productor → procesamiento → procesamiento → consumidor

```
def consumidor(s):
    for item in s:
        ...
```

El consumidor es un ciclo `for`. Obtiene los elementos `item` y los usa para algo.

productor → procesamiento → procesamiento → consumidor

```
def procesamiento(s):
    for item in s:
        ...
        yield itemnuevo
    ...
```

Las etapas intermedias de procesamiento simultáneamente consumen y producen datos, pueden alterar el flujo de datos, eliminar o modificar datos según su función.

productor → procesamiento → procesamiento → consumidor

```
def productor():
    ...
    yield item          # yield devuelve un item que será recibido por
`procesamiento`
    ...

def procesamiento(s):
    for item in s:      # item viene del `productor`
        ...
        yield newitem   # este yield devuelve un nuevo item
    ...

def consumidor(s):
    for item in s:      # item viene de `procesamiento`
    ...
```

Vamos a construir un pipeline con la siguiente arquitectura:

```
a = productor()
b = procesamiento(a)
c = consumidor(b)
```

Como te darás cuenta, los datos van pasando de una función a la siguiente.

Ejercicios

Para este ejercicio, necesitás que el programa `sim_mercado.py` aún esté corriendo. Vas a usar la función `vigilar()` que escribiste en el [Ejercicio 9.7](#)

Ejercicio 9.8: Configuremos un pipeline simple

Escribí la siguiente función y veamos como funciona un pipeline.

```
>>> def filematch(lines, substr):
    for line in lines:
        if substr in line:
            yield line

>>>
```

Esta función es casi idéntica al primer ejemplo de generador en el ejercicio anterior, salvo que ya no abre un archivo sino que opera directamente de una secuencia de líneas que recibe como argumento. Ahora probá lo siguiente:

```
>>> lines = vigilar('Data/mercadolog.csv')
>>> naranjas = filematch(lines, 'Naranja')
>>> for line in naranjas:
    print(line)

... esperá que aparezca la salida ...
```

Puede pasar que tarde unos segundos en darte una salida, pero vas a ver información sobre naranjas tan pronto como sean añadidas al archivo por el primer generador.

Ejercicio 9.9: Un pipeline más en serio

Llevemos esta idea un poco más lejos. Probemos esto:

```
>>> from vigilante import vigilar
>>> import csv
>>> lineas = vigilar('Data/mercadolog.csv')
>>> filas = csv.reader(lineas)
>>> for fila in filas:
    print(fila)

...
['Rabanito', ' 249.37', ' 357']
['Batata', ' 15.75', ' 1040']
['Rabanito', ' 211.31', ' 324']
['Zuccini', ' 83.12', ' 612']
...
```

¡Interesante! La salida de la función `vigilar()` fué usada como entrada a la función `csv.reader()` (que habíamos usado para leer un archivo del disco) y el resultado es una secuencia de filas "parseadas", separadas por las comas.

Ejercicio 9.10: Un pipeline más largo

Veamos si podemos construir un pipeline más largo basado en la misma idea.

Creá un archivo nuevo llamado `ticker.py`, te lo vamos a pedir al final de la clase. Comenzá creando una función que lea un archivo CSV como hiciste antes:

```
# ticker.py

from vigilante import vigilar
import csv

def parsear_datos(lines):
    rows = csv.reader(lines)
    return rows

if __name__ == '__main__':
    lines = vigilar('Data/mercadolog.csv')
    rows = parsear_datos(lines)
    for row in rows:
        print(row)
```

Escríbí una función nueva que elija algunas columnas específicas:

```
# ticker.py
...
def elegir_columnas(rows, indices):
    for row in rows:
        yield [row[index] for index in indices]
...
def parsear_datos(lines):
    rows = csv.reader(lines)
    rows = elegir_columnas(rows, [0, 2])
    return rows
```

Ejecútalo de nuevo, Sam. La salida ahora debería estar restringida a esto:

```
['Brócoli', ' 388']
['Ajo', ' 120']
['Caqui', ' 108']
['Mandarina', ' 1170']
...
```

Escribí funciones generadoras que conviertan el tipo de datos a diccionarios:

```
# ticker.py
...
def cambiar_tipo(rows, types):
    for row in rows:
        yield [func(val) for func, val in zip(types, row)]

def hace_dicts(rows, headers):
    for row in rows:
        yield dict(zip(headers, row))

def parsear_datos(lines):
    rows = csv.reader(lines)
    rows = elegir_columnas(rows, [0, 1, 2])
    rows = cambiar_tipo(rows, [str, float, float])
    rows = hace_dicts(rows, ['nombre', 'precio', 'volumen'])
    return rows
...
```

Correlo de nuevo. Ahora la salida debería ser una serie de diccionarios:

```
{'nombre': 'Frutilla', 'precio': 276.81, 'volumen': 249.0}
{'nombre': 'Morrón', 'precio': 2988.42, 'volumen': 702.0}
{'nombre': 'Morrón', 'precio': 3108.63, 'volumen': 498.0}
{'nombre': 'Naranja', 'precio': 11.5, 'volumen': 870.0}
...
```

Ejercicio 9.11: Filtremos los datos

Para seguir agregando procesamiento a nuestro pipeline, escribí un filtro de datos:

```
# ticker.py
...
def filtrar_datos(filas, nombres):
    for fila in filas:
        if fila['nombre'] in nombres:
            yield fila
```

Esto se usa para dejar pasar únicamente aquéllos lotes incluídos en el camión. Probalo.

```
import informe
camion = informe.leer_camion('Data/camion.csv')
filas = parsear_datos(vigilar('Data/mercadolog.csv'))
```

```
filas = filtrar_datos (filas, camion)
for fila in filas:
    print(fila)
```

Ejercicio 9.12: El pipeline ensamblado

En el programa `ticker.py` (esta versión te vamos a pedir que la entregues) escribí una función `ticker(camion_file, log_file, fmt)` que cree un indicador en tiempo real para un camión, archivo log, y formato de tabla de salida particulares. Fijate:

```
>>> from ticker import ticker
>>> ticker('Data/camion.csv', 'Data/mercadolog.csv', 'txt')
  Nombre      Precio      Volumen
  ----- -----
  Caqui       796.73        96
  Mandarina   12.12       1120
  Lima        2659.37       222
  Naranja     11.70       1040
  Durazno     281.76       704
...
...
>>> ticker('Data/camion.csv', 'Data/mercadolog.csv', 'csv')
Nombre,Precio,Volumen
Mandarina,14.19,1140
Naranja,9.37,1150
Durazno,280.56,872
Lima,2624.94,232
...
...
```

Discusión

¿Qué aprendimos hoy? Si creás varias funciones generadoras y las ponés "en serie" (una recibe los datos de la anterior) podés crear pipelines que controlen el flujo de datos: los procesen, modifiquen o filtren entre el primer generador y el último consumidor. Además viste lo fácil que es cambiar el comportamiento del programa cuando tenés interfases bien definidas. Por supuesto, podés empaquetar un conjunto de etapas de procesamiento en una función sola, si tiene sentido hacerlo.

9.4 Más sobre generadores

Esta sección introduce algunos temas adicionales relacionados con generadores, entre ellas: expresiones generadoras y el módulo `itertools`

Expresiones generadoras

Una expresión generadora es una lista por comprensión en su "versión generadora", que devuelve un elemento por vez.

```
>>> a = [1, 2, 3, 4]
>>> b = (2*x for x in a)
>>> b
<generator object at 0x58760>
>>> for i in b:
...     print(i, end=' ')
...
2 4 6 8
>>>
```

¿Cuáles son las diferencias entre expresiones generadoras y comprensión de listas? Bueno, las expresiones generadoras ...

- No construyen una lista
- Son construidas para ser iteradas
- Una vez consumidas, no pueden ser reutilizadas.

La sintaxis general es:

```
(<expression> for i in s if <conditional>)
```

Que puede leerse como el valor es para cada elemento `i` perteneciente a `s` siempre y cuando se cumpla.

Las podés usar como argumento de una función.

```
sum(x*x for x in a)
```

Las podés usar en lugar de cualquier iterable.

```
>>> a = [1, 2, 3, 4]
>>> b = (x*x for x in a)
>>> c = (-x for x in b)
>>> for i in c:
...     print(i, end=' ')
...
-1 -4 -9 -16
>>>
```

El uso principal de las expresiones generadoras es en código que realiza un cómputo con una serie de elementos pero sólo necesita cada elemento una única vez. Ejemplo: quitar todas las líneas de un programa que sean comentarios:

```
f = open('unarchivo.txt')
lines = (line for line in f if not line.startswith('#'))
for line in lines:
    ...
f.close()
```

Al usar generadores, tu código ejecuta más rápido y usa menos memoria. Se portan como un filtro en el flujo de datos por un pipeline.

Motivos para usar generadores

- Muchos problemas se expresan mejor en términos de iteración.
 - Recorrer una colección de items para hacer algún cómputo (buscar, reemplazar, modificar, etc.).
 - Los pipelines de procesamiento resuelven un amplio abanico de problemas.
- Son más eficientes en el uso de memoria.
 - Sólo producís valores cuando los necesitás.
 - Varias ventajas sobre construir una larga lista.
 - Pueden operar sobre datos en pipelines.
- Un generador facilita la reutilización de código.
 - Separa la propia *iteración* del código que utiliza sus resultados.
 - Podés construir tu propio conjunto de herramientas de iteración y ensamblarlas como necesites en cada caso.

El módulo `itertools`

El módulo `itertools` es una biblioteca con varias funciones útiles para construir generadores e iteradores.

```
itertools.chain(s1,s2)
itertools.count(n)
itertools.cycle(s)
itertools.dropwhile(predicate, s)
itertools.groupby(s)
itertools.ifilter(predicate, s)
itertools imap(function, s1, ... sN)
itertools.repeat(s, n)
itertools.tee(s, ncopies)
itertools.izip(s1, ... , sN)
```

Todas estas funciones procesan datos iterativamente, e implementan distintos tipos de patrones de iteración.

Si querés profundizar más en estos conceptos, te recomendamos el curso que escribió Beazley hace unos años sobre [Generadores e iteradores](#).

Ejercicios

En ejercicios anteriores escribiste código que vigilaba un archivo log esperando líneas nuevas escritas al final y las presentaba como una secuencia de filas. Este ejercicio continua aquel, de modo que vas a necesitar que `sim_mercado.py` esté ejecutándose. Acordate de pararlo cuando termines... agrega una línea por segundo y eventualmente te va a llenar tu disco rígido.

Ejercicio 9.13: Expresiones generadoras

Fijate este ejemplo de una expresión generadora:

```
>>> nums = [1, 2, 3, 4, 5]
>>> cuadrados = (x*x for x in nums)
>>> cuadrados
<generator object <genexpr> at 0x109207e60>
>>> for n in cuadrados:
...     print(n)
...
1
4
9
16
25
```

A diferencia de una definición por comprensión de listas, una expresión generadora sólo puede recorrerse una vez. Si intentás recorrerla de nuevo con otro `for`, no obtenés nada.

```
>>> for n in squares:
...     print(n)
...
>>>
```

Ejercicio 9.14: Expresiones generadoras como argumentos en funciones.

A veces es útil (y muy claro al leerlo) si pasás una expresión generadora como argumento de una función. A primera vista parece un poco raro, pero probá esto:

```
>>> nums = [1,2,3,4,5]
>>> sum([x*x for x in nums])      # una lista por comprensión
55
>>> sum(x*x for x in nums)      # una expresión generadora
55
>>>
```

En ese ejemplo, la segunda versión (que usa un generador) requeriría mucha menos memoria que si construyera toda la lista simultáneamente (si la lista fuera grande).

En tu archivo `camion.py` hiciste algunos cálculos usando comprensión de listas. Reemplazá esas expresiones por expresiones generadoras (podés entregar esta nueva versión del archivo o la anterior al final de la clase).

Ejercicio 9.15: Código simple

Las expresiones generadoras son a menudo un buen reemplazo para pequeñas funciones generadoras. Por ejemplo, en lugar de escribir una función como esta:

```
def filtrar_datos(filas, nombres):
    for fila in filas:
        if fila['nombre'] in nombres:
            yield fila
```

La podrías reemplazar con una expresión así:

```
rows = (row for row in rows if row['name'] in names)
```

Modificá este último para que use expresiones generadoras en lugar de funciones generadoras. Al final de la clase podés entregar el `ticker.py` anterior o este nuevo (¡mejor el nuevo!).

Ejercicio 9.16: Volviendo a ordenar imágenes

Te proponemos aquí que retomes el [Ejercicio 7.5](#) que tenés guardado en el archivo `listar_imgs.py`. Usá los datos que te proporciona `os.walk` y una expresión generadora para filtrar las imágenes `png` (con sus directorios correspondientes). Este filtro debería generar pares (`directorio, archivo.png`)

Más aún, opcionalmente diseñá un generador que, dada la secuencia filtrada (directorios y archivos `png`), genere ternas consistentes de:

('viejo_dir/viejo_nombre', 'nuevo_dir/nuevo_nombre', fecha_a_setear) de manera que pueda ser fácilmente usada por una función para completar las tareas del [Ejercicio 7.6](#).

9.5 Predador Presa

Los [modelos de depredación y competencia](#) forman parte de la batería de herramientas clásicas del ecólogo. Vito Volterra en Italia y Alfred Lotka en Estados Unidos fueron los precursores en este tema y crearon modelos que, con diversas modificaciones y mejoras, seguimos usando hoy en día. El modelo de Volterra para depredación comienza suponiendo la existencia de dos poblaciones de animales, una de las cuales (el depredador) se alimenta de la otra (la presa). Se supone que las dos poblaciones están formadas por individuos idénticos, mezclados en el espacio.

La gran mayoría de los modelos usualmente estudiados tienen solamente en cuenta la dinámica temporal, desatendiendo la dinámica espacial. Con la excusa de implementar una versión espacio-temporal de este modelo, proponemos un ejercicio guiado que usa fuertemente programación orientada a objetos.

Lo que sigue es un ejercicio optativo. Como decíamos, el ejercicio está muy guiado y toda la estructura del modelo está basada en objetos. La idea es recrear un mundo que imaginaremos como un valle rodeado de montañas en el que existen depredadores (que llamaremos Leones) y presas (que llamaremos Antílopes). Este valle será bidimensional, y lo representaremos por medio de una grilla rectangular que llamaremos tablero.

El modelo inicial con el que trabajaremos tiene definidas 4 etapas que determinan un ciclo:

- Etapa de movimiento: en esta etapa cada animal se desplaza a alguna posición vecina desocupada (si es que existe, sino permanece en el lugar). La decisión de a cuál desplazarse será responsabilidad del animal, que, sabiendo las disponibles, elegirá una al azar.
- Etapa de alimentación: en esta etapa los animales se alimentan. Los antílopes comerán pasto en su lugar, mientras que los leones buscarán un antílope en las posiciones vecinas y, de existir, se lo comerán. Esta acción de un león se verá reflejada en el tablero con su desplazamiento a la posición del antílope, el cual ya no será más parte de nuestro mundo.
- Etapa de reproducción: cada animal buscará en sus posiciones vecinas alguien de su misma especie. Si lo encuentra y además hay una posición vacía en el tablero se incluirá un nuevo animal (de la misma especie) en el

tablero. Nuevamente la elección de la pareja y del lugar del nuevo animal serán aleatorias. Este modelo inicial no prevee el atributo sexo, ni un límite entre la cantidad reproducciones en las que puede participar un animal por etapa.

- Cierre de ciclo: en la última etapa de un ciclo todos los animales "envejecen" en 1 unidad, aquellos que se reprodujeron y siguen en edad reproductiva vuelven a ser fértiles. Si alguno alcanzó la edad máxima de su especie se considera que ya puede retirárselo del mundo (es decir, se muere). Sucede lo mismo si, al pasar una etapa, alcanza el límite de etapas sin alimentarse, en cuyo caso muere de hambre.

Clases del modelo inicial:

El modelo inicial te lo podés bajar de [acá](#). Ya tiene definidas la clase `Animal` de la que heredan dos nuevas clases: `León` y `Antílope`. También tiene definida una clase `Tablero`. Todas estas clases se integran en una clase `Mundo`. Vamos a ver una por una estas clases y te indicaremos algunos métodos faltantes que tendrás que implementar vos y otros que te proponemos mejorar.

Animales

Esta clase está definida en el archivo `animal.py`.

La clase `Animal` será una clase abstracta, de la cual heredarán la clase `León` y la clase `Antílope`. Los métodos y atributos comunes que representen a un animal deberán estar en esta clase.

Constructor

El constructor no recibe parámetros, sólo instancia los atributos iniciales. En esta documentación se instancian algunos.

```
def __init__(self):
    super(Animal, self).__init__()
    self.edad = 0
    self.energia = self.energia_maxima # cantidad de ciclos que aguanta
sin alimentarse
```

Consultas

A continuación se muestran los métodos para obtener información del animal. Es importante notar que un animal en sí, no es ni León ni Antílope, por eso ambas preguntas devuelven `False`.

Estos métodos definen a su vez el comportamiento, no siendo únicamente una observación de las variables, sino que definen que un animal está vivo únicamente si no alcanzó la edad máxima y además no pasó el límite de tiempo que puede estar sin alimentarse.

Lo mismo sucede con `tiene_hambre` que podría modificarse para que un animal que ya se alimentó no necesite alimentarse instantáneamente. Se pueden agregar métodos como `puede_tener_cria` y cualquier otra consulta que sea pertinente para un animal.

```
def en_vida(self):
    return (self.edad <= self.edad_maxima) and self.energia > 0

def tiene_hambre(self):
    """Acá se puede poner comportamiento para que no tenga hambre todo el tiempo
    debería depender de la diferencia entre su nivel de energía y su energía máxima"""
    return True

def es_leon(self):
    return False

def es_antilope(self):
    return False
```

Modificadores

A continuación presentamos algunos métodos modificadores del objeto.

```
def pasar_un_ciclo(self):
    self.energia -= 1 # consume energía
    self.edad += 1 # envejece
    if self.edad >= 2 and self.reproducciones_pendientes > 0:
        self.es_reproductor = True #se puede reproducir

def tener_cria(self):
    """Acá se puede poner comportamiento que sucede al tener cria
    y que evita que tengas más de una cria por ciclo, ¿quizás tener_cria consume más energía que un ciclo común?"""
    pass

def alimentarse(self, animales_vecinos = None):
    self.energia = self.energia_maxima
```

```

        return None

    def reproducirse(self, vecinos, lugares_libres):
        pos = None
        if vecinos:
            pareja = random.choice(vecinos)
            if lugares_libres:
                self.tener_cria()
                pareja.tener_cria()
                pos = random.choice(lugares_libres)

    return pos

    def moverse(self, lugares_libres):
        pos = None
        if lugares_libres:
            pos = random.choice(lugares_libres)
        return pos

```

León

Al heredar de `Animal`, la clase `Leon` es pequeña:

```
class Leon(Animal):
```

Todas los métodos y atributos definidos allí están disponibles. Se hace un *override* del método `es_leon` para aclarar que en este caso Sí es un león. Y como el león es carnívoro, también se extiende el método `alimentarse` que sólo sucede cuando puede comer.

```

    def __init__(self):
        super(Leon, self).__init__()
        self.autonomia = 3
        self.edad_maxima = 5

    def es_leon(self):
        return True

    def alimentarse(self, animales_vecinos):
        # Se alimenta si puede e indica la posición del animal que se pudo
        comer
        pos = None
        if self.tiene_hambre(): # no está lleno
            presas_cercanas = [ (pos,animal) for (pos, animal) in
animales_vecinos if animal.es_antilope() ]
            if presas_cercanas: # y hay presas cerca
                super(Leon, self).alimentarse()
                (pos, animal) = random.choice(presas_cercanas)

```

```
    return pos
```

Ejemplo: Un León

Probá las siguientes instrucciones:

```
L = Leon()
L.energia
L.edad
L.es_leon()
L.es_antilope()
L.pasar_un_ciclo()
L.energia
L.edad
L.tiene_hambre()
```

Antílope

La clase `Antilope` es más pequeña aún y únicamente indica que es un Antílope. Acá se asume fuertemente que el antílope come pasto y que pasto hay siempre.

```
def __init__(self):
    self.autonomia = 8
    self.edad_maxima = 8
    super(Antilope, self).__init__()

def es_antilope(self):
    return True
```

Ejemplo: Un León y dos Antílopes

Agregá ahora dos antílopes:

```
A1 = Antilope()
A2 = Antilope()
A1.energia
A1.edad
A1.es_antilope()
```

Y hacé que el León, que ya tiene hambre, se coma alguno de los dos antílopes (aleatoriamente):

```
vecinos = [(1,A1), (2,A2)]
pos = L.alimentarse(vecinos)
if pos:
    print(f'El león se come al antílope A{pos}')
```

```
else:  
    print(f'El león no come')
```

Fijate que el método alimentarse no mata al antílope, solo devuelve su posición. Alguien más deberá ocuparse de retirarlo del mundo.

Si corrés este código varias veces, vas a ver que el León obtiene energía máxima al comer, pero sigue comiendo. Implementá adecuadamente el método tiene_hambre() de la clase Animal de manera que solo tenga hambre cuando su energía baje. Volvé a probar el código. El león debería comer solo cuando pase al menos un ciclo desde la última vez que comió.

Ejemplo: Un León y una Leona

Definamos ahora una leona `M` y hagamos que pasen unos ciclos hasta que sea reproductora.

```
M = Leon()  
M.puede_reproducir()  
M.pasar_un_ciclo()  
M.puede_reproducir()
```

Repetí hasta que `L` y `M` sean reproductores.

Luego ponelos cerca. Digamos que `M` tiene como vecinos a los dos antílopes y al león `L` de antes. Supongamos que las crías pueden ocupar algunos lugares libres en nuestro mundo (aún no definido):

```
vecinos = [L]  
lugares_libres = [4,5,6,7,8]  
L.puede_reproducir()  
M.puede_reproducir()
```

Están dadas las condiciones, que se reproduzcan:

```
pos = M.reproducirse(vecinos, lugares_libres)  
print(f'nace un nuevo leoncito en la posición {pos}')  
M.puede_reproducir()  
M.pasar_un_ciclo()  
M.puede_reproducir()
```

Vemos que el método `reproducirse` nos devuelve una de las posiciones que le dijimos que estaban libres y que los leones no pueden volverse a reproducir en el mismo ciclo pero sí en el siguiente.

Pasemos ahora a ver cómo se estructura el tablero.

El tablero

Constructor

La clase tablero tendrá un único constructor que recibirá los atributos que darán lugar al tablero del tamaño deseado:

```
def __init__(self, filas, columnas):
    super(Tablero, self).__init__()
    self.filas = filas
    self.columnas = columnas
    self.posiciones = {}
    self.n_posiciones_libres = self.filas * self.columnas
```

Esta clase tendrá 4 atributos: el número de filas, el de columnas, un diccionario donde se almacenará el contenido, y la cantidad de celdas vacías. El diccionario tendrá únicamente las posiciones con algún contenido. La cantidad de celdas vacías es un atributo que podría calcularse a partir de los 3 primeros atributos, pero se decidió tenerlo precalculado por ser un valor que se usará mucho.

Modificadores

El tablero no tiene una gran complejidad, permite ser modificado de 2 maneras: ubicar un elemento en el tablero, y retirar un elemento del tablero. Se provee además de la funcionalidad que combina estas cosas bajo el nombre de mover (cambia de lugar un elemento).

```
def ubicar(self, pos, elem):
    if not self.ocupa(pos):
        self.n_posiciones_libres -= 1
    self.posiciones[pos] = elem
    return pos in self.posiciones

def retirar(self, pos):
    self.n_posiciones_libres += 1
    return self.posiciones.pop(pos)

def mover(self, p_origen, p_destino):
    self.ubicar(p_destino, self.retirar(p_origen))
```

Getters

Además la clase tablero provee métodos para consultar sobre estado de posiciones y los valores que se encuentran en el tablero.

```
def posicion(self, pos):
    #devuelve qué hay en pos
    return self.posiciones[pos]

def ocupada(self, pos):
    return pos in self.posiciones

def libre(self, pos):
    return pos not in self.posiciones

def elementos(self):
    return list(self.posiciones.values())

def hay_posiciones_liberas(self):
    return self.n_posiciones_liberas > 0
```

Modificadores complejos

Una de las funcionalidades deseables para que se pueda usar un tablero, es la posibilidad de ubicar al azar un elemento en algún lugar del mismo (que esté libre). El tablero provee esa funcionalidad:

```
def ubicar_en_posicion_vacia(self, elem):
    if not self.hay_posiciones_liberas():
        raise RuntimeError("Estás intentando agregar algo al tablero y
está lleno")

    pos = choice(self.posiciones_liberas())
    self.ubicar(pos, elem)
```

Consultas

También ofrece consultas más complejas:

```
def posiciones_ocupadas(self):
    res = []
    for f in range(self.filas):
        for c in range(self.columnas):
            if self.ocupada((f,c)):
                res.append((f,c))

    return res

def posiciones_liberas(self):
    res = []
```

```

        for f in range(self.filas):
            for c in range(self.columnas):
                if self.libre((f,c)):
                    res.append((f,c))

    return res

def posiciones_vecinas_libre(self, pos):
    res = self.posiciones_vecinas(pos)
    res = [ p for p in res if self.libre(p) ]

    return res

def posiciones_vecinas_con_ocupantes(self, pos):
    res = self.posiciones_vecinas(pos)
    res = [ (p, self.posicion(p)) for p in res if self.ocupada(p) ]

    return res

```

Auxiliares

El método `posiciones_vecinas` es el que define la topología del terreno. En este caso, una celda tiene como vecinos a las 8 celdas con las que comparte un borde o un vértice (excepto en los bordes). Es posible modificar la dinámica del mundo modificando esta función (por ejemplo, hacerlo cilíndrico, esférico o toroidal).

```

def posiciones_vecinas(self, pos):
    desp=[(-1, -1), (-1, 0), (-1, 1), (0, 1), (1, 1), (1, 0), (1, -1),
(0, -1)]

    for i in range(len(desp)):
        f = (desp[i][0] + pos[0])
        c = (desp[i][1] + pos[1])
        desp[i] = (f, c)

    desp = [ (f,c) for f,c in desp if (0<=f and f<self.filas) and (0<=c
and c<self.columnas) ]

    return desp

```

Se pueden definir como adyacentes sólo las que comparten un borde, o las que estén a 2 celdas de distancia, incluso es posible definir que en el borde del tablero vuelva a empezar del otro lado (condiciones periódicas).

El mundo

El mundo será el encargado de hacer un tablero que usará de soporte, llenarlo de animales y ordenarlos para que se comporten de cierta forma en cada etapa que se implemente.

Constructor

En el constructor se define el tamaño del mundo, la cantidad inicial de Leones y Antílopes, tiene la opción de que imprima información de lo que va sucediendo con el parámetro `debug`.

```
def __init__(self, columnas, filas, n_leones, n_antilopes, debug=False):
    super(Mundo, self).__init__()

    self.debug = debug

    self.ciclo = 0
    self.tablero = Tablero(filas, columnas)
    self.llenar_mundo(n_leones, n_antilopes)
```

El constructor delega la tarea de llenar el tablero al método `llenar_mundo`:

```
def llenar_mundo(self, n_leones, n_antilopes):
    for _ in range(n_leones):
        if self.tablero.hay_posiciones_libres():
            print_debug("ubicando un leon", self.debug)
            self.tablero.ubicar_en_posicion_vacia(Leon())

    for _ in range(n_antilopes):
        if self.tablero.hay_posiciones_libres():
            print_debug("ubicando un Antilope", self.debug)
            self.tablero.ubicar_en_posicion_vacia(Antilope())
```

Modelado de la dinámica

En la etapa de movimiento, para cada posición ocupada del tablero, se indican cuales son sus vecinos libres y se lo manda a moverse, en caso de que el animal responda con una posición, se lo mueve a la posición indicada.

```
def etapa_movimiento(self):
    print_debug(f"Iniciando Movimiento en ciclo {self.ciclo}", self.debug)
    for p in self.tablero.posiciones_ocupadas():
        animal = self.tablero.posicion(p)

        posiciones_libres = self.tablero.posiciones_vecinas_libre(p)
        nueva_posicion = animal.moverse(posiciones_libres)
        if nueva_posicion:
```

```
        self.tablero.mover(p, nueva_posicion)
```

En la etapa de alimentación, es similar a la anterior, salvo que se indican cuales son sus vecinos ocupados y se los manda a alimentarse, en caso de que el animal responda con una posición, se lo mueve a la posición indicada.

```
def etapa_alimentacion(self):
    print_debug(f"Iniciando Alimentación en ciclo {self.ciclo}",
    self.debug)
    for p in self.tablero.posiciones_ocupadas():
        animal = self.tablero.posicion(p)
        animales_cercanos = self.tablero.posiciones_vecinas_con_ocupantes(p)
        desplazo = animal.alimentarse(animales_cercanos)
        if desplazo:
            self.tablero.ubicar(desplazo, self.tablero.retirar(p))
```

La etapa de reproducción no está implementada. Más adelante te pediremos que la implementes. Se deben pasar los animales vecinos. En este punto se debe realizar una importante decisión de modelado. ¿Quién es el encargado de que sólo se aparezcan animales de la misma especie? ¿El mundo? ¿O cada animal? En base a esto se deberá quizás modificar código existente.

```
def etapa_reproduccion(self):
    print_debug(f"Iniciando Reproducción en ciclo {self.ciclo}",
    self.debug)
    pass
```

Finalmente el método que cierra un ciclo haciendo que todos los animales cumplan años, gasten energía y retirando los que ya no están con vida.

```
def cerrar_un_ciclo(self):
    print_debug(f"Concluyendo ciclo {self.ciclo}", self.debug)
    for p in self.tablero.posiciones_ocupadas():
        animal = self.tablero.posicion(p)
        animal.pasar_un_ciclo() #envejecer, consumir alimento
        if not animal.en_vida():
            self.tablero.retirar(p)
    self.ciclo += 1
```

Los 4 métodos están para ser llamados todos juntos con:

```
def pasar_un_ciclo(self):
    self.etapa_movimiento()
    self.etapa_alimentacion()
    self.etapa_reproduccion()
    self.cerrar_un_ciclo()
```

Probando el modelo completo

Una forma para probar el funcionamiento es tener el siguiente código que corre el mundo varias veces con una pausa de tiempo para poder verlo.

```
m = Mundo(12, 6, 5, 15, debug=True)

import time
for i in range(20):
    m.pasar_un_ciclo()
    time.sleep(2)
    print(i +1)
    print(m)
```

Para empezar a explorar más en serio el modelo deberías completar los métodos que estén indicados con `pass` y otros detalles, según te indicamos en los siguientes ejercicios. Agregá además cualquier método que consideres necesario para obtener información, o modelar algún comportamiento.

Ejercicio 9.17: Etapa de reproducción

Implementá el método `etapa_reproduccion` en la clase `Mundo`.

Ejercicio 9.18: Acotando la reproducción

Implementá la lógica necesaria para que los animales pueden reproducirse únicamente una vez por año (ya sean los que inician la reproducción o los que son compañeros).

Ejercicio 9.19: Alcanzando la madurez

Implementá la lógica necesaria para que un animal sólo puede reproducirse cuando ya tiene 2 años cumplidos.

Si llegaste hasta acá, por favor guardá todo junto en un solo archivo `simulacion.py` y entregalo al finalizar la clase.

Una vez realizado esto hay diversas opciones para usarlo y expandirlo.

Extensiones del modelo

A continuación una lista no exhaustiva de las extensiones posibles al modelo, podés incorporar algunas, o todas:

- Ningún animal se alimenta más de una vez en una etapa. Modifica el método `tiene_hambre()` de la clase `Animal` para que no siempre tenga hambre.
- Si un animal se alimenta en una etapa, no necesita alimentarse más por un turno entero.
- Cuando un León ataca un Antílope, no siempre se lo come, a veces el Antílope logra escapar. Modelar esto con 3 posibles escenarios:
 - Existe una probabilidad `p` (fija), de que el León tenga éxito (con el módulo `random` se puede hacer `random.random() > p`)
 - La probabilidad es variable dependiendo de la edad del León, siendo muy baja cuando es cachorro, alta cuando es adulto, y baja de nuevo en su vejez.
 - La probabilidad es variable dependiendo de la edad del León, igual al anterior, y la edad del Antílope, siendo la probabilidad de que escape muy baja cuando es cachorro, alta cuando es adulto, y baja de nuevo en su vejez. Una forma de calcular esto es:

$$p_{comer} = p_{leon}(edadLeon) \times (1 - p_{antilope}(edadAntilope))$$

- Se puede hacer uso de los vecinos para modificar esta probabilidad, entonces si los Antílopes están en manada, que la probabilidad de comer del León sea más baja aún.
- Extender la vecindad a un radio de 2 para todos los animales.
- Extender la vecindad para distinto radio dependiendo la especie.
- Que el Antílope pueda visualizar Leones a cierta distancia (ej. radio 3), y desplazarse en otra dirección.

Exploraciones

- Analizar distintos valores para la construcción del mundo a lo largo del tiempo. ¿Es posible encontrar un equilibrio entre la cantidad de leones y antílopes?
- ¿Qué sucede con la supervivencia si se modifican los atributos propios de la clase León y Antílope de manera de que tengan mayor (o menor) autonomía, o sean más longevos?
- ¿Se detectan ciclos de mayoría de una especie y después de la otra? ¿Hay dominación?

9.6 Cierre de la novena clase

Cierre de clase

En esta clase aprendiste sobre generadores e iteradores, dos conceptos muy interesantes de Python. Viste que el mecanismo de iteración es una forma de dialogar con un objeto. Además, aprendiste los métodos que necesitás implementar para que un objeto que creaste sea iterable.

Discutimos también las ventajas de construir un programa con estos conceptos. Los programas resultan mas versátiles, modulares, y ligeros. Como con el concepto de comprensión de listas, podés usar la sintaxis de comprensión para construir un iterable: una expresión sobre la cual podés iterar sin necesidad de almacenar todos los elementos posibles en una lista ni escribir una función para calcularlos.

Por último, viste la ventaja de crear pipelines: estructuras de procesamiento de datos configurables en vuelo, altamente modulares.

Aunque no la uses de inmediato, cuando te la apropies, esta arquitectura puede cambiarte (un poquito) la vida.

Para cerrar esta clase, prepará:

- El archivo `informe.py` del [Ejercicio 9.2](#).
- El archivo `camion.py` del [Ejercicio 9.3](#) (o [Ejercicio 9.14](#)) que va a jugar en la revisión de pares.
- El archivo `vigilante.py` del [Ejercicio 9.7](#).
- El archivo `ticker.py` del [Ejercicio 9.12](#) (o del [Ejercicio 9.15](#)).
- Y, opcionalmente, el archivo `simulacion.py` del [Ejercicio 9.19](#).

Además te pedimos que completes [este formulario](#) usando tu dirección de mail como identificación. Al terminar vas a obtener un link para enviarnos tus ejercicios.

¡Nos vemos!

10. Recursión y regresión

En esta clase discutimos el concepto de recursión, proponemos algunas bases para el diseño de algoritmos recursivos y elaboramos algunos ejemplos con ellos, algunos un tanto esotéricos.

En otro orden de cosas, en la última sección damos un acercamiento práctico a las técnicas de regresión lineal y dejamos la posibilidad de profundizar más en el tema con referencias y ejercicios optativos.

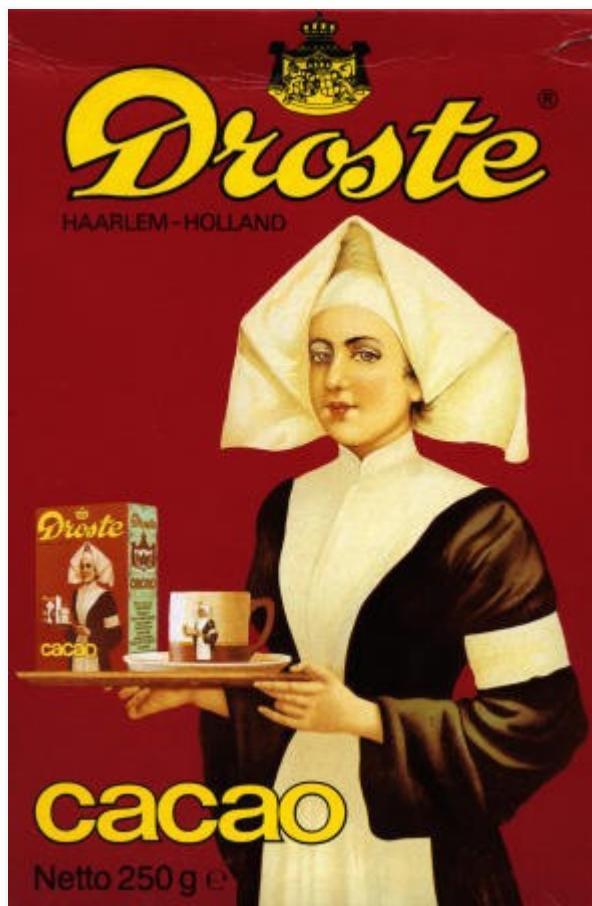
- [10.1 Intro a la Recursión](#)
- [10.2 Diseño de algoritmos recursivos](#)
- [10.3 Práctica de Recursión](#)
- [10.4 Regresión lineal](#)
- [10.5 Cierre de la clase de Recursión y Regresión](#)

10.1 Intro a la Recursión

La recursión y cómo puede ser que funcione

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

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



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

Una función recursiva matemática

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

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

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

```
def factorial(n):  
    """Precondición: n entero positivo  
    Devuelve: n!"""  
    if n == 1:  
        return 1
```

```
    return n * factorial(n - 1)
```

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

```
>>> factorial(1)
1
>>> factorial(3)
6
```

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

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

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

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

Ejercicio 10.1:

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

```
def factorial(n):
    if n == 1:
        r = 1
        return r

    f = factorial(n-1)
    r = n * f
    return r
```

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

Algoritmos recursivos y algoritmos iterativos

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

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

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

```
def factorial(n):
    """Precondición: n entero positivo
    Devuelve: n!"""
    fact = 1
    for num in range(n, 1, -1):
        fact *= num
    return fact
```

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

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

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

Un ejemplo de recursión elegante

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

La función `potencia(b, n)` que vimos cuando hablamos de invariantes en la [Sección 6.4](#), realiza n iteraciones para poder obtener el valor de b^n .

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

$$\begin{aligned} b^n &= b^{(n/2)} * b^{(n/2)} && \text{si } n \text{ es par, y} \\ b^n &= b^{((n-1)/2)} * b^{((n-1)/2)} * b && \text{si } n \text{ es impar.} \end{aligned}$$

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

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

```
def potencia(b, n):
    """Precondición: n >= 0
    Devuelve: b^n."""

    if n <= 0:
        # caso base
        return 1

    if n % 2 == 0:
        # caso n par
        p = potencia(b, n // 2)
        return p * p
    else:
        # caso n impar
        p = potencia(b, (n - 1) // 2)
        return p * p * b
```

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

```
>>> potencia(2, 10)
1024
>>> potencia(3, 3)
27
>>> potencia(5, 0)
1
```

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

```
potencia(2, 10)
  potencia(2, 5)
    potencia(2, 2)
      potencia(2, 1)
        potencia(2, 0)
          return 1
    return 1 * 1 * 2
```

```

        return 2 * 2
    return 4 * 4 * 2
    return 32 * 32
return 1024

```

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

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

```

def potencia(b, n):
    """Precondición: n >= 0
    Devuelve: b^n"""

    pila = []
    while n > 0:
        if n % 2 == 0:
            pila.append(True)
            n //= 2
        else:
            pila.append(False)
            n = (n - 1) // 2

    p = 1
    while pila:
        es_par = pila.pop()
        if es_par:
            p *= p
        else:
            p *= p * b

    return p

```

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

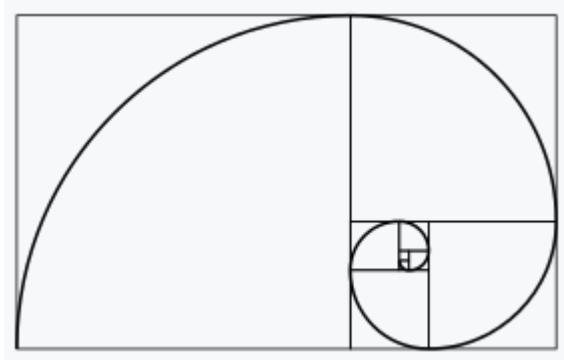
Un ejemplo de recursión poco eficiente

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

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

```
F(0) = 0  
F(1) = 1  
F(n) = F(n - 1) + F(n - 2) si n > 1
```

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

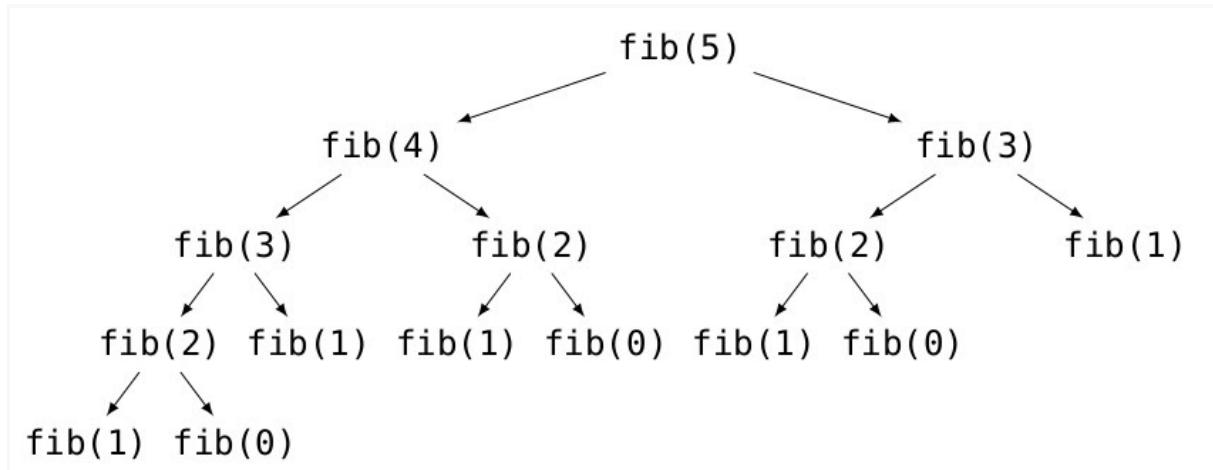


Dada la definición recursiva de la sucesión, puede resultar muy tentador escribir una función que calcule en valor de `fib(n)` de la siguiente forma:

```
def fib(n):  
    """Precondición: n >= 0.  
    Devuelve: el número de Fibonacci número n."""  
    if n == 0:  
        res = 0  
    elif n == 1:  
        res = 1  
    else:  
        res = fib(n - 1) + fib(n - 2)  
    return res
```

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

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



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

```
def fib(n):
    """Precondición: n >= 0.
       Devuelve: el número de Fibonacci número n."""
    if n == 0 or n == 1:
        return n
    ant2 = 0
    ant1 = 1
    for i in range(2, n + 1):
        fibn = ant1 + ant2
        ant2 = ant1
        ant1 = fibn
    return fibn
```

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

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

10.2 Diseño de algoritmos recursivos

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

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

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

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

Un primer diseño recursivo

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

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

Nuestro caso base será algo así como:

```
if len(lista) == 0:  
    return 0
```

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

Nuestra llamada recursiva podría ser algo así como:

```
sumar(lista[1:])
```

Lo más complejo ahora es pensar el caso general.

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

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

```
def sumar(lista):  
    """Precondición: len(lista) >= 1.  
    Devuelve: La suma de los elementos en la lista."""  
    return lista[0] + sumar2(lista[1:])
```

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

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

Finalmente nuestra primera función recursiva quedaría:

```
def sumar(lista):
    """Devuelve la suma de los elementos en la lista."""
    res = 0
    if len(lista) != 0:
        res = lista[0] + sumar(lista[1:])
    return res
```

Recursión de cola

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

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

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

```
def sumar(lista, suma = 0):
    """Devuelve la suma de los elementos en la lista."""
    res = suma
    if len(lista) != 0:
        res = sumar(lista[1:], lista[0] + suma)
    return res
```

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

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

```
def sumar(lista):
    """Devuelve la suma de los elementos en la lista."""
    suma = 0
    while lista:
        lista, suma = lista[1:], lista[0] + suma
    return suma
```

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

Modificación de la firma

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

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

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

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

```
def promediar(lista):
    if len(lista) == 1:
        res = lista[0]
    else:
```

```

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

```

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

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

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

```

def promediar_aux(lista):
    suma = lista[0]
    cantidad = 1
    if len(lista) > 1:
        suma_resto, cantidad_resto = promediar_aux(lista[1:])
        suma += suma_resto
        cantidad += cantidad_resto
    return suma, cantidad

```

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

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

```

def promediar(lista):
    """Devuelve el promedio de los elementos de una lista de números."""

    def promediar_aux(lista):
        suma = lista[0]
        cantidad = 1
        if len(lista) > 1:
            suma_resto, cantidad_resto = promediar_aux(lista[1:])
            suma += suma_resto
            cantidad += cantidad_resto
        return suma, cantidad

    suma, cantidad = promediar_aux(lista)

```

```
    return suma / cantidad
```

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

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

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

```
def potencia(b, n):
    """Precondición: n es entero
    Devuelve: b^n."""
    if n < 0:
        b = 1 / b
        n = -n
    return _potencia(b, n)
```

Limitaciones

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

Éste es un bucle infinito y corre para siempre.

```
i = 0
while i < 10:
    suma = suma + i
```

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

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

```
def inutil(n):
    return inutil(n - 1)
```

Se obtendrá un resultado como el siguiente:

```
>>> inutil(1)
  File "<stdin>", line 2, in inutil
  File "<stdin>", line 2, in inutil
  ...
  File "<stdin>", line 2, in inutil
RecursionError: maximum recursion depth exceeded
```

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

Sabías que

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

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

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

Resumen

- A medida que se realizan llamadas a funciones, el estado de cada función se almacena en la *pila de ejecución*.

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

10.3 Práctica de Recursión

Ejercicios

Ejercicio 10.2: Números triangulares

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

Fijate que este ejercicio es un caso particular de la función `sumar_enteros(desde, hasta)` que implementaste en el [Ejercicio 6.6](#). La implementación que hiciste en el primer inciso de ese ejercicio es una forma de reemplazar la recursión por un ciclo. La implementación que hiciste en el segundo inciso es mucho más eficiente.

Ejercicio 10.3: Dígitos

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

Ejercicio 10.4: Potencias

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

Ejemplos:

```
es_potencia(8, 2) -> True
es_potencia(64, 4) -> True
es_potencia(70, 10) -> False
es_potencia(1, 2) -> True
```

Ejercicio 10.5: Subcadenas

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

Ejemplo:

```
posiciones_de('Un tete a tete con Tete', 'te') -> [3, 5, 10, 12, 21]
```

Ejercicio 10.6: Paridad

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

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

Ejercicio 10.7: Máximo

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

Ejercicio 10.8: Replicar

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

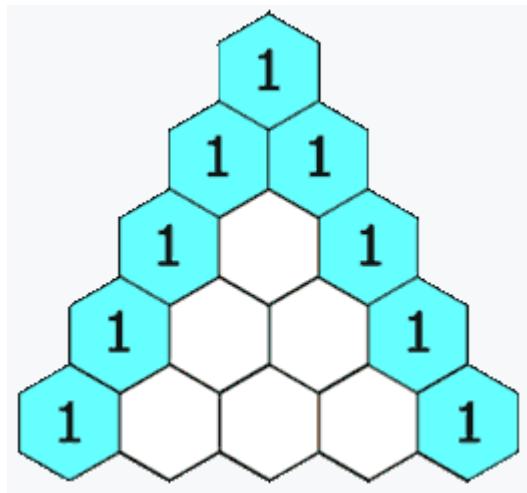
```
replicar([1, 3, 3, 7], 2) -> ([1, 1, 3, 3, 3, 7, 7])
```

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

Ejercicio 10.9: Pascal

El **triángulo de Pascal** es un arreglo triangular de números que se define de la siguiente manera: Las filas se enumeran desde $n = 0$, de arriba hacia abajo. Los valores de cada fila se enumeran desde $k = 0$ (de izquierda a derecha). Los valores

que se encuentran en los bordes del triángulo son todos unos. Cualquier otro valor se calcula sumando los dos valores contiguos de la fila de arriba.



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

Ejemplo:

```
>>> pascal(5, 2)
10
```

Ejercicio 10.10: Combinatorios

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

Ejemplo:

```
>>> combinaciones(['a', 'b', 'c'], 2)
aa ab ac ba bb bc ca cb cc
```

Ejercicio 10.11: Búsqueda binaria

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

```
def bbinaria_rec(lista, e):
    if len(lista) == 0:
        res = False
    elif len(lista) == 1:
        res = lista[0] == e
```

```

    else:
        medio = len(lista) //2

        # completar

    return res

```

Guardá tu solución en el archivo `bbin_rec.py`.

Ejercicio 10.12: Envuelviendo a Fibonacci

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

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

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

```

def fibonacci(n):
    """
    Toma un entero positivo n y
    devuelve el n-ésimo número de Fibonacci
    donde F(0) = 0 y F(1) = 1.
    """

    def fibonacci_aux(n, dict_fibo):
        """
        Calcula el n-ésimo número de Fibonacci de forma recursiva
        utilizando un diccionario para almacenar los valores ya computados.
        dict_fibo es un diccionario que guarda en la clave 'k' el valor de
        F(k)
        """

        if n in dict_fibo.keys():
            F = dict_fibo[n]
        else:
            ?? # completar
        return ?? # completar

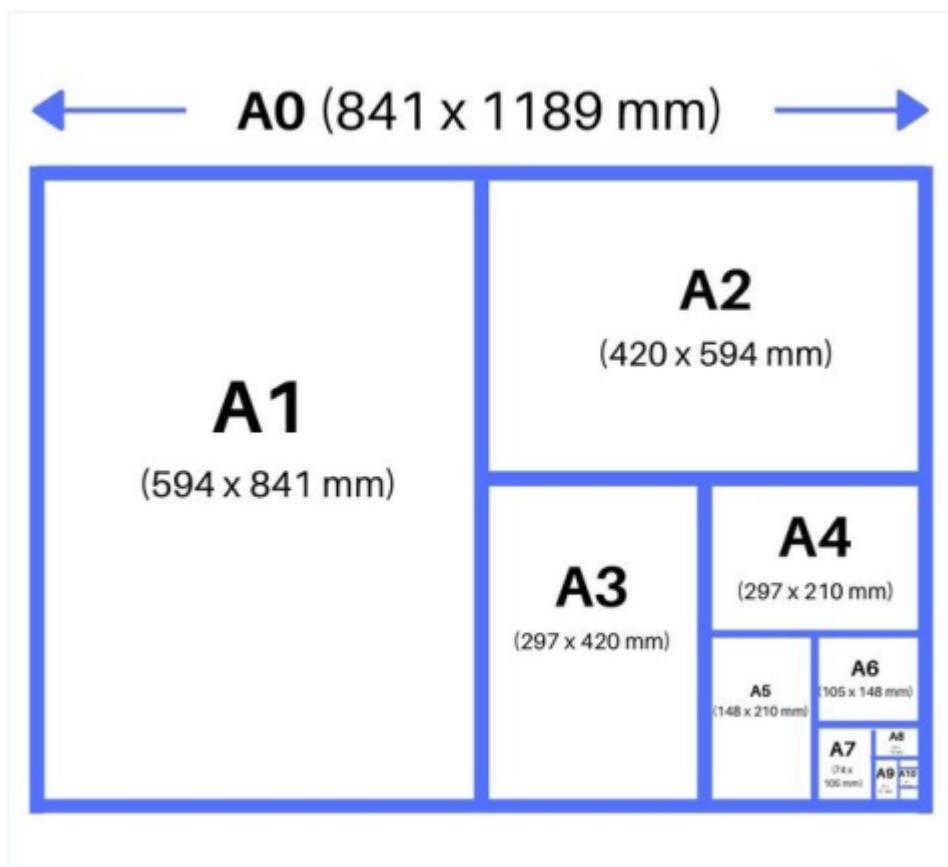
    dict_fibo = {0:0, 1:1}
    F, dict_fibo = fibonacci_aux(n, dict_fibo)
    return F

```

Guardala en el archivo `fibonacci_envuelto.py`.

Ejercicio 10.13: Hojas ISO y recursión

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



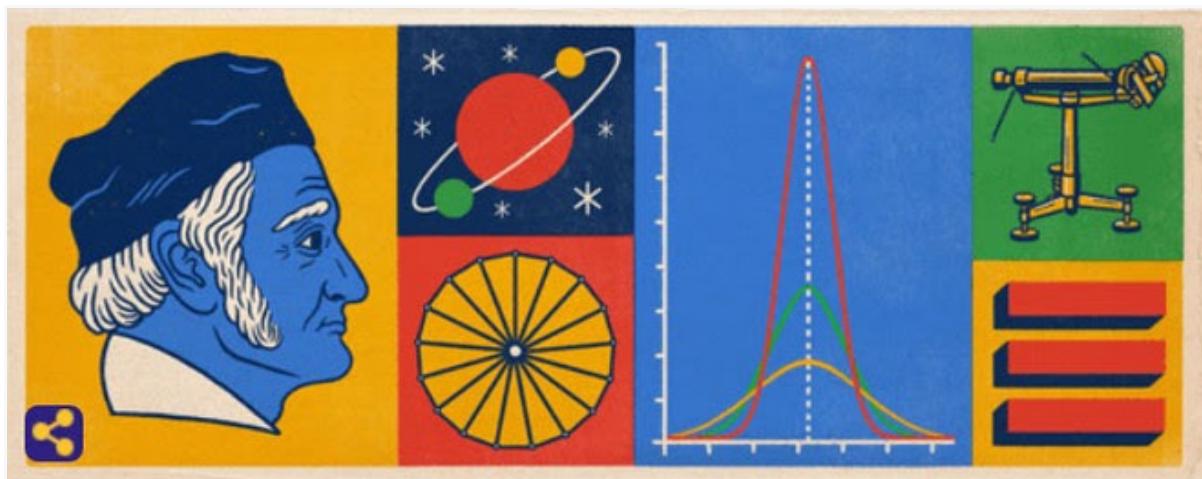
Escribí una función recursiva que para una entrada N mayor que cero, devuelva el ancho y el largo de la hoja A(N) calculada recursivamente a partir de las medidas de la hoja A(N-1), usando la hoja A0 como caso base.

Guardala en el archivo `hojas_ISO.py`.

10.4 Regresión lineal

En esta sección vamos a trabajar con regresión lineal. No es una clase con todos los fundamentos del tema, sino un acercamiento práctico a las técnicas y sus formas

de uso en Python. Para un desarrollo más profundo te recomendamos por ejemplo las notas de [Andrew Ng](#).



Regresión lineal simple

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

A partir de un conjunto de datos de tipo (x_i, y_i) , planteamos el modelo $y = a \cdot x + b$.

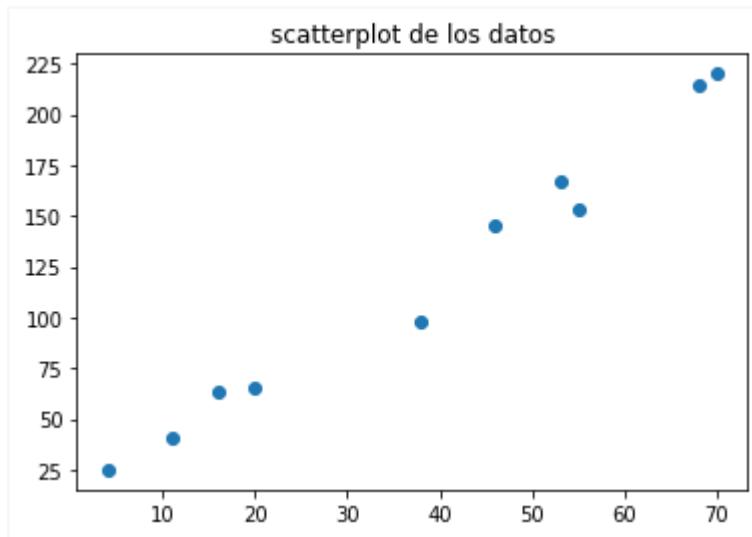
En general el modelo no va a ser exacto, es decir, no se va a cumplir que $y_i = a \cdot x_i + b$ para los valores (x_i, y_i) (salvo que estén, justamente, todos los valores sobre una línea recta). En general, decíamos, vamos a tener que $y_i = a \cdot x_i + b + r_i$ donde, los valores r_i , llamados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

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

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([55.0, 38, 68, 70, 53, 46, 11, 16, 20, 4])
y = np.array([153.0, 98, 214, 220, 167, 145, 41, 63, 65, 25])
g = plt.scatter(x = x, y = y)
```

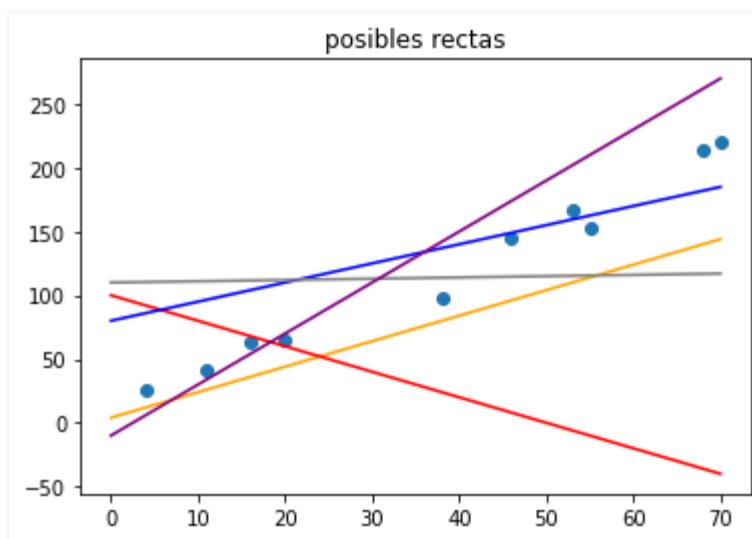
```
plt.title('scatterplot de los datos')
plt.show()
```



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

Criterio de cuadrados mínimos

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



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

$$\sum_{i=1}^n (a \cdot x_i + b - y_i)^2$$

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

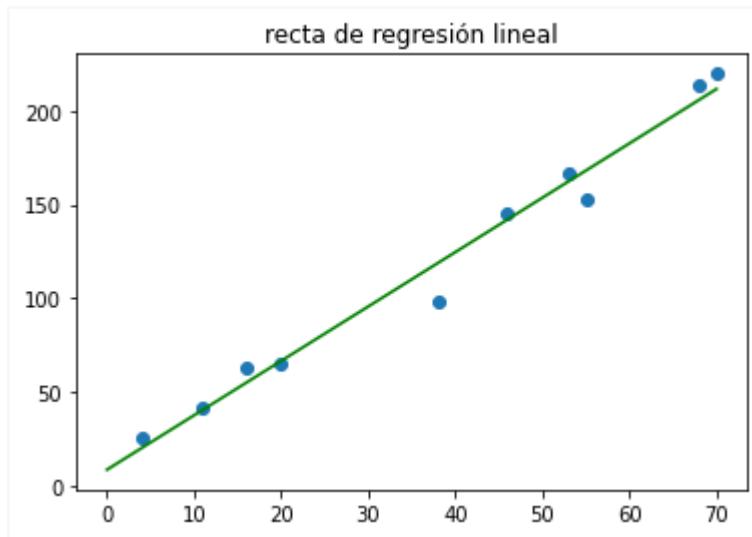
- Por un lado, minimizar el error cuadrático medio puede resolverse derivando la fórmula del error. Los que sepan algo de análisis matemático, recordarán que la derivada nos permite encontrar mínimos y que la derivada de una función cuadrática es una función lineal. Por lo tanto, encontrar la recta que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

Estas cosas se explican muy bien en [el apunte de Andrew Ng](#) que citamos antes.

Recordá que en la [Sección 4.3](#) vimos que calcular el promedio de estos errores cuadráticos es muy sencillo en numpy. También podés usar la función `mean_squared_error` del módulo `sklearn.metrics` que trae muchas métricas muy útiles.

Ejemplo: el modelo de cuadrados mínimos

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



¿Cómo se encuentran estos coeficientes?

Ajuste del modelo de cuadrados mínimos

Como buscamos el mínimo de la expresión $\sum_{i=1}^n (a \cdot x_i + b - y_i)^2$ podemos derivar respecto de los parámetros a , b e igualar a cero para despejarlos. No es una cuenta difícil. El único cero que va a tener la derivada se corresponde con un mínimo (porque la recta se puede ajustar *tan mal como uno quiera*). De esta manera se obtienen las siguientes fórmulas para el ajuste:

```
def ajuste_lineal_simple(x, y):
    a = sum(((x - x.mean()) * (y - y.mean()))) / sum(((x - x.mean()) ** 2))
    b = y.mean() - a*x.mean()
    return a, b
```

Ejemplo: datos sintéticos

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

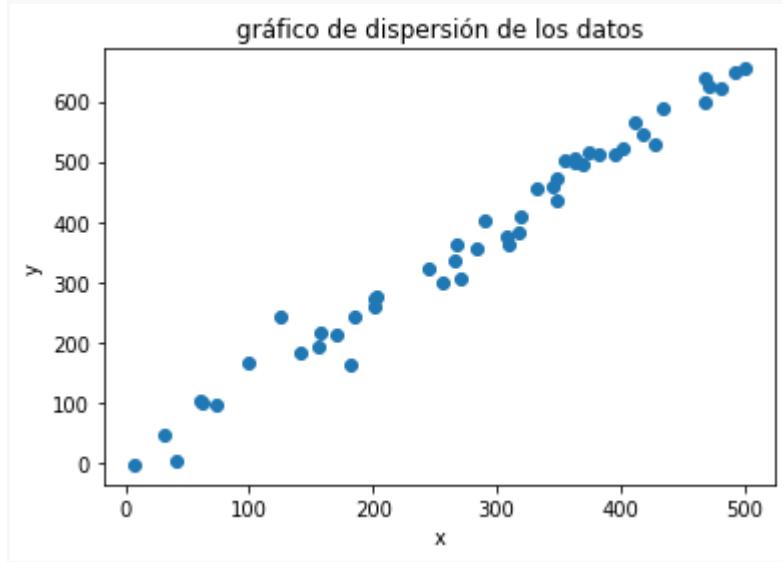
```
import numpy as np

N = 50
minx = 0
maxx = 500
x = np.random.uniform(minx, maxx, N)
r = np.random.normal(0, 25, N) # residuos simulados
y = 1.3*x + r
```

```

g = plt.scatter(x = x, y = y)
plt.title('gráfico de dispersión de los datos')
plt.xlabel('x')
plt.ylabel('y')
plt.show()

```



Ahora ajustamos con las fórmulas que vimos antes:

```

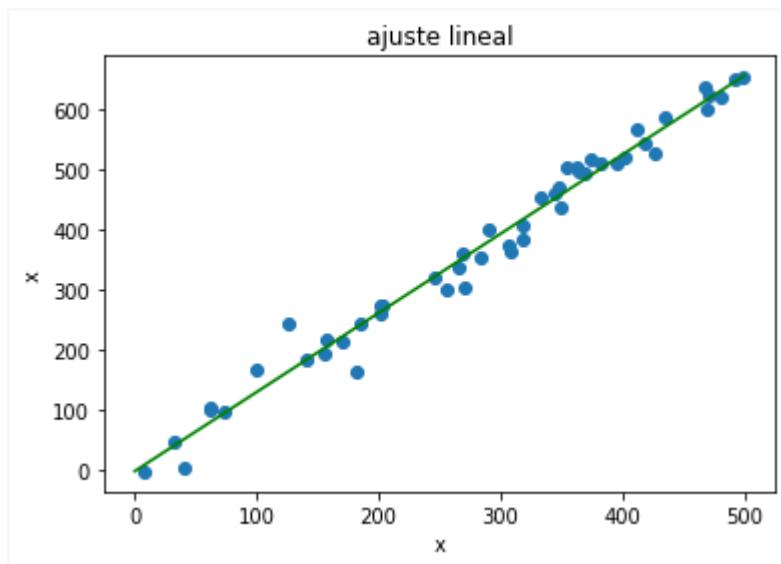
a, b = ajuste_lineal_simple(x, y)

grilla_x = np.linspace(start = minx, stop = maxx, num = 1000)
grilla_y = grilla_x*a + b

g = plt.scatter(x = x, y = y)
plt.title('y ajuste lineal')
plt.plot(grilla_x, grilla_y, c = 'green')
plt.xlabel('x')
plt.ylabel('y')

plt.show()

```



Ejercicio 10.14: precio_alquiler ~ superficie

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

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

```
superficie = np.array([150.0, 120.0, 170.0, 80.0])
alquiler = np.array([35.0, 29.6, 37.4, 21.0])
```

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

```
errores = alquiler - (a*superficie + b)
print(errores)
print("ECM:", (errores**2).mean())
```

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

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

Ejemplo: relación cuadrática

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

```

np.random.seed(3141) # semilla para fijar la aleatoriedad
N=50
indep_vars = np.random.uniform(size = N, low = 0, high = 10)
r = np.random.normal(size = N, loc = 0.0, scale = 8.0) # residuos
dep_vars = 2 + 3*indep_vars + 2*indep_vars**2 + r # relación cuadrática

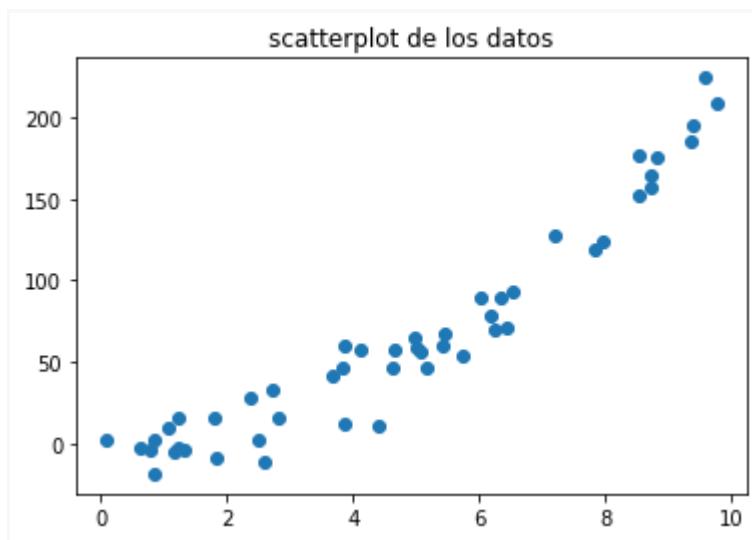
```

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

```

x = indep_vars
y = dep_vars
plt.scatter(x,y)
plt.title('scatterplot de los datos')
plt.show()

```



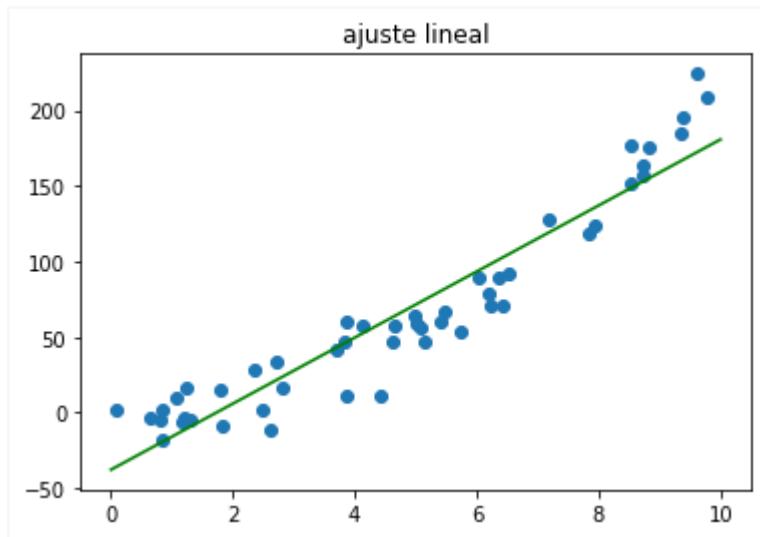
Y ajutemos un modelo lineal (notado: $y \sim x$) a estos datos.

```
a, b = ajuste_lineal_simple(x, y)
```

```

grilla_x = np.linspace(start = 0, stop = 10, num = 1000)
grilla_y = grilla_x*a + b
g = plt.scatter(x = x , y = y)
plt.title('ajuste lineal')
plt.plot(grilla_x, grilla_y, c = 'green')
plt.show()

```



Veamos cuánto vale el error cuadrático medio.

```
errores = y - (x*a + b)
print("ECM", (errores**2).mean())
```

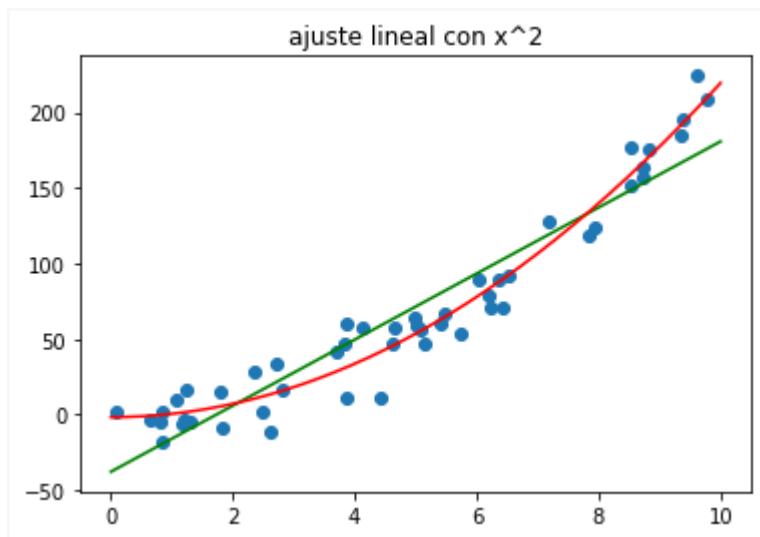
Parte optativa:

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

Ejemplo: precómputo de atributos adecuados

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

```
xc = x**2
ap, bp = ajuste_lineal_simple(xc, y)
grilla_y_p = (grilla_x**2)*ap + bp
plt.scatter(x,y)
plt.plot(grilla_x, grilla_y, c = 'green')
plt.plot(grilla_x, grilla_y_p, c = 'red')
plt.title('ajuste lineal con x^2')
plt.show()
```



Y si queremos cuantificar el error en este modelo:

```
yhat = (x**2)*ap + bp      # valores estimados
residuos = y - yhat          # diferencia entre el valor original y el
estimado
ecm = (residuos**2).mean()   # error cuadrático medio
print("ECM:", ecm)
```

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

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

Scikit-Learn

La biblioteca [scikit-learn](#) tiene herramientas muy útiles para el análisis de datos y el desarrollo de modelos de aprendizaje automático, aunque se mantiene relativamente alejada de la inferencia estadística. En particular, para regresión lineal tiene el módulo `linear_model`, y en el siguiente ejemplo mostramos cómo puede usarse. Para los que estén habituados al lenguaje R, quizás les conviene usar la biblioteca [stastmodels](#) que tiene un funcionamiento más cercano.

Al igual que el modelo de clustering que usamos en el [Ejercicio 8.19](#) de teledetección, el objeto de tipo `LinearRegression` de `sklearn.linearmodel` también

tiene un método `fit()` que permite ajustar el modelo a los datos y otro `predict()` que permite usar el modelo ajustado con nuevos datos.

Acá rehacemos el primer ejemplo que dimos ([Sección 10.4](#)), usando pandas y el módulo `linear_model`.

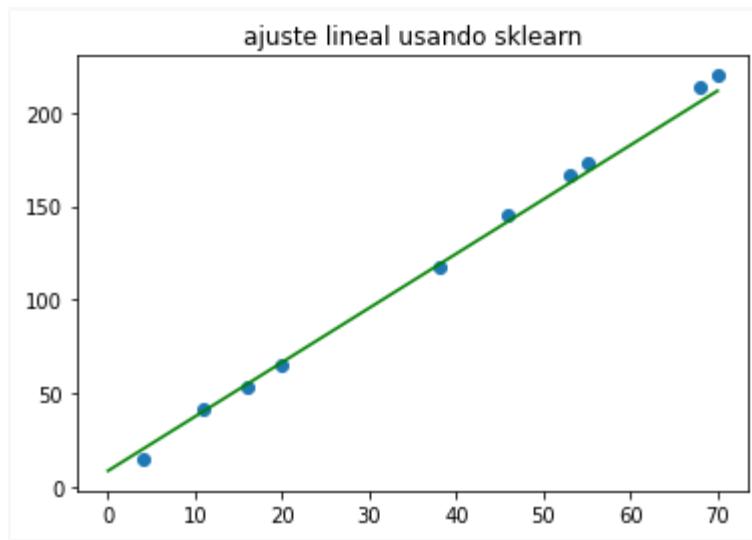
```
import pandas as pd
from sklearn import linear_model

x = np.array([55.0, 38, 68, 70, 53, 46, 11, 16, 20, 4]) # mismos datos x, y
y = np.array([153.0, 98, 214, 220, 167, 145, 41, 63, 65, 25])
datosxy = pd.DataFrame({'x': x, 'y': y}) # paso los datos a un dataframe

ajus = linear_model.LinearRegression() # llamo al modelo de regresión lineal
ajus.fit(datosxy[['x']], datosxy['y']) # ajusto el modelo

grilla_x = np.linspace(start = 0, stop = 70, num = 1000)
grilla_y = ajus.predict(grilla_x.reshape(-1,1))

datosxy.plot.scatter('x','y')
plt.title('ajuste lineal usando sklearn')
plt.plot(grilla_x, grilla_y, c = 'green')
plt.show()
```



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

Regresión Lineal Múltiple

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

$$y = b_0 + \sum_{j=1}^k b_j x_j$$

Ejemplo: superficie y antigüedad

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

```
superficie = np.array([150.0, 120.0, 170.0, 80.0])
alquiler = np.array([35.0, 29.6, 37.4, 21.0])
antiguedad = [50.0, 5.0, 25.0, 70.0]

data_deptos = pd.DataFrame({'alquiler': alquiler, 'superficie': superficie,
                            'antiguedad': antiguedad})

X = pd.concat([data_deptos.superficie, data_deptos.antiguedad], axis = 1)

ajuste_deptos = linear_model.LinearRegression()
ajuste_deptos.fit(X, data_deptos.alquiler)

errores = data_deptos.alquiler - (ajuste_deptos.predict(X))
print(errores)
print("ECM:", (errores**2).mean()) # error cuadrático medio
```

Usando los atributos `intercept_` y `coef_` de `ajuste_deptos` escribí a mano la fórmula de la regresión múltiple obtenida y respondé las siguientes preguntas respecto al modelo obtenido:

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

Ejercicio 10.15: Peso específico

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

Sabemos que el volumen de una barra de largo m es $m\text{cm}^3$, por lo que su peso debería ser $R*m$. Queremos estimar R . Utilizando la balanza, tendremos los pesos aproximados de distintas barras, con ciertos errores de medición. Si ajustamos un modelo lineal a los datos de volumen y peso aproximado vamos a tener una buena aproximación para R (la pendiente de la recta).

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

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

```
import requests  
  
import io  
  
  
enlace =  
'https://raw.githubusercontent.com/python-unsam/UNSAM_2020c2_Python/master/  
Notas/10_Recursion/longitudes_y_pesos.csv'  
  
r = requests.get(enlace).content  
  
data_lyp = pd.read_csv(io.StringIO(r.decode('utf-8')))
```

- Hacé una regresión lineal simple con `sklearn`, con variable explicativa `longitud` y variable explicada `peso` ($\text{peso} \sim \text{longitud}$).
- Estimá el peso específico del metal mirando el coeficiente obtenido.
- Graficá los datos junto con la recta del ajuste, y calculá el error cuadrático medio.
- Guardá el código en un archivo `peso_especifico.py`.

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

Ejercicio 10.16: Modelo cuadrático

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

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

```
x = indep_vars  
xc = x**2  
y = dep_vars
```

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

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

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

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

Navaja de Ockham

Al agregar covariables (regresores) a un modelo, el ajuste tiende a mejorar. Si ajusto un modelo con variables x_1, x_2, x_3 para explicar una variable y no puedo obtener un peor ajuste que si lo ajusto usando solo las variables x_1 y x_2 ya que todo modelo con las dos variables es un caso particular del modelo con las tres (simplemente hay que poner el coeficiente de la tercera variable igual a cero). Por eso, en general, al agregar variables a un modelo, su error cuadrático disminuye. Sin embargo un modelo con mejor ajuste no es *necesariamente* mejor.

El principio metodológico conocido como la [navaja de Ockham](#) nos indica que de un conjunto de variables explicativas debe seleccionarse la combinación más reducida y simple posible.

Esto ayuda a evitar fenómenos como el sobreajuste que causa [problemas muy serios y a veces graciosos](#).

Ejercicio 10.17: Modelos polinomiales para una relación cuadrática

Vimos en el [Ejercicio 10.16](#) que los datos de ese ejercicio se ajustan mejor con una regresión múltiple (usando x y x^2) que una regresión simple (basada en una sola variable). Te proponemos ahora que te fijes qué ocurre si seguimos aumentando el grado de las potencias de x que admitimos en la regresión múltiple (es decir, usar x , x^2, \dots , etc. hasta x^n). ¿Sigue bajando el error cuadrático medio? ¿Pueden considerarse *mejores* los modelos obtenidos?

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

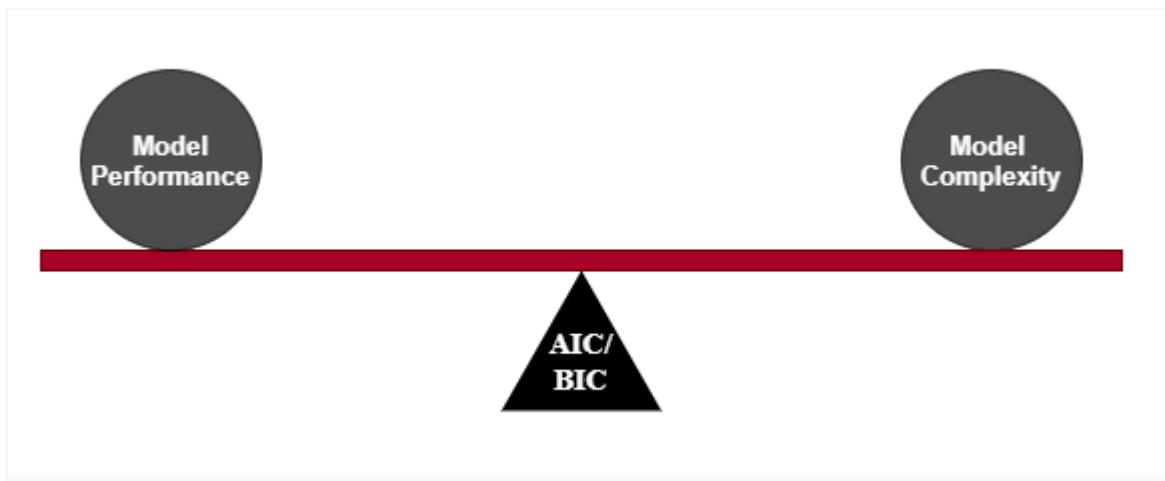
```
-----
Grado del polinomio: 1
Cantidad de parámetros: 2
ECM: 201.194
-----
Grado del polinomio: 2
Cantidad de parámetros: 3
ECM: 36.325
...
...
```

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

```
def pot(x,n):
    X=x.reshape(-1,1)
    for i in range(n-1):
        X=np.concatenate((X,(x**(i+2)).reshape(-1,1)),axis=1)
    return X
```

Ejercicio 10.18: selección de modelos

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



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

```
def AIC(k, ecm, num_params):
    '''Calcula el AIC de una regresión lineal múltiple de 'num_params' parámetros, ajustada sobre una muestra de 'k' elementos, y que da lugar a un error cuadrático medio 'ecm'.'''
    aic = k * np.log(ecm) + 2 * num_params
    return aic
```

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

```
-----
Grado del polinomio: 1
Cantidad de parámetros: 2
ECM: 201.194
AIC: 269.213
-----
Grado del polinomio: 2
Cantidad de parámetros: 3
ECM: 36.325
AIC: 185.626
...
...
```

Cuando complejizamos el modelo mejorando el error cuadrático medio, pero sin disminuir el AIC, es probable que el modelo se esté [sobreajustando](#) a los datos de entrenamiento.

Si seleccionamos el modelo ya no por su bondad de ajuste (ECM) sino buscando el mínimo AIC ¿Qué modelo queda seleccionado? Responde esta pregunta usando el comando `np.argmin()` para encontrar el grado del polinomio que minimiza el AIC y

comentá adecuadamente tu código. Guardalo en el archivo `selección_modelos.py` para entregar.

Ejercicio 10.19: Datos para la evaluación

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

```
N=50  
#genero datos para evaluar  
x_test = np.random.uniform(size = N, low = 0, high = 10)  
r_test = np.random.normal(size = N, loc = 0.0, scale = 8.0) # residuos  
y_test = 2 + 3*x_test + 2*x_test**2 + r_test # misma relación cuadrática
```

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

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

Ejercicio 10.20: Altura y diámetro de árboles.

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

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

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

- Cargá los datos en un DataFrame `data_arbolado_parques`.
- Para cada especie, seleccioná los datos correspondientes, realizá un ajuste lineal (sin ordenada al origen) de la altura dependiendo del diámetro a la altura del pecho. Realizá un scatterplot de los datos de la especie junto con la recta de regresión lineal.
- Realizá un gráfico comparando los cuatro modelos obtenidos.

- Guardá el código de este ejercicio en un archivo `ajuste_arboles.py`.

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

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

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

Ejercicio 10.21: Gráficos de ajuste lineal con Seaborn

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

```
filtro = data_arbolado_parques ['nombre_com'].isin(esp_selec)
```

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

```
g = sns.FacetGrid(datos_selec_p, col = 'nombre_com')
g.map(sns.regplot, 'diametro', 'altura_tot')
```

Observación: Nos quedaron afuera de esta clase temas importantes como sobreajuste (*Overfitting*), partición de los datos en conjuntos de entrenamiento y

evaluación, validación cruzada, presencia de datos atípicos (outliers), tests de hipótesis, selección de modelos... No era nuestra idea dar estos contenidos sino mostrar un acercamiento práctico desde Python al problema de la regresión lineal.

10.5 Cierre de la clase de Recursión y Regresión

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

Cierre de la clase

- El archivo `larenga.py` del [Ejercicio 10.9](#).
- El archivo `bbin_rec.py` del [Ejercicio 10.11](#).
- El archivo `hojas_ISO.py` del [Ejercicio 10.13](#).
- El archivo `alquiler.py` del [Ejercicio 10.14](#).
- Opcional: El archivo `seleccion_modelos.py` del [Ejercicio 10.18](#).

Como de costumbre, completá por favor [el formulario](#) asociado a la clase y adjuntá los archivos correspondientes. Ésta es la última semana en la que podés participar de la corrección de pares con el [Ejercicio 10.13](#).

11. Ordenamiento

Ordenar una lista de números es una de las tareas fundamentales que realiza un ordenador (también llamado computadora). Los algoritmos conceptualmente más sencillos tienen una complejidad computacional (en el peor caso) que crece de forma cuadrática (n^2) con la longitud, n , de la lista a ordenar. Veremos en detalle dos algoritmos sencillos: el algoritmo de selección y el de inserción y luego *divide and conquer* que lleva a la idea de merge sort...

- [11.1 Ordenamientos sencillos de listas](#)
- [11.2 Divide y reinarás](#)
- [11.3 Algoritmos de clasificación supervisada](#)

- 11.4 Cierre de la clase de Ordenamiento

11.1 Ordenamientos sencillos de listas

El problema del ordenamiento es tan fundamental que, a pesar de que Python ya lo hace con su método `sort()` por ejemplo, nos interesa discutirlo. Hay una diversidad de soluciones para ordenar listas. Vamos a empezar viendo las más sencillas de escribir (que en general suelen ser las más caras).

Ordenamiento por selección

El método de *ordenamiento por selección* se basa en la siguiente idea:

- **Paso 1.1:** Buscar el mayor de todos los elementos de la lista.

3	2	-1	5	0	2
---	---	----	---	---	---

Encuentra el valor 5 en la posición 3.

- **Paso 1.2:** Poner el mayor al final (intercambiar el que está en la última posición de la lista con el mayor encontrado).

3	2	-1	2	0	5
---	---	----	---	---	---

Intercambia el elemento de la posición 3 con el de la posición 5.

En la última posición de la lista está el mayor de todos.

- **Paso 2.1:** Buscar el mayor de todos los elementos del segmento de la lista entre la primera y la anteúltima posición.

3	2	-1	2	0	5
---	---	----	---	---	---

Encuentra el valor 3 en la posición 0.

- **Paso 2.2:** Poner el mayor al final del segmento (intercambiar el que está en la última posición del segmento –o sea anteúltima posición de la lista– con el mayor encontrado).

0	2	-1	2	3	5
---	---	----	---	---	---

Intercambia el elemento de la posición 0 con el valor de la posición 4.

En la anteúltima y última posición de la lista están los dos mayores en su posición definitiva.

...

- **Paso n:** Se termina cuando queda un único elemento sin tratar: el que está en la primera posición de la lista, y que es el menor de todos porque todos los mayores fueron reubicados.

-1	0	2	2	3	5
----	---	---	---	---	---

La lista se encuentra ordenada.

La siguiente animación muestra un algoritmo de ordenamiento por selección (que busca el menor en cada paso, en lugar del mayor):

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Una implementación en Python puede verse en el siguiente código.

```
def ord_seleccion(lista):
    """Ordena una lista de elementos según el método de selección.
       Pre: los elementos de la lista deben ser comparables.
       Post: la lista está ordenada."""

    # posición final del segmento a tratar
    n = len(lista) - 1

    # mientras haya al menos 2 elementos para ordenar
    while n > 0:
        # posición del mayor valor del segmento
        p = buscar_max(lista, 0, n)

        # intercambiar el valor que está en p con el valor que
        # está en la última posición del segmento
        lista[p], lista[n] = lista[n], lista[p]
        print("DEBUG: ", p, n, lista)

        # reducir el segmento en 1
        n = n - 1

def buscar_max(lista, a, b):
    """Devuelve la posición del máximo elemento en un segmento de
       lista de elementos comparables.
       La lista no debe ser vacía.
       a y b son las posiciones inicial y final del segmento"""

```

```

pos_max = a
for i in range(a + 1, b + 1):
    if lista[i] > lista[pos_max]:
        pos_max = i
return pos_max

```

La función principal, `ord_seleccion()` es la encargada de recorrer la lista, ubicando el mayor elemento al final del segmento y luego reduciendo el segmento a analizar.

La función `buscar_max()` busca el mayor elemento de un segmento de la lista y devuelve su posición.

A continuación, algunas ejecuciones de prueba de ese código:

```

>>> lista = [3, 2, -1, 5, 0, 2]
>>> ord_seleccion(lista)
DEBUG: 3 5 [3, 2, -1, 2, 0, 5]
DEBUG: 0 4 [0, 2, -1, 2, 3, 5]
DEBUG: 1 3 [0, 2, -1, 2, 3, 5]
DEBUG: 1 2 [0, -1, 2, 2, 3, 5]
DEBUG: 0 1 [-1, 0, 2, 2, 3, 5]
>>> lista
[-1, 0, 2, 2, 3, 5]
>>> lista = []
>>> ord_seleccion(lista)
>>> l = [1]
>>> ord_seleccion(lista)
>>> lista
[1]
>>> lista = [1, 2, 3, 4, 5]
>>> ord_seleccion(lista)
DEBUG: 4 4 [1, 2, 3, 4, 5]
DEBUG: 3 3 [1, 2, 3, 4, 5]
DEBUG: 2 2 [1, 2, 3, 4, 5]
DEBUG: 1 1 [1, 2, 3, 4, 5]

```

Podés observar que incluso cuando la lista ya está ordenada, se la recorre buscando los mayores elementos y ubicándolos en la misma posición en la que se encuentran.

Invariante en el ordenamiento por selección

Todo ordenamiento tiene un invariante que permite asegurarse de que cada paso que se toma va en la dirección de obtener una lista ordenada.

En el caso del ordenamiento por selección, el invariante es que los elementos en las posiciones desde $n + 1$ hasta el final de la lista están ordenados y son mayores que los elementos ubicados de 0 a n ; es decir que ya están en su posición definitiva.

¿Cuánto cuesta ordenar por selección?

Como se puede ver en el código de la función `buscar_max`, para buscar el máximo elemento en un segmento de lista se debe recorrer todo ese segmento, por lo que en nuestro caso debemos recorrer en el primer paso N elementos, en el segundo paso $N-1$ elementos, en el tercer paso $N-2$ elementos, etc. Cada visita a un elemento implica una cantidad constante y pequeña de comparaciones (que no depende de N). Por lo tanto tenemos que

$$T(N) \sim c * (2 + 3 + \dots + N) \sim c * N * (N+1)/2 \sim N^2$$

O sea que ordenar por selección una lista de tamaño N insume tiempo del orden de N^2 . Como ya mencionamos, este tiempo es independiente de si la lista estaba previamente ordenada o no.

En cuanto al espacio utilizado, sólo se tiene en memoria la lista que se desea ordenar y algunas variables de tamaño 1.

Ordenamiento por inserción

El método de *ordenamiento por inserción* se basa en la siguiente idea:



6 5 3 1 8 7 2 4

- **Paso 0:** Partimos de la misma lista de ejemplo utilizada para el ordenamiento por selección.

3	2	-1	5	0	2
✓					

- **Paso 1:** Considerar el segundo elemento de la lista, y ordenarlo respecto del primero, desplazándolo hasta la posición correcta, si corresponde.

2	3	-1	5	0	2
✓ ✓					

Se desplaza el valor 2 antes de 3.

- **Paso 2:** Considerar el tercer elemento de la lista, y ordenarlo respecto del primero y el segundo, desplazándolo hasta la posición correcta, si corresponde.

-1	2	3	5	0	2
✓ ✓ ✓					

Se desplaza el valor -1 antes de 2 y de 3.

- **Paso 3:** Considerar el cuarto elemento de la lista, y ordenarlo respecto del primero, el segundo y el tercero, desplazándolo hasta la posición correcta, si corresponde.

-1	2	3	5	0	2
✓ ✓ ✓ ✓					

El 5 está correctamente ubicado respecto de -1, 2 y 3 (como el segmento hasta la tercera posición está ordenado, basta con comparar con el tercer elemento del segmento para verificarlo).

...

- **Paso N-1:**

-1	0	2	3	5	2
✓ ✓ ✓ ✓ ✓					

Todos los elementos excepto el ante-último ya se encuentran ordenados.

- **Paso N:** Considerar el N -ésimo elemento de la lista, y ordenarlo respecto del segmento formado por el primero hasta el $N - 1$ -ésimo, desplazándolo hasta la posición correcta, si corresponde.

-1	0	2	2	3	5
✓ ✓ ✓ ✓ ✓ ✓					

Se desplaza el valor 2 antes de 3 y de 5.

Una posible implementación en Python de este algoritmo se incluye en el siguiente código:

```
def ord_insercion(lista):
    """Ordena una lista de elementos según el método de inserción.
    Pre: los elementos de la lista deben ser comparables.
    Post: la lista está ordenada."""
    for i in range(len(lista) - 1):
```

```

        # Si el elemento de la posición i+1 está desordenado respecto
        # al de la posición i, reubicarlo dentro del segmento [0:i]
        if lista[i + 1] < lista[i]:
            reubicar(lista, i + 1)
        print("DEBUG: ", lista)

def reubicar(lista, p):
    """Reubica al elemento que está en la posición p de la lista
    dentro del segmento [0:p-1].
    Pre: p tiene que ser una posición válida de lista."""

    v = lista[p]

    # Recorrer el segmento [0:p-1] de derecha a izquierda hasta
    # encontrar la posición j tal que lista[j-1] <= v < lista[j].
    j = p
    while j > 0 and v < lista[j - 1]:
        # Desplazar los elementos hacia la derecha, dejando lugar
        # para insertar el elemento v donde corresponda.
        lista[j] = lista[j - 1]
        j -= 1

    lista[j] = v

```

La función principal, `ord_insercion()`, recorre la lista desde el segundo elemento hasta el último, y cuando uno de estos elementos no está ordenado con respecto al anterior, llama a la función auxiliar `reubicar()`, que se encarga de colocar el elemento en la posición que le corresponde.

En la función `reubicar()` se busca la posición correcta donde debe colocarse el elemento, a la vez que se van corriendo todos los elementos un lugar a la derecha, de modo que cuando se encuentra la posición, el valor a insertar reemplaza al valor que se encontraba allí anteriormente.

En las siguientes ejecuciones puede verse que funciona correctamente.

```

>>> lista = [3, 2, -1, 5, 0, 2]
>>> ord_insercion(lista)
DEBUG:  [2, 3, -1, 5, 0, 2]
DEBUG:  [-1, 2, 3, 5, 0, 2]
DEBUG:  [-1, 2, 3, 5, 0, 2]
DEBUG:  [-1, 0, 2, 3, 5, 2]
DEBUG:  [-1, 0, 2, 2, 3, 5]
>>> lista
[-1, 0, 2, 2, 3, 5]
>>> lista = []
>>> ord_insercion(lista)
>>> lista = [1]
>>> ord_insercion(lista)

```

```

>>> lista
[1]
>>> lista = [1, 2, 3, 4, 5, 6]
>>> ord_insercion(lista)
DEBUG: [1, 2, 3, 4, 5, 6]
>>> lista
[1, 2, 3, 4, 5, 6]

```

Invariante del ordenamiento por inserción

En el ordenamiento por inserción, en cada paso se satisface que los elementos que se encuentran en el segmento de 0 a i están ordenados, de manera que agregar un nuevo elemento implica colocarlo en la posición correspondiente y el segmento seguirá ordenado.

¿Cuánto cuesta ordenar por inserción?

Del código de `ord_insercion()` se puede ver que la función principal avanza por la lista de izquierda a derecha, mientras que la función `reubicar()` cambia los elementos de lugar de derecha a izquierda.

Lo peor que le puede pasar a un elemento que está en la posición j es que deba ser ubicado al principio de la lista. Y lo peor que le puede pasar a una lista es que todos sus elementos deban ser reubicados.

Por ejemplo, en la lista `[10, 8, 6, 2, -2, -5]`, todos los elementos deben ser reubicados al principio de la lista.

En el primer paso, el segundo elemento se debe intercambiar con el primero; en el segundo paso, el tercer elemento se compara con el segundo y el primer elemento, y se ubica adelante de todo; en el tercer paso, el cuarto elemento se compara con el tercero, el segundo y el primer elemento, y se ubica adelante de todo; etc...

$$T(N) \sim c * (2 + 3 + *s + N) \sim c * N * (N+1)/2 \sim N^2$$

Es decir que ordenar por inserción una lista de tamaño N puede insumir (en el peor caso) tiempo del orden de N^2 ($O(N^2)$). En cuanto al espacio utilizado, nuevamente sólo se tiene en memoria la lista que se desea ordenar y algunas variables de tamaño 1.

Inserción en una lista ordenada

Resulta interesante observar que cuando la lista de entrada se encuentra ordenada, este algoritmo no hace ningún movimiento de elementos. Simplemente compara cada elemento con el anterior, y si es mayor sigue adelante.

Es decir que para el caso de una lista de N elementos que se encuentra ordenada, el tiempo que insume el algoritmo de inserción es:

$$T(N) \sim N.$$

Resumen

- El *ordenamiento por selección* es uno de los más sencillos, pero es bastante ineficiente: se basa en la idea de *buscar el máximo* en una secuencia, ubicarlo al final y seguir analizando la secuencia sin el último elemento.

Tiene como ventaja que hace una baja cantidad de intercambios (N), pero como desventaja que necesita una alta cantidad de comparaciones (N^2). Siempre tiene el mismo comportamiento.

- El *ordenamiento por inserción* es un algoritmo bastante intuitivo y se suele usar para ordenar en la vida real. Se basa en la idea de ir *insertando ordenadamente*: en cada paso se considera la inserción de un elemento más de secuencia y la inserción se empieza a hacer desde el final de los datos ya ordenados.

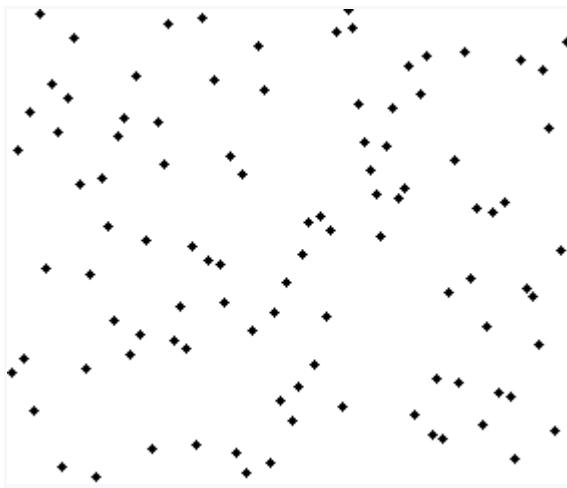
Tiene como ventaja que en el caso de tener los datos ya ordenados no hace ningún intercambio (y hace sólo $N-1$ comparaciones). En el peor caso, cuando la secuencia está invertida, se hace una gran cantidad de intercambios y comparaciones (N^2). Si bien es un algoritmo ineficiente, para secuencias cortas el tiempo de ejecución es bastante bueno.

Ejercicios

Ejercicio 11.1:

Describí los pasos del ordenamiento de la lista `[0, 9, 3, 8, 5, 3, 2, 4]` con los algoritmos de inserción y selección.

Ejercicio 11.2: burbujeo



El ordenamiento por burbujeo se basa en una idea bastante sencilla. El algoritmo compara dos elementos contiguos de la lista y, si el orden es adecuado, los deja como están, si no, los intercambia. La repetición de este *paso elemental* (una burbuja) a lo largo de la lista (recorriéndola desde el comienzo hasta el final) garantiza llevar el mayor elemento al final de la lista, pero no garantiza que el menor elemento haya quedado en el primer lugar. De hecho, el menor elemento solo se mueve un paso hacia la izquierda en una recorrida completa de la lista. Es por esto que estas recorridas se repiten sucesivas veces (¿cuántas hace falta?) de manera de garantizar que el lista quede completamente ordenada.

Como en el primer paso tenemos la garantía de que el mayor elemento quedó al final de la lista, la segunda recorrida puede evitar llegar hasta esa última posición. Así, cada recorrida es más corta que la anterior. En cada recorrida se comparan todos los pares de elementos sucesivos (en el rango correspondiente) y se intercambian si no están ordenados.

Programá una función `ord_burbujeo(lista)` que implemente este método de ordenamiento. ¿Cuántas comparaciones realiza esta función en una lista de largo n ?

Probá tu código con las siguientes listas.

```
lista_1 = [1, 2, -3, 8, 1, 5]
lista_2 = [1, 2, 3, 4, 5]
lista_3 = [0, 9, 3, 8, 5, 3, 2, 4]
lista_4 = [10, 8, 6, 2, -2, -5]
lista_5 = [2, 5, 1, 0]
```

Guardá tu solución en el archivo `burbujeo.py` comentando la complejidad del algoritmo y cómo la calculaste.

Ejercicio 11.3: ordenar a mano

Elegí dos listas de las 5 del ejercicio anterior y ordenalas a mano (con papel y lápiz) con los 3 métodos: selección, inserción y burbujeo.

Ejercicio 11.4: experimento con 3 métodos

Hacé una función `generar_lista(N)` que genere una lista aleatoria de largo `N` con números enteros del 1 al 1000 (puede haber repeticiones).

Modificá el código de las tres funciones para que cuenten cuántas comparaciones entre elementos de la lista realiza cada una. Por ejemplo, `ord_seleccion` realiza comparaciones (entre elementos de la lista) sólo cuando llama a `buscar_max(lista, a, b)` y en ese caso realiza $b-a$ comparaciones.

Realizá un experimento que genere una lista de largo `N` y la ordene con los tres métodos (burbujeo, inserción y selección).

Para `N = 10`, realizá `k = 100` repeticiones del siguiente experimento. Generar una lista de largo `N`, ordenarla con los tres métodos y guardar la cantidad de operaciones. Al final, debe imprimir el promedio de comparaciones realizado por cada método.

Cuidado: usá las mismas listas para los tres métodos así la comparación es justa.

Ejercicio 11.5: comparar métodos gráficamente

Vamos a tratar de comparar visualmente la cantidad de comparaciones que hacen estos algoritmos para diferentes largos de listas. Hacé un programa `comparaciones_ordenamiento.py` que para `N` entre 1 y 256 genere una lista de largo `N` con números enteros del 1 al 1000, calcule la cantidad de comparaciones realizadas por cada método y guarde estos resultados en tres vectores de largo 256: `comparaciones_seleccion`, `comparaciones_insercion` y `comparaciones_burbujeo`.

Graficá estos tres vectores. Si las curvas se superponen, graficá una de ellas con línea punteada para poder verlas bien. ¿Cómo dirías que crece la complejidad de estos métodos? ¿Para cuáles depende de la lista a ordenar y para cuáles solamente depende del largo de la lista?

Guardá `comparaciones_ordenamiento.py` para seguir trabajando sobre él y para entregarlo.

¿Se te ocurre un algoritmo de ordenamiento que sea sustancialmente mejor que estos? Ese será el tema de la próxima sección.

Extra: ¿Las curvas de complejidad quedaron suaves? ¿Se te ocurre cómo hacer para suavizarlas?

11.2 Divide y reinarás

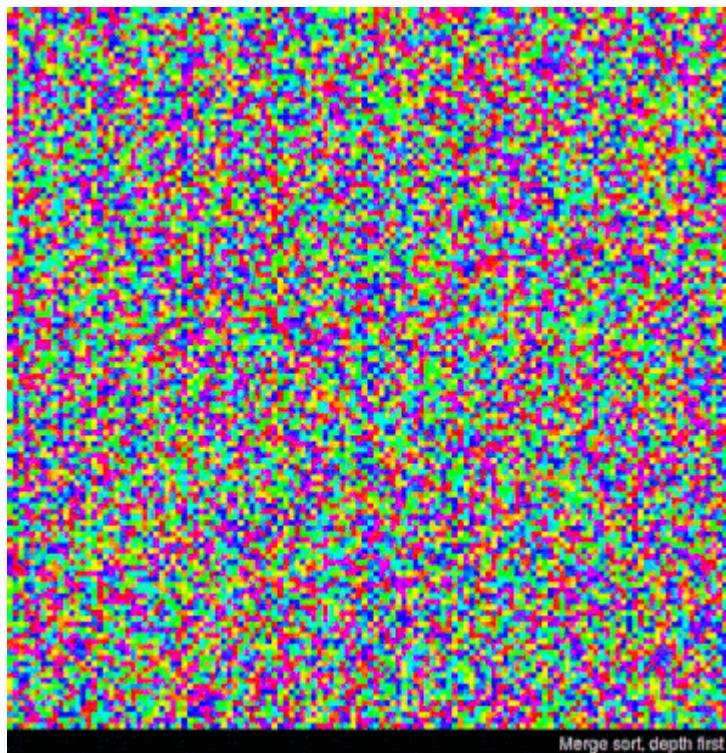
El problema del ordenamiento es un problema fundamental y hay [muchísimos algoritmos que lo resuelven](#). Los métodos de ordenamiento vistos en la sección anterior eran métodos iterativos cuyo tiempo de ejecución era cuadrático.

Veremos ahora el merge sort que es un algoritmo un poco más complejo conceptualmente pero menos complejo computacionalmente. El algoritmo está basado en una idea muy fecunda en el diseño de algoritmos eficientes que se denomina divide y reinarás (ó *divide and conquer* en inglés).

Divide y reinarás es un paradigma de diseño de algoritmos recursivos que trabaja partiendo (dividiendo) el problema original en subproblemas del mismo tipo pero más sencillos de resolver. Las soluciones de estos subproblemas luego se combinan para obtener una solución del problema original.

La correctitud de los algoritmos de este tipo suele probarse utilizando la inducción matemática y el cálculo de su complejidad involucra la resolución de ecuaciones de recurrencia cuyos detalles escapan el alcance de este curso.

El algoritmo *merge sort* (u ordenamiento por mezcla)



Merge sort, depth first

El *merge sort* se basa en la siguiente idea:

- Si la lista es pequeña (vacía o de tamaño 1) ya está ordenada y no hay nada que hacer. De lo contrario hacer lo siguiente:
- Dividir la lista al medio, formando dos sublistas de (aproximadamente) el mismo tamaño cada una.
- Ordenar cada una de esas dos sublistas (usando este mismo método).
- Una vez que se ordenaron ambas sublistas, intercalarlas (mergearlas) de manera ordenada.

Por ejemplo, si la lista original es [6, 7, -1, 0, 5, 2, 3, 8] deberemos ordenar recursivamente [6, 7, -1, 0] y [5, 2, 3, 8] con lo cual obtendremos [-1, 0, 6, 7] y [2, 3, 5, 8]. Si intercalamos ordenadamente las dos listas ordenadas obtenemos la solución buscada: [-1, 0, 2, 3, 5, 6, 7, 8].

Veamos otro ejemplo con un gif animado:

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

Diseñemos la función `merge_sort(lista)`:

- Si lista es pequeña (vacía o de tamaño 1) ya está ordenada y no hay nada que hacer. Se devuelve lista original.
- De lo contrario:
 - `medio = len(lista) // 2`
 - `izq = merge_sort(lista[:medio])`
 - `der = merge_sort(lista[medio:])`
 - Se devuelve `merge(izq, der)`.

Falta sólo diseñar la función `merge`: dadas dos listas ordenadas debe obtener una nueva lista que resulte de intercalar a ambas de manera ordenada:

- Utilizaremos dos índices, `i` y `j`, para recorrer cada una de las dos listas.
- Utilizaremos una tercera lista, `resultado`, donde almacenaremos el resultado.
- Mientras `i` sea menor que el largo de `lista1` y `j` menor que el largo de `lista2`, significa que hay elementos para comparar en ambas listas.
- Si el menor es el de `lista1`:
 - Agregar el elemento `lista1[i]` al final de la lista `resultado`.
 - Incrementar el índice `i`.
- de lo contrario:
 - Agregar el elemento `lista2[j]` al final de la lista `resultado`.
 - Incrementar el índice `j`.
- Una vez que una de las dos listas se termina, simplemente hay que agregar todo lo que queda en la otra al final de la lista `resultado`.

El código resultante del diseño de ambas funciones puede verse a continuación:

```
import random

def merge_sort(lista):
    """Ordena lista mediante el método merge sort.
    Pre: lista debe contener elementos comparables.
    Devuelve: una nueva lista ordenada."""
    if len(lista) < 2:
```

```

        lista_nueva = lista
    else:
        medio = len(lista) // 2
        izq = merge_sort(lista[:medio])
        der = merge_sort(lista[medio:])
        lista_nueva = merge(izq, der)
    return lista_nueva

def merge(list1, list2):
    """Intercala los elementos de list1 y list2 de forma ordenada.
       Pre: list1 y list2 deben estar ordenadas.
       Devuelve: una lista con los elementos de list1 y list2."""
    i, j = 0, 0
    resultado = []

    while(i < len(list1) and j < len(list2)):
        if (list1[i] < list2[j]):
            resultado.append(list1[i])
            i += 1
        else:
            resultado.append(list2[j])
            j += 1

    # Agregar lo que falta de una lista
    resultado += list1[i:]
    resultado += list2[j:]

    return resultado

```

El método divide y reinarás que hemos usado para resolver el problema de ordenar una lista puede aplicarse también en otras situaciones. Hace falta que sea posible resolver el problema partiéndolo en varios subproblemas de tamaño menor, resolver cada uno de esos subproblemas por separado aplicando la misma técnica (en nuestro caso ordenar por mezcla cada una de las dos sublistas), y finalmente juntar estas soluciones parciales en una solución completa del problema mayor (en nuestro caso la intercalación ordenada de las dos sublistas ordenadas).

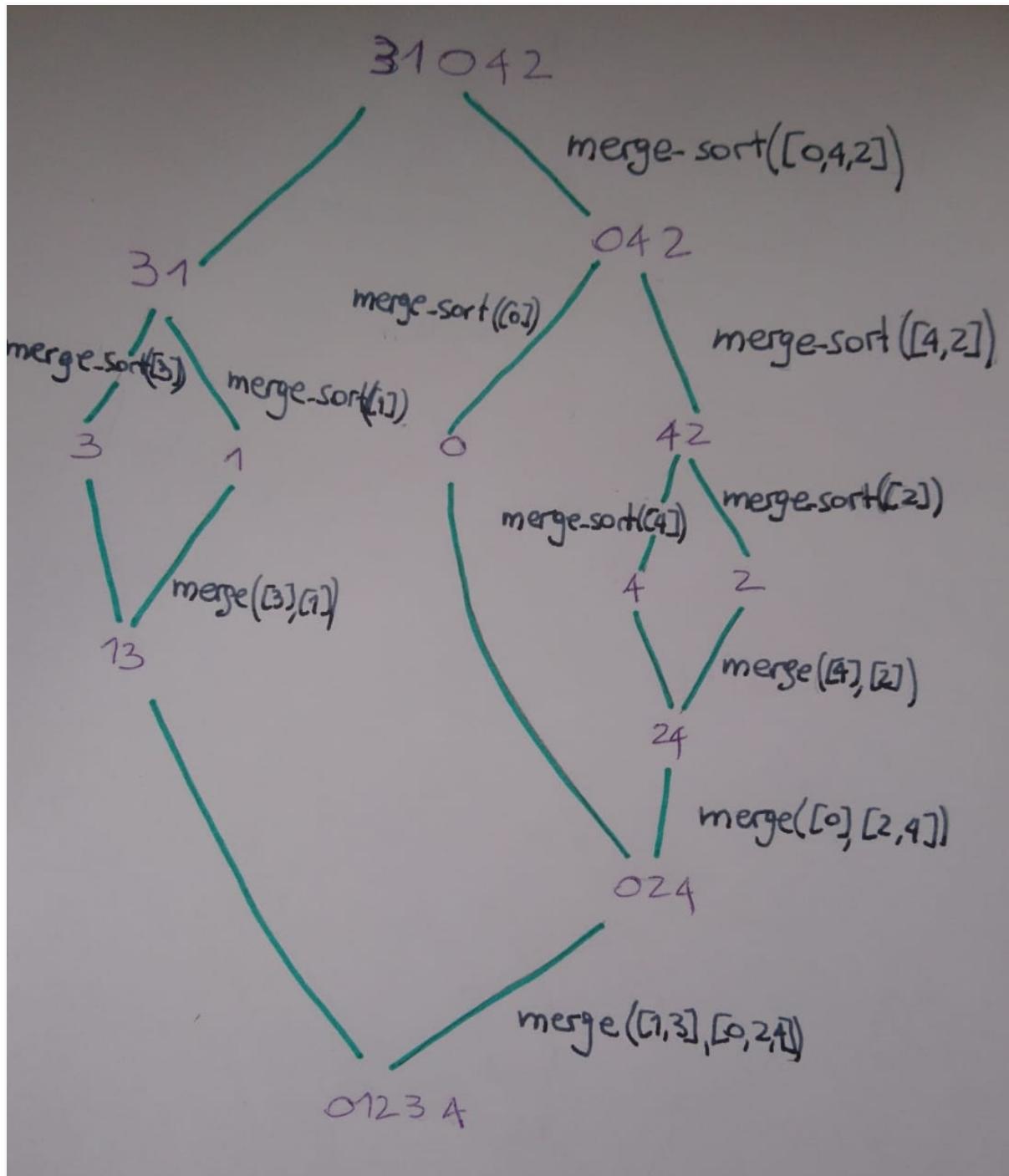
Como siempre sucede con las soluciones recursivas, debemos encontrar un caso base en el cual no se aplica la llamada recursiva (en nuestro caso: si la lista tiene largo cero o uno, ya está ordenada y no hay nada que hacer). Además debemos asegurar que siempre se alcanza el caso base, y en nuestro caso aseguramos eso porque, si no estamos en el caso base, la lista se divide en mitades decrementando su longitud.

El método divide y reinarás es fecundo y ha dado lugar a algoritmos muy eficientes para tareas muy disímiles como multiplicar matrices, calcular la transformada de Fourier o realizar análisis sintácticos (parsear).

Ejemplo: Árbol de recursión

Para representar gráficamente las llamadas recursivas de la función `merge_sort()` podemos hacer un árbol similar al que mostramos en la [Sección 10.1](#) para la sucesión de Fibonacci. Esta vez, como queremos mostrar no sólo las llamadas sino también lo que devuelve cada llamada, espejaremos el árbol obteniendo un grafo.

Acá mostramos el árbol de recursión de `merge_sort([3, 1, 0, 4, 2])`.



¿Cuánto cuesta el Merge sort?

Supongamos que tenemos que ordenar una lista de longitud N . Observamos lo siguiente:

- Para intercalar dos listas de longitud $N/2$ hace falta recorrer ambas listas que en total tienen N elementos. La cantidad de operaciones resulta proporcional a N . Llamemos $a * N$ a ese tiempo.
- Si llamamos $T(N)$ al tiempo que tarda el algoritmo en ordenar una lista de longitud N , vemos que $T(N) = 2 * T(N/2) + a * N$.
- Además, cuando la lista es pequeña, la operación es de tiempo constante:
 $T(1) = T(0) = b$.

Para simplificar la cuenta vamos a suponer que $N = 2^k$.

$$\begin{aligned}T(N) &= T(2^k) = 2 * T(2^{k-1}) + a * 2^k \\&= 2 * (2 * T(2^{k-2}) + a * 2^{k-1}) + a * 2^k \\&= 2^2 * T(2^{k-2}) + a * 2^k + a * 2^k \\&\quad \cdot \\&\quad \cdot \\&\quad \cdot \\&= 2^i * T(2^{k-i}) + i * a * 2^k \\&\quad \cdot \\&\quad \cdot \\&\quad \cdot \\&= 2^k * T(1) + k * a * 2^k \\&= b * 2^k + k * a * 2^k\end{aligned}$$

Pero si $N = 2^k$ entonces $k = \log_2(N)$, y por lo tanto hemos demostrado que:

$$T(N) = b * N + a * N * \log_2(N).$$

Como lo que nos interesa es aproximar el valor, diremos (despreciando el término de menor orden) que

$$T(N) \sim N * \log_2(N)$$

Dado que $\log_2(N)$ es un número mucho más pequeño que N , hemos mostrado entonces que el merge sort se porta mucho mejor (es decir, es más eficiente) que los tres métodos de ordenamiento que discutimos en la sección anterior (que eran cuadráticos).

Si analizamos el espacio que consume, vemos que a cada paso la función `merge` genera una nueva lista cuya longitud es la suma de los tamaños de las dos listas, por lo que `merge_sort` usa el doble de espacio que la lista de entrada.

Resumen

- Los métodos de ordenamiento de selección, inserción y burbujeo presentados en la sección anterior son métodos conceptualmente sencillos pero costosos en cantidad de operaciones (intercambios y/o comparaciones). Sin embargo, es posible conseguir métodos más eficientes usando algoritmos recursivos.
- El algoritmo *merge sort* consiste en dividir la lista a ordenar hasta que tenga 1 ó 0 elementos y luego combinar la lista de forma ordenada. De esta manera se logra un tiempo proporcional a $N * \log_2(N)$.

Ejercicios:

Ejercicio 11.6:

Ordená la lista [6, 0, 3, 2, 5, 7, 4, 1] usando el método merge sort. Dibujá el árbol de recursión explicando las llamadas que se hacen en cada paso, y el orden en el que se realizan, como mostramos más arriba para la lista [3, 1, 0, 4, 2].

Ejercicio 11.7:

Modificá la función `merge_sort` para que también devuelva la cantidad de comparaciones hechas. Rehacé el último ejercicio de la sección anterior ([Ejercicio 11.5](#)) incorporando el merge sort a la comparación y al gráfico. Describí con tus palabras qué observas.

Guardá el archivo `comparaciones_ordenamiento.py` con estas modificaciones, para entregarlo.

El módulo `timeit`

Hay casos en que contar la cantidad de operaciones que un algoritmo realiza se vuelve muy engorroso. Una alternativa simple aunque menos exacta es medir su tiempo de ejecución para problemas de distintos tamaños y estimar el orden del algoritmo a partir del cambio en el tiempo de ejecución al cambiar el tamaño del problema.

Existe un módulo llamado `timeit` que permite medir tiempos de ejecución de código Python.

El comando `timeit()` del modulo `timeit` devuelve, en segundos, el tiempo de ejecución total para el comando y número de repeticiones especificadas. El código a ejecutar y la cantidad de ejecuciones se pasan como parámetros. En esta sección, vamos a usarlo para comparar algoritmos de ordenamiento.

En el siguiente ejemplo se le pide a Python una demora de 1 segundo (`sleep(1)`) y se le pide a `timeit()` que devuelva el tiempo de ejecución de ese comando, ejecutado una sola vez). Probálo varias veces:

```
In [1]: import time
In [2]: import timeit as tt
In [3]: tt.timeit('time.sleep(1)', number = 1)
Out[3]: 1.0010360410087742
```

Notarás que el tiempo de ejecución está muy cerca del esperado (1 segundo), pero no es exactamente ese valor y además hay cierta variación entre repeticiones.

Ahora evaluemos la siguiente expresión, que concatena en un string los primeros 100 números enteros. Ejecútala por lo menos diez veces y mirá como varía la salida:

```
In [4]: tt.timeit('"-".join(str(n) for n in range(100))', number = 1)
Out[4]: 6.670000296551734e-05
```

Si medimos un proceso que tarda poco tiempo como en el último ejemplo, obtendremos resultados con una variabilidad de magnitud similar a la duración del proceso. Si queremos comparar duraciones relativas es mejor medir procesos largos o, si se trata de un proceso corto, repetirlo muchas veces. Usando `timeit()` podemos hacer esto cambiando el parametro `number`.

Probá lo siguiente:

```
In [5]: tt.timeit('"-".join(str(n) for n in range(100))', number = 10000)
Out[5]: 0.3018611848820001
```

Ahora comparemos la duración de ese código python con la duración de otras expresiones que dan el mismo resultado. Ejecutá cada una varias veces:

```
In [6]: tt.timeit('"-".join(str(n) for n in range(100))', number = 10000)
Out[6]: 0.3018611848820001
In [7]: tt.timeit('"-".join([str(n) for n in range(100)])', number = 10000)
Out[7]: 0.2727368790656328
In [8]: tt.timeit('"-".join(map(str, range(100)))', number = 10000)
Out[8]: 0.23702679807320237
```

Ejemplo: evaluar el método de selección con timeit.

Queremos evaluar cuánto tarda del método de selección dependiendo de la longitud de la lista de entrada.

Para eso, primero generamos listas de longitudes entre 1 y 256.

```
listas = []
for N in range(1, 256):
    listas.append(generar_lista(N))
```

Luego, definimos una función `experimento_timeit_seleccion(listas, num)` que realiza un experimento usando `timeit` para evaluar el método de selección (repitiendo `num` veces) con las listas pasadas como entrada, y devuelve los tiempos de ejecución para cada lista en un vector.

```
def experimento_timeit_seleccion(listas, num):
    """
    Realiza un experimento usando timeit para evaluar el método
    de selección para ordenamiento de listas
    con las listas pasadas como entrada
    y devuelve los tiempos de ejecución para cada lista
    en un vector.

    El parámetro 'listas' debe ser una lista de listas.
    El parámetro 'num' indica la cantidad de repeticiones a ejecutar el
    método para cada lista.

    """
    tiempos_seleccion = []

    global lista

    for lista in listas:

        # evalúo el método de selección
        # en una copia nueva para cada iteración
        tiempo_seleccion = tt.timeit('ord_seleccion(lista.copy())', number
= num, globals = globals())

        # guardo el resultado
        tiempos_seleccion.append(tiempo_seleccion)

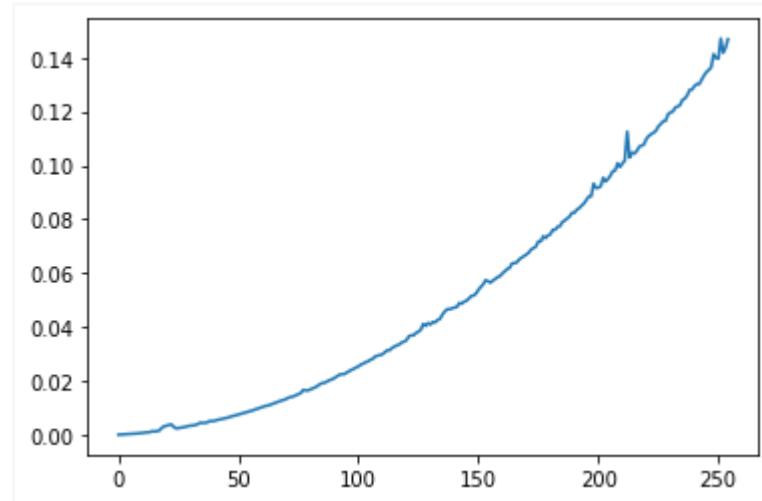
    # paso los tiempos a arrays
    tiempos_seleccion = np.array(tiempos_seleccion)

    return tiempos_seleccion
```

El parámetro `globals = globals()` permite a `timeit()` acceder a las variables y funciones definidas en el namespace global en que se está ejecutando. El comando `global lista` hace que la variable `lista` que va recorriendo la lista de listas, sea *global* y por lo tanto accesible a `timeit()`.

Y ahora realizamos el experimento y lo graficamos.

```
tiempos_seleccion = experimento_timeit_seleccion(listas, 100)
plt.plot(tiempos_seleccion)
```



Ejercicio 11.8:

La idea de este ejercicio es, nuevamente, comparar los algoritmos de ordenamiento que vimos hasta ahora pero usando `timeit()` en lugar de contando a mano la cantidad de operaciones.

- Juntá en el archivo `time_ordenamiento.py` los métodos de búsqueda del [Ejercicio 11.7](#).
- Antes de empezar el experimento, eliminá de las funciones a medir todo código no esencial, en particular los prints para debug. Consumen tiempo y no son parte del algoritmo. También eliminá las cuentas de comparaciones, que ahora no son necesarias.
- Escribí un experimento que, tal como hiciste en el [Ejercicio 11.5](#), para `N` entre 1 y 256 genere una lista de largo `N` con números enteros del 1 al 1000, calcule el tiempo que tarda cada método en ordenar la lista y guarde estos resultados en vectores de largo 256.
- En este caso, vas a tener que generar y guardar todas las listas a ser utilizadas antes de correr el experimento, para poder usar las mismas para evaluar cada método. Definí para eso una función `generar_listas(Nmax)` que genere una lista de listas, una de cada longitud entre 1 y `Nmax`, con valores aleatorios entre 1 y 1000.

- Asegurate de evaluar todos los métodos de ordenamiento con las mismas listas (siempre usá copias para no reordenar listas ya ordenadas) y guardar esta información para poder mostrarla o usarla.
- Graficá los datos de tiempos de ejecución en función de longitudes de la lista. ¿Coinciden las curvas con lo que habías predicho estimando el número de operaciones?
- Guardá el archivo `time_ordenamiento.py` para entregarlo.

Ejercicio 11.9:

Opcional: Escribí una función `merge3sort` que funcione igual que el merge sort pero en lugar de dividir la lista de entrada en dos partes, la divide en tres partes. Deberás escribir la función `merge3sort(list1, lista2, lista3)`.

Probá tu función en las siguientes listas:

```
unalista = [1, 4, 3, 1, 7, 5]
otralista = [7, 6, 5, 4, 3, 2, 1, 0]
```

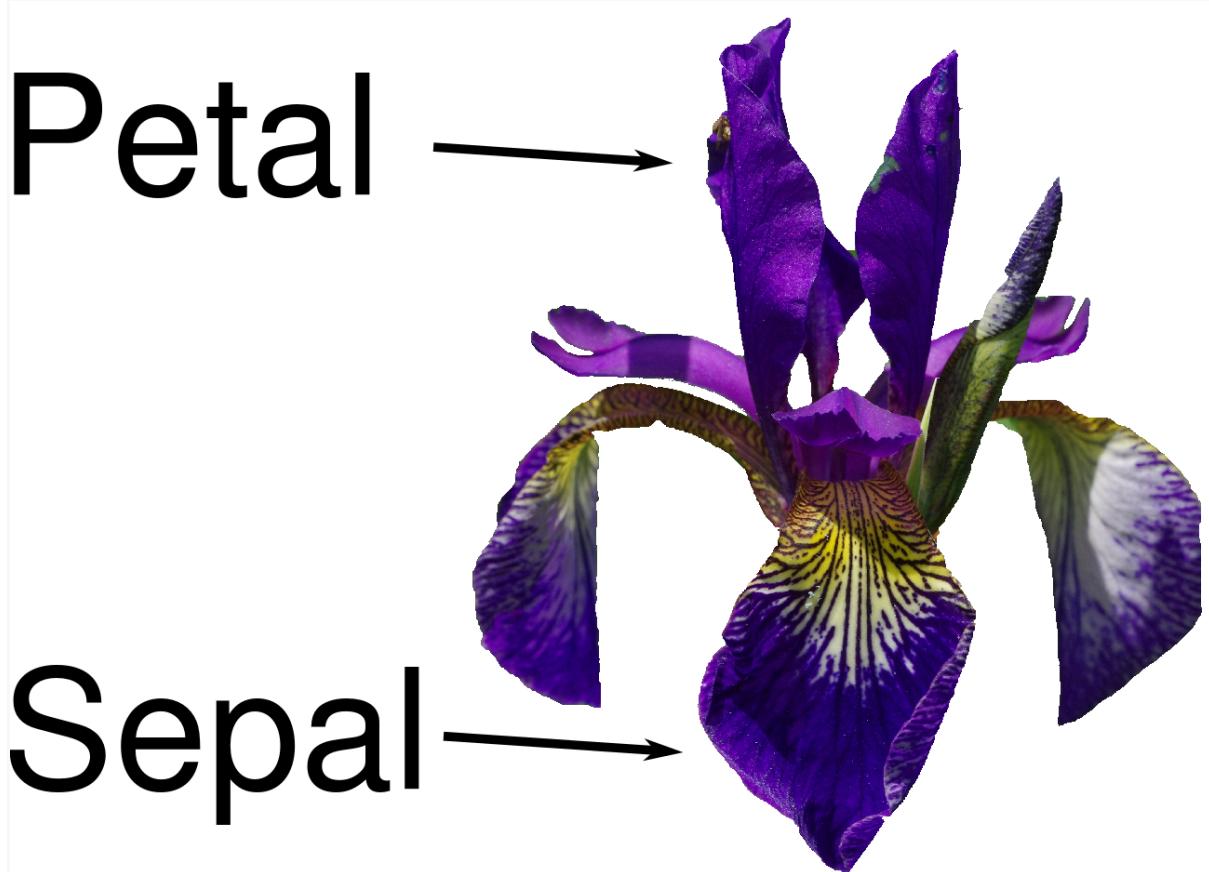
¿Cómo te parece que se va a comportar este método con respecto al merge sort original? Agregá este nuevo método a la comparación del ejercicio anterior.

11.3 Algoritmos de clasificación supervisada

En esta sección veremos un algoritmo de clasificación. Un problema de clasificación es un problema en el que tenemos algunas clases fijas (en nuestro ejemplo serán tres tipos de flores) y algunos atributos (medidas de los pétalos y sépalos, en nuestro ejemplo) a partir de los cuales queremos *inferir* la clase. Típicamente el algoritmo de clasificación se *entrena* con alguna parte de los datos para que *aprenda* y luego se *evalúa* cuán bien aprendió con el resto de los datos. Para esto hace falta tener un conjunto de datos *etiquetados* (es decir, con la clase bien definida). Luego, si funciona bien, el algoritmo podrá usarse para etiquetar nuevos datos de los que no se conoce la clase.

En esta sección nos concentraremos en el entrenamiento y la evaluación de los algoritmos.

Trabajaremos con la librería sklearn de python que está diseñada para realizar tareas de aprendizaje automático. La misma trae algunos conjuntos de datos de ejemplo. Trabajaremos con el clásico ejemplo de Clasificación de Especies de flores Iris según medidas del pétalo y el sépalo.



Veamos los datos

```
from sklearn.datasets import load_iris  
  
iris_dataset = load_iris()
```

Este dataset trae una serie de datos medidos de los pétalos y sépalos de 150 flores Iris y su clasificación en tres especies (setosa, versicolor y virginica). La idea es usar algunos de los datos de flores para entrenar un algoritmo y si podemos deducir la especie de las otras flores (no clasificadas) usando solo sus medidas.

El dataset es un diccionario con diferentes datos. Esencialmente en "data" tiene un array con las medidas de ancho y largo de pétalo y sépalo (atributos, o "features" en inglés) de 150 flores y en "target" tiene un numero (0, 1 ó 2) que representa la especie de estas flores. Veamos un poco la estructura de estos datos. El diccionario tiene las siguientes claves:

```
>>> print("Claves del diccionario iris_dataset:\n", iris_dataset.keys())
Claves del diccionario iris_dataset:
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR',
'feature names', 'filename'])
```

Las flores se clasifican en tres:

```
>>> print("Target names:", iris_dataset['target_names'])
Target names: ['setosa' 'versicolor' 'virginica']
```

Y los atributos son cuatro por cada flor:

```
>>> print("Feature names:\n", iris_dataset['feature_names'])
    Feature names:
        ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal
width (cm)']
```

Son 150 flores etiquetadas, con cuatro atributos cada una, en un array de numpy. Las etiquetas son 0, 1 y 2 y se guardan también en un array:

Visualización de los datos

Hagamos primero unos gráficos exploratorios para ver los datos y entender las correlaciones entre los atributos, usando un color diferente para cada especie de flor.

```
import pandas as pd

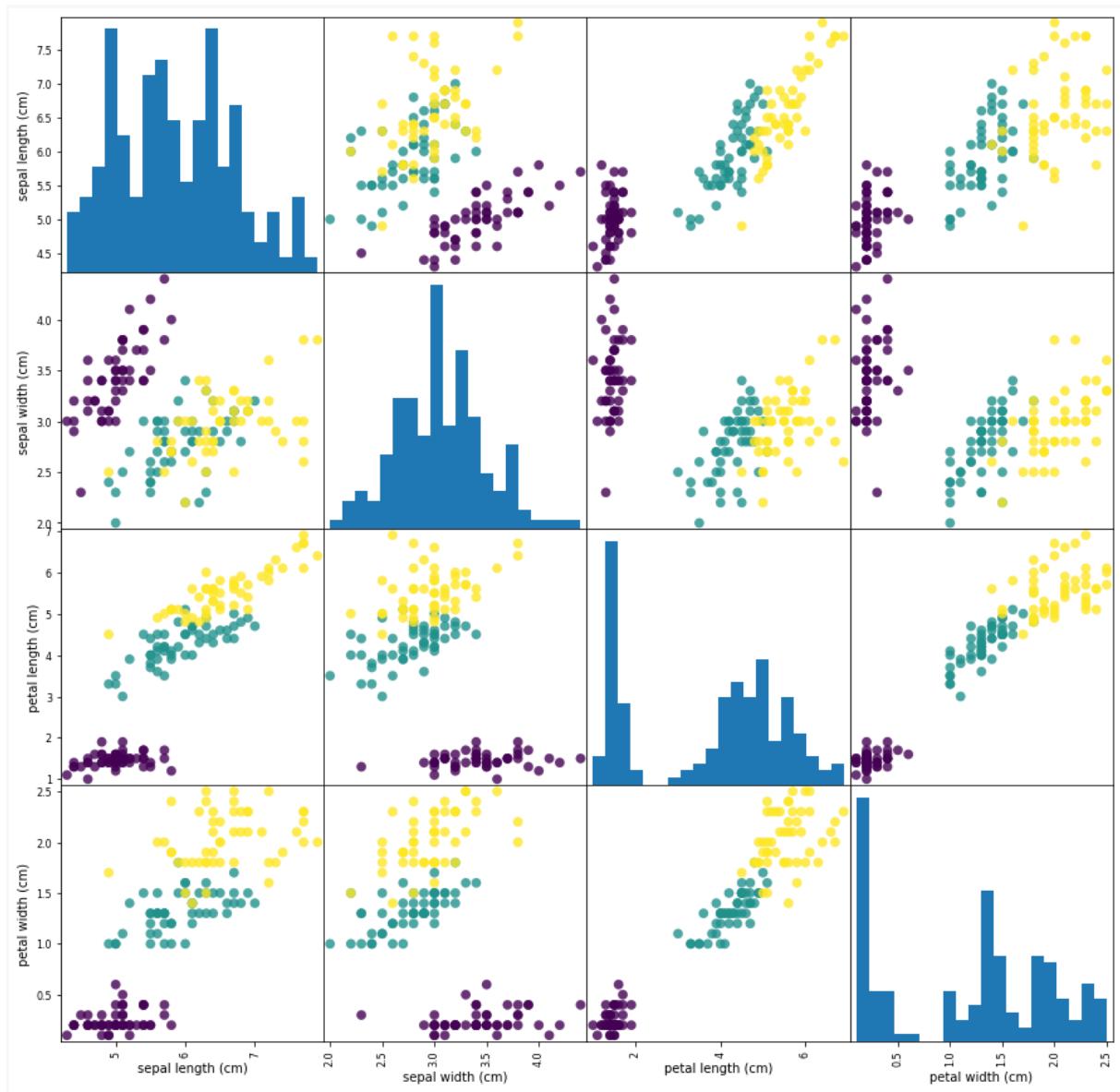
# creamos un dataframe de los datos de flores

# etiquetamos las columnas usando las cadenas de iris_dataset.feature_names

iris_dataframe = pd.DataFrame(iris_dataset['data'], columns = iris_dataset.feature_names)

# y hacemos una matriz de gráficos de dispersión, asignando colores según la especie

pd.plotting.scatter_matrix(iris_dataframe, c = iris_dataset['target'], figsize = (15, 15), marker = 'o', hist_kwds = {'bins': 20}, s = 60, alpha = 0.8)
```



Notamos que una de las especies se distingue más fácilmente de las otras dos, mientras que las otras presentan cierta superposición.

Ejercicio 11.10: Seaborn

Repetí el gráfico anterior pero usando seaborn en lugar de pandas para graficar, y guardá el código correspondiente en un archivo `iris_seaborn.py` para entregarlo.

Sugerencia: Usando `iris_dataframe['target'] = iris_dataset['target']`, agregá al DataFrame el atributo `target` de cada flor para poder hacer un `sns.pairplot()` seteando `hue` sobre las especies de iris.

Training y testing

Como dijimos antes, vamos a entrenar un algoritmo y luego a evaluar su capacidad de clasificar. Para evitar sesgos y sobreajustes tenemos que partir al conjunto de datos en dos:

- una parte de los datos (training) será de entrenamiento del algoritmo y
- otra parte (testing) será usada para la evaluación.

La librería sklearn trae funciones que hacen esta separación (split) de forma aleatoria, como se ve a continuación (en este caso fijamos una semilla con `random_state = 0`, luego la sacaremos). Obviamente separamos tanto los atributos (features) como su clase (target). En este caso usaremos el 75% de los datos para entrenar y el 25% restante para evaluar.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state = 0)

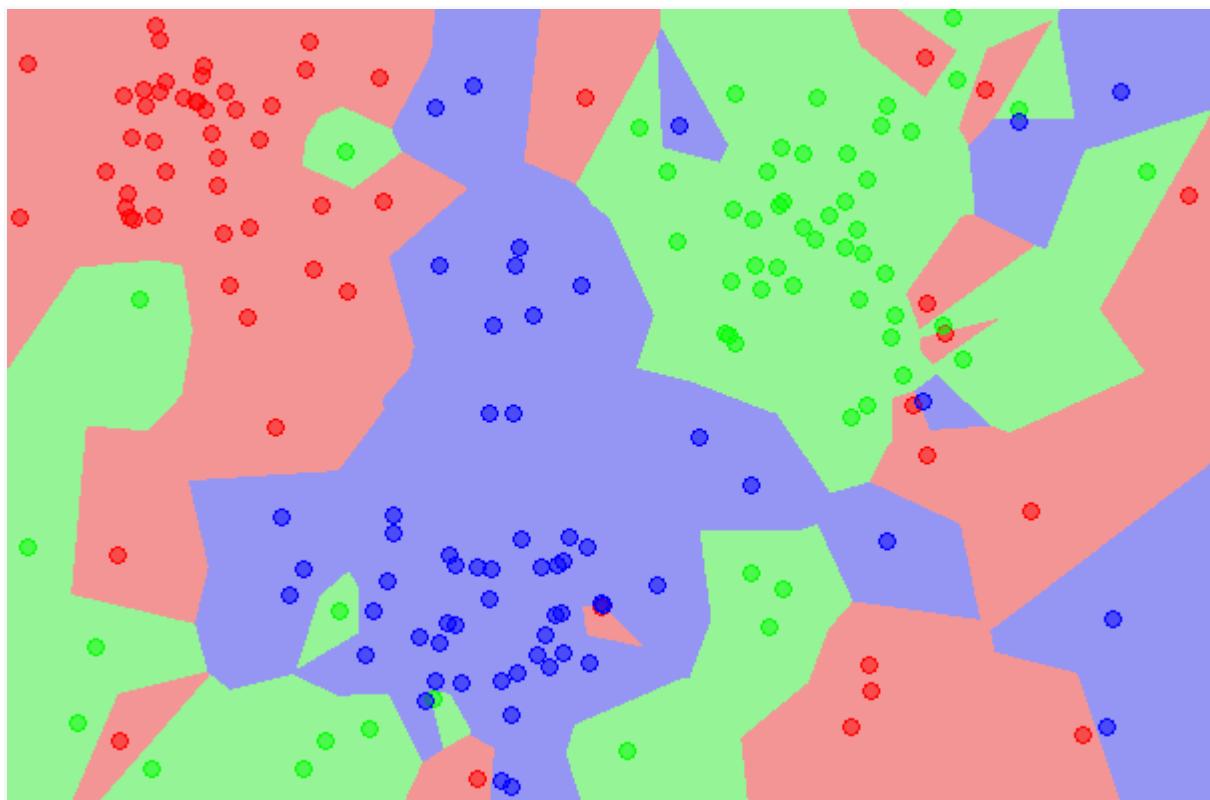
>>> print("X_train shape:", X_train.shape)
>>> print("y_train shape:", y_train.shape)
    X_train shape: (112, 4)
    y_train shape: (112,)

>>> print("X_test shape:", X_test.shape)
>>> print("y_test shape:", y_test.shape)
    X_test shape: (38, 4)
    y_test shape: (38,)
```

Modelar

Ahora vamos a construir nuestro primer modelo. Usaremos un algoritmo sencillo que se llama de "vecinos más cercanos" (K-nearest neighbors_ en inglés, ver [wikipedia](#)). Lo entrenaremos con los datos de entrenamiento y al consultarle por un nuevo dato (de los de testing) lo que hará el algoritmo es buscar al dato de entrenamiento más cercano en el espacio de atributos y asignarle al nuevo dato la especie de esa flor. En otras palabras: cuando le preguntamos por la especie de una flor nueva va a contestarnos con la especie de la flor "más cercana" en el espacio de atributos (ancho y largo del pétalo y el sépalo).

De esta forma el espacio de atributos queda dividido en regiones a las que se asignará cada especie. En el siguiente gráfico puede verse una partición de un espacio de dos atributos y tres clases considerando un vecino más cercano ($k=1$) y entrenado con los datos del gráfico:



A un nuevo punto en este plano el clasificador así entrenado le asignará la clase correspondiente al color de fondo, que coincide con la clase del vecino más cercano.

Creamos una instancia de la clase `KNeighborsClassifier`

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 1)
```

Y la entrenamos con los datos de entrenamiento

```
knn.fit(X_train, y_train)
```

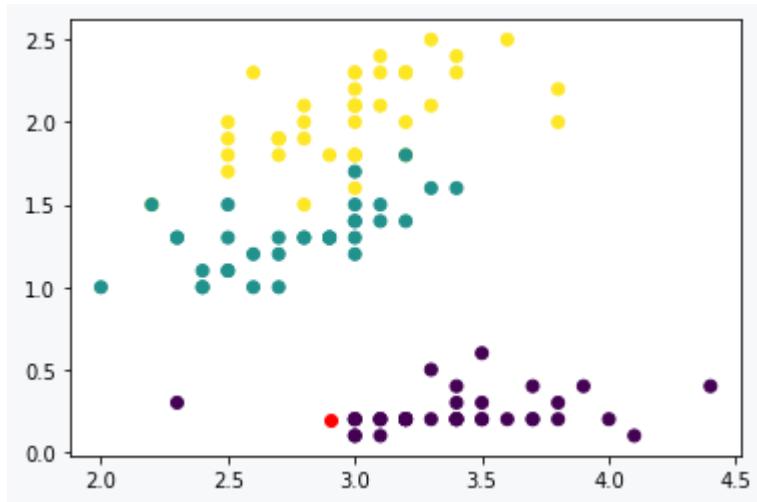
Listo, tenemos el clasificador entrenado. Ahora lo podemos usar para predecir la clase de una nueva flor a partir de sus cuatro medidas:

```
>>> import numpy as np
>>> X_new = np.array([[5, 2.9, 1, 0.2]])
>>> print("X_new.shape:", X_new.shape)
X_new.shape: (1, 4)
```

Grafiquemos este nuevo punto en rojo y veamos su relación con los datos de entrenamiento en dos de los atributos.

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

```
plt.scatter(X_train[:, 1], X_train[:, 3], c = y_train)
plt.scatter(X_new[:, 1], X_new[:, 3], c = 'red')
```



Acá se ve que el punto rojo esta cerca de la clase "setosa". Utilicemos ahora el algoritmo knn entrenado para clasificar el punto `x_new`:

```
>>> prediction = knn.predict(X_new)
>>> print("Predicción:", prediction)
>>> print("Nombre de la Especie Predicha:",
      iris_dataset['target_names'][prediction])
Predicción: [0]
Nombre de la Especie Predicha: ['setosa']
```

Evaluación del modelo

Finalmente, usemos el 25% de los datos etiquetados que nos guardamos para evaluar cuán bien funciona nuestro clasificador.

```
>>> y_pred = knn.predict(X_test)
>>> print("Predicciones para el conjunto de Test:\n", y_pred)
>>> print("Etiquetas originales de este conjunto:\n", y_test)
Predicciones para el conjunto de Test:
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2
1 0
2]
Etiquetas originales de este conjunto:
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2
1 0
1]
```

Se ve que coinciden todos salvo el último. Podemos medir el éxito calculando la fracción de clasificaciones bien hechas (calculamos el promedio de "1 si está bien, 0 si está mal"):

```
>>> print(y_pred == y_test)
>>> print("Test set score: {:.2f}".format(np.mean(y_pred == y_test)))
[ True  True  True  True  True  True  True  True  True  True
 True  True  True  True  True  True  True  True  True  True
 True  True  True  True  True  True  True  True  True  True
 True  True  True  True  True  True  True  True  True  True
 True False]
Test set score: 0.97
```

O, directamente, usando el método `score` que ya viene en el clasificador:

```
>>> print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
Test set score: 0.97
```

Pasando en limpio todo

Lo que hicimos hasta ahora fue:

- 1) Separar los datos en dos conjuntos: `train` y `test`.
- 2) Sefinir un clasificador `knn` y entrenarlo con los datos de `training`.
- 3) Evaluar el clasificador con los datos de `testing`.

```
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'])

knn = KNeighborsClassifier(n_neighbors = 1)
knn.fit(X_train, y_train)

print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

Observá que en este último fragmento de código el split en `test` y `train` es aleatorio, y va a dar resultados (scores) diferentes cada vez que lo corramos.

Ejercicios:

Ejercicio 11.11:

Leé sobre los [clasificadores basados en arboles de decisión](#) y luego usá el objeto clasificador `clf` (definido a continuación) como se usó `knn` en el ejemplo anterior (es

decir, entrená el clasificador sobre el conjunto train y evalúalo sobre el conjunto test). Tanto `knn` como `clf` son clasificadores y heredan los métodos "fit", "predict" y "score" de forma que su uso es casi idéntico. Ventajas del polimorfismo, del que hablamos antes (ver [Ejercicio 8.7](#)). ¿Qué clasificador dió mejores resultados?

```
from sklearn.tree import DecisionTreeClassifier  
  
clf = DecisionTreeClassifier()
```

Ejercicio 11.12:

La comparación anterior de los dos clasificadores puede resultar injusta ya que está basada en *una* partición del conjunto de datos en test y train que podría darle ventaja a uno u otro clasificador, arbitrariamente.

Para evitar esto, repetí 100 veces lo siguiente y calculá el promedio de los scores:

- a) Partición del conjunto original en test y train aleatoriamente (sin fijar la semilla).
- b) Entrenamiento de ambos modelos (`knn` y `clf`) con el conjunto train resultante.
- c) Evaluación de ambos clasificadores (`score`) con el conjunto test resultante.

¿Te animás a agregar también un clasificador de [Random Forest](#)?

Imprimí el promedio de los scores obtenidos y guardá el código en el archivo `clasificadores.py` para entregar.

11.4 Cierre de la clase de Ordenamiento

Cierre clase

- El archivo `burbujeo.py` del [Ejercicio 11.2](#).
- El archivo `comparaciones_ordenamiento.py` del [Ejercicio 11.7](#).
- El archivo `time_ordenamiento.py` del [Ejercicio 11.8](#).
- El archivo `iris_seaborn.py` del [Ejercicio 11.10](#).
- Opcionalmente, el archivo `clasificadores.py` del [Ejercicio 11.12](#).

Como de costumbre, completá por favor el formulario asociado a la clase y adjuntá los archivos correspondientes.

Gracias !