Laboratorio de Datos

Verano 2024

Introducción a Python - parte 2

Contenido

- Copias
- IDE - Spyder
- Diccionarios
- Módulos
- Manejo de archivos
- Numpy
- Pandas

Copias

Copias

Cuando creamos una lista igual a “a”, al modificar una, se modifica la otra también.

```
a = [1,2,3]
b = a               # b = [1,2,3] igual que a
a.append(1)         # acá el 1 se agrega en ambas
b.append(2)         # acá el 2 se agrega en ambas
```

Copias

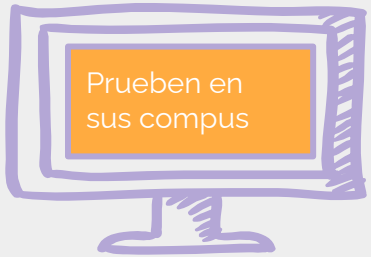
Las listas (y otros objetos) tienen un método para hacer copias. Cuando creamos una copia b de a, modificar una no tiene efecto sobre la otra.

```
a = [2,3,[100,101],4]
b = a.copy()
b == a # True
b is a # False
```

Probar en **pythontutor**

```
a = [2,3,[100,101],4]
b = a.copy()
b == a # True
b is a # False
```

```
a.append(5)
print(b)
```

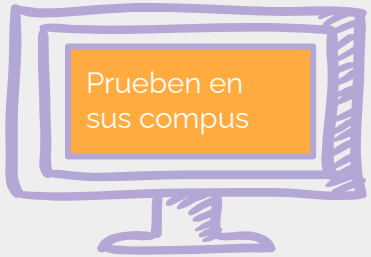


Probar en **pythontutor**

```
a = [2,3,[100,101],4]
b = a.copy()
b == a # True
b is a # False
```

```
a.append(5)
print(b)
```

```
a[2].append(102)
print(b)
```

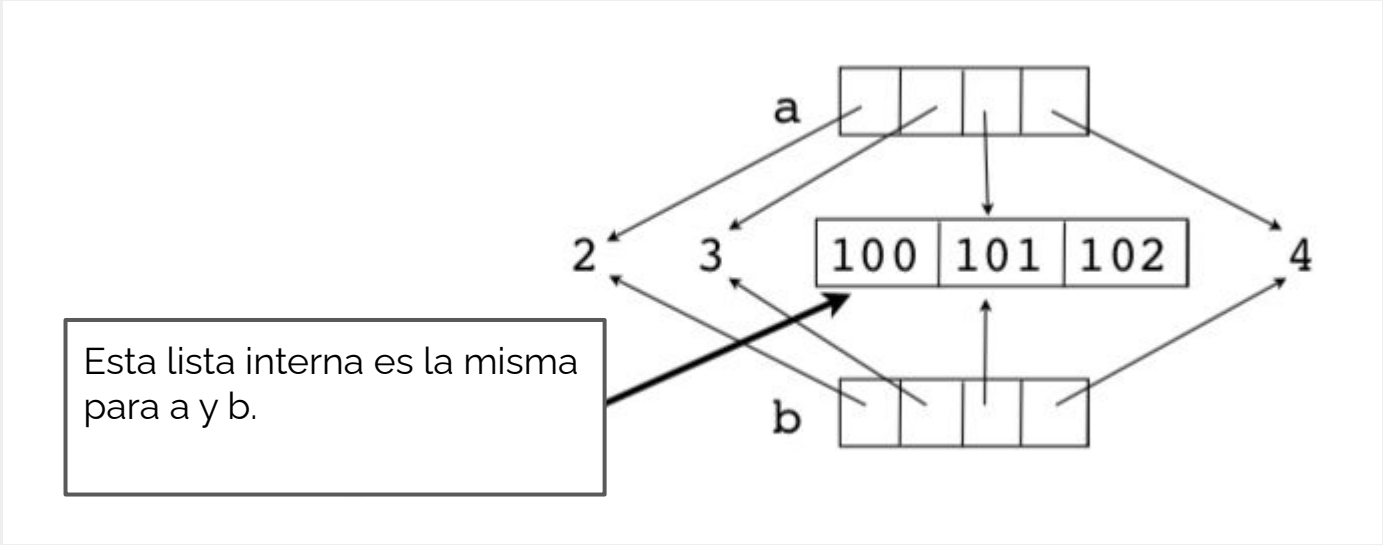


Probar en **pythontutor**

```
a = [2,3,[100,101],4]
b = a.copy()
b == a # True
b is a # False
```

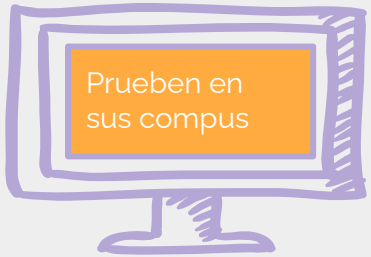
```
a.append(5)
print(b)
```

```
a[2].append(102)
print(b)
```



Deepcopy

```
import copy

a = [2,3,[100,101],4]
b = copy.deepcopy(a)

a.append(5)
print(b)
a[2].append(102)
print(b)
```



Ejercicio

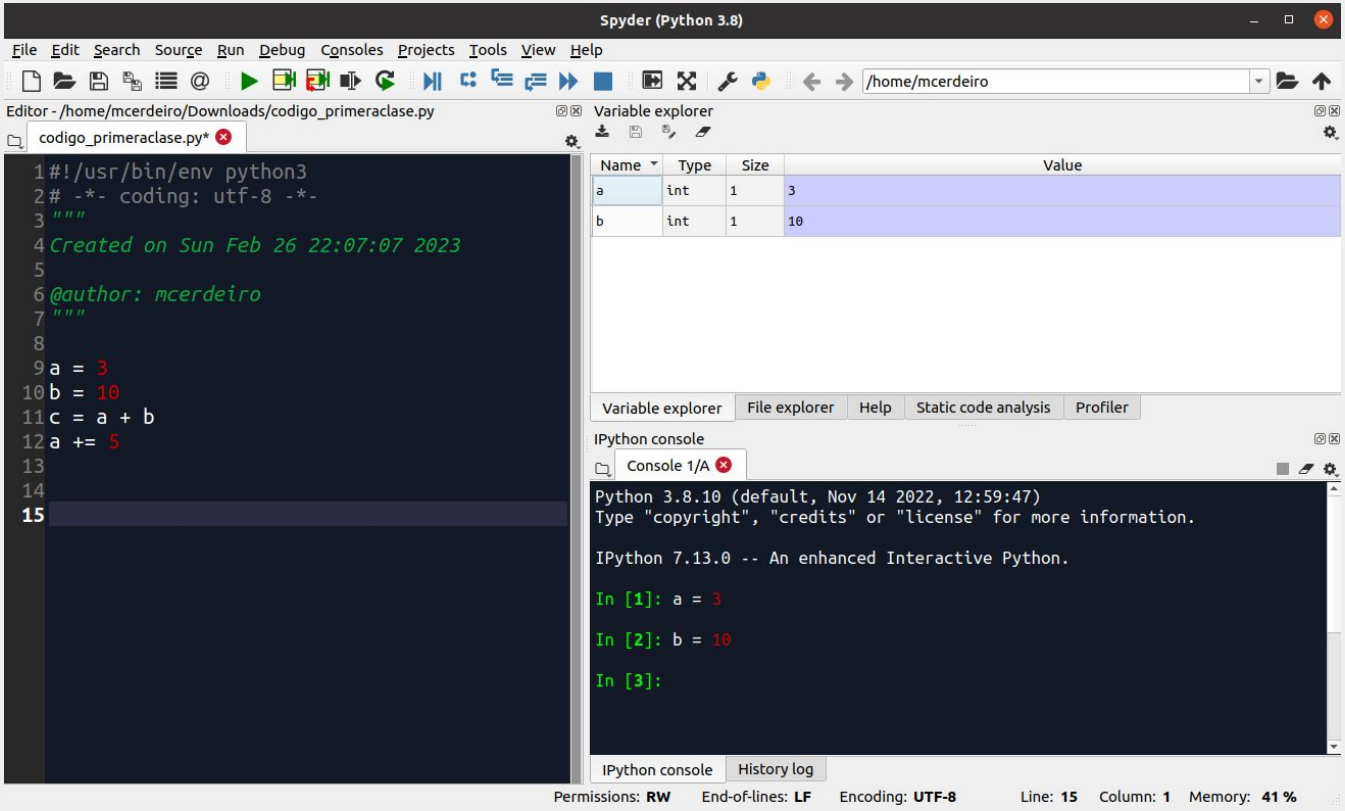
Probar las funciones anteriores utilizando y no utilizando copias.

Spyder

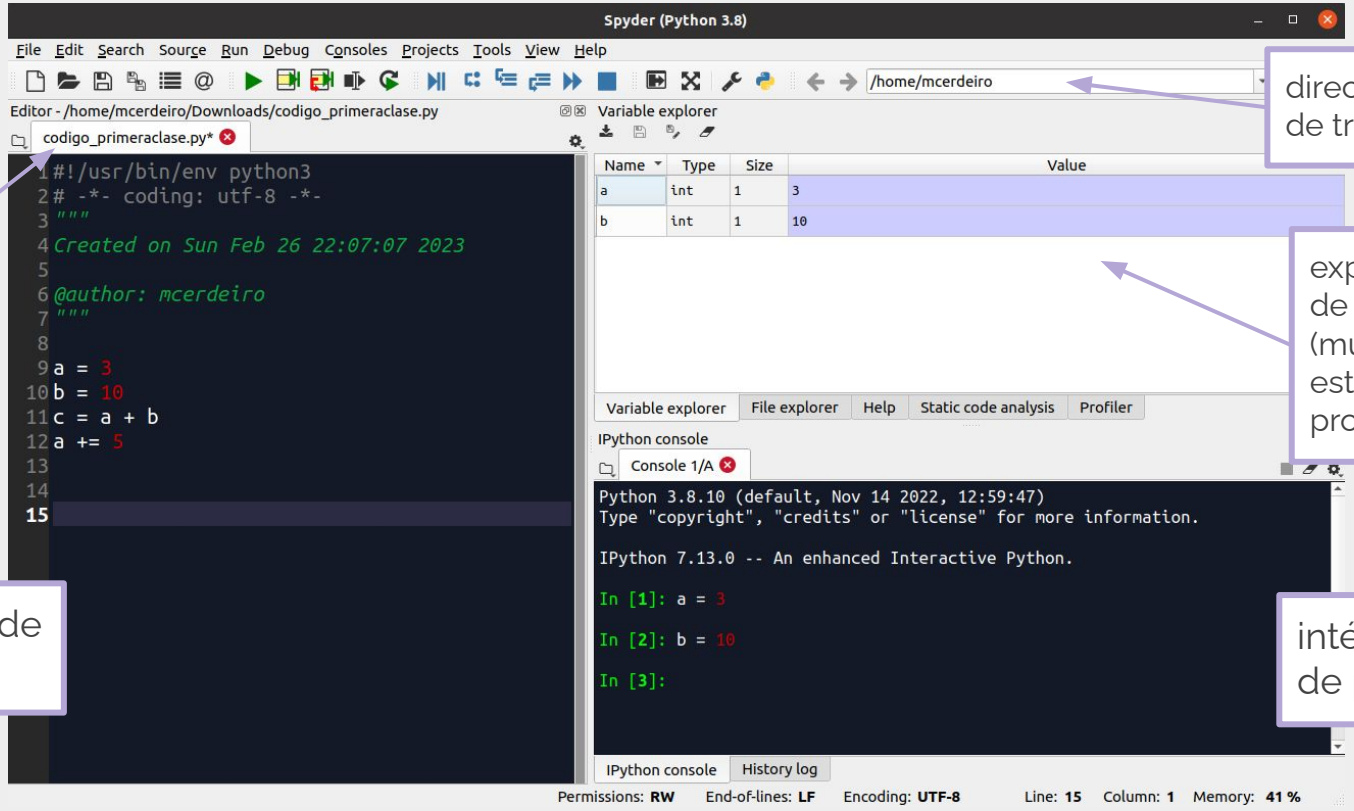
Entorno de desarrollo integrado (*IDE*) - Spyder

+ código abierto (open source)
+ multiplataforma (sirve en linux, ios, windows…)
+ es cómodo para escribir código, ejecutarlo, corregirlo, probarlo y utilizarlo, en el mismo entorno
+ el editor de texto remarca palabras clave del lenguaje
+ tiene atajos (shortcuts) útiles (y modificables a gusto de cada unx)

Entorno de desarrollo integrado (*IDE*) - Spyder



Entorno de desarrollo integrado (*IDE*) - Spyder



nombre del programa

directorio actual de trabajo

explorador de variables (muestra el estado del programa)

editor de texto

intérprete de python

Diccionarios

Diccionarios

Los diccionarios son útiles si vamos a querer buscar rápidamente (por claves).

- ➔ Se construyen con llaves
- ➔ Cada entrada tiene una clave y un valor, separados por dos puntitos :
- ➔ Las entradas se separan con comas

```
{clave1: valor1, clave2: valor2, … }
```

- ➔ Se acceden con corchetes indicando una clave
- ➔ Tanto las claves como los valores pueden ser de distintos tipos de objetos
- ➔ Las claves deben ser de tipo inmutable

Ejemplo

```
dias_engl = {'lunes': 'monday', 'martes': 'tuesday', 'miércoles': 'wednesday', 'jueves':
'thursday'}

>>> dias_engl['lunes']
'monday'

>>> dias_engl['viernes']
Traceback (most recent call last):

  File "<ipython-input-33-ee0fa133453b>", line 1, in <module>
    dias_engl['viernes']

KeyError: 'viernes'

>>> dias_engl['viernes'] = 'friday'     # agrego la entrada
>>> dias_engl['viernes']
'friday'
```

Ejemplo

También se pueden armar y modificar agregando entradas:

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

# Agregamos elementos
feriados[(1, 1)] = 'Año nuevo'
feriados[(1, 5)] = 'Día del trabajador'
feriados[(13, 9)] = 'Día del programador'

# Modifico una entrada
feriados[(13, 9)] = 'Día de la programadora'
```

Diccionarios

También se pueden armar a partir de una lista de tuplas (clave, valor).

```
>>> cuadrados = dict([(1,1),(2,4),(3,9),(4,16)])
>>> cuadrados[2]
4
```

La función **zip** genera tuplas a partir de dos listas:

```
>>> for t in zip([1,2,3,4],[1,4,9,16]):
          print(t)
(1, 1)
(2, 4)
(3, 9)
(4, 15)
```

Es decir que podemos construir el diccionario a partir de dos listas (claves y valores);

```
>>> cuadrados = dict(zip([1,2,3,4],[1,4,9,16]))
```

Ejercicio

Construí una función `traductor_geringoso(lista)` que, a partir de una lista de palabras, devuelva un diccionario geringoso.
Las claves del diccionario deben ser las palabras de la lista y los valores deben ser sus traducciones al geringoso.

Por ejemplo:

```
>>> lista = ['banana', 'manzana', 'mandarina']
>>> traductor_geringoso(lista)

{'banana': 'bapanapanapa', 'manzana': 'mapanzapanapa', 'mandarina': 'mapandaparipinapa'}
```

Módulos

Módulos

Si bien Python tiene muchas funciones que se pueden usar directamente, hay muchas otras que están disponibles dentro de módulos.

Un **módulo** es una **colección de funciones** que alguien (o una comunidad) desarrollaron y empaquetaron para que estén disponibles para todo el mundo.

Para que las funciones estén disponibles para ser utilizadas en mi programa, tengo que usar la instrucción **import**.

Módulos

Si quiero generar números aleatorios, que están en el módulo random, tengo que escribir:

```
import random
prueba = random.random()
print(prueba)
prueba = random.random()
print(prueba)
prueba = random.random()
print(prueba)

random.seed(COMPLETAR con un NÚMERO)
prueba = random.random()
print(prueba)
prueba = random.random()
print(prueba)
```

Módulos

Módulo math tiene funciones matemáticas.

```
import math

math.sqrt(2)

math.exp(x)

math.cos(x)

math.gcd(15,12)
```

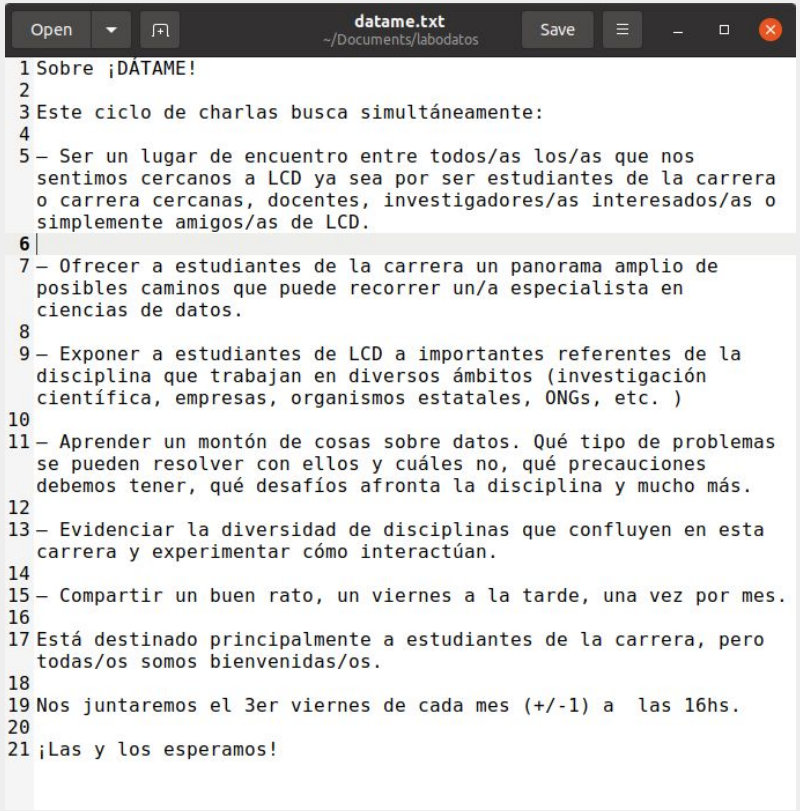
Archivos

Archivos

Frecuentemente vamos a utilizar una fuente de datos, que en muchos casos va a estar en un archivo. Tenemos que poder manejar archivos: leer, crear, modificar, guardar archivos de distintos tipos.

```
f = open(nombre_archivo, 'rt' )     # abrir para lectura ('r' de read, 't' de text)
data = f.read()
f.close()
data
print(data)
```

Ejemplo



Ejemplo:
nombre_archivo = ‘datame.txt’

```
nombre_archivo = ‘datame.txt’
f = open(nombre_archivo, 'rt' )
data = f.read()
f.close()

data
print(data)
```

Archivos

```
with open(nombre_archivo, 'rt') as file:     # otra forma de abrir archivos
    data = file.read()
    # 'data' es una cadena con todo el texto en el archivo

data
print(data)
```

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

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

Python tiene dos modos de salida. En el primero, escribimos data en el intérprete y Python muestra la representación **cruda** de la cadena, incluyendo comillas y códigos de escape (\n).
Cuando escribimos print(data), en cambio, se imprime la salida **formateada** de la cadena.

Archivos .csv

csv = comma separated values

- ➔ son “planillas” guardadas como texto
- ➔ cada línea de texto es una fila de la planilla
- ➔ las comas separan las columnas

```
a,b,c
d,e,f
x,y,z
u,v,w
```

a	b	c
d	e	f
x	y	z
u	v	w

Archivos .csv

csv = comma separated values

Ejemplo:
nombre_archivo = 'cronograma_sugerido.csv'

Cuatrimestre	Asignatura	Correlatividad de Asignaturas
3	Álgebra I	CBC
3	Algoritmos y Estructuras de Datos I	CBC
4	Análisis I	CBC
4	Electiva de Introducción a las Ciencias Naturales	CBC
5	Análisis II	Análisis I
5	Álgebra Lineal Computacional	Álgebra I – Algoritmos y Estructuras de Datos I
5	Laboratorio de Datos	Algoritmos y Estructuras de Datos I
6	Análisis Avanzado	Análisis II, Álgebra I
6	Algoritmos y Estructuras de Datos II	Algoritmos y Estructuras de Datos I
7	Probabilidad	Análisis Avanzado
7	Algoritmos y Estructura de Datos III	Algoritmos y Estructuras de Datos II
8	Intr. a la Estadística y Ciencia de Datos	Lab de Datos, Probabilidad, Álgebra Lineal Computacional
8	Intr. a la Investigación Operativa y Optimización	Alg y Estruc de Datos III, Análisis II, Álgebra Lineal Computacional
8	Intr. al Modelado Continuo.	Análisis Avanzado, Álgebra Lineal Computacional, Alg y Estructura de Datos II

Archivos .csv

```
nombre_archivo = 'cronograma_sugerido.csv'
with open(nombre_archivo, 'rt') as file:
    for line in file:
        datos_linea = line.split(',')
        print(datos_linea[1])
```

¿Cómo podemos armar una lista con todas las materias del cronograma?

Módulo csv

Es útil para trabajar con archivos .csv

```
f = open(nombre_archivo)
filas = csv.reader(f)
for fila in filas:
    instrucciones

f.close()
```

Acá `filas` es un iterador.

Si la primera fila son encabezados, podemos guardarlos así:

```
f = open(nombre_archivo)
filas = csv.reader(f)
encabezado = next(filas)  # un paso del iterador
for fila in filas:        # ahora el iterador sigue desde la segunda fila
    instrucciones

f.close()
```

Ejemplo

Definimos `registros(nombre_archivo)` que recorre el archivo indicado, conteniendo información de un cronograma sugerido de cursada, y devuelve la información como una lista de diccionarios.
Las claves de los diccionarios son las columnas del archivo, y los valores son las entradas de cada fila para esa columna.

```
def registros(nombre_archivo):
    lista = []
    with open(nombre_archivo, 'rt') as f:
        filas = csv.reader(f)
        encabezado = next(filas)
        for fila in filas:
            registro = dict(zip(encabezado,fila)) # armo el diccionario de cada fila
            lista.append(registro)                # lo agrego a la lista
    return lista
```

Ejercicios

+ Escribir una función `generala_tirar()` que simule una tirada de dados para el juego de la generala. Es decir, debe devolver una lista aleatoria de 5 valores de dados. Por ejemplo `[2,3,2,1,6]`.
+ Opcional:
  Agregar al ejercicio `generala_tirar()` que además imprima en pantalla si salió poker, full, generala, escalera o ninguna de las anteriores. Por ejemplo, si sale `2,1,1,2,2` debe devolver `[2,1,1,2,2]` e imprimir en pantalla `Full`
+ Escribir un programa que recorra las líneas del archivo `'datame.txt'` e imprima solamente las líneas que contienen la palabra `'estudiante'`.

SIN utilizar el módulo csv:

+ Utilizando el archivo `cronograma_sugerido`, armar una lista de las materias del cronograma, llamada "`lista_materias`".
+ Luego, definir una función "`cuantas_materias`(n)" que, dado un número de cuatrimestre (n entre 3 y 8), devuelva la cantidad de materias a cursar en ese cuatrimestre.
  Por ejemplo: `cuantas_materias(5)` debe devolver 3.

UTILIZANDO el módulo csv:

+ Definir una función **`materias_cuatrimestre(nombre_archivo, n)`** que recorra el archivo indicado, conteniendo información de un cronograma sugerido de cursada, y devuelva una lista de diccionarios con la información de las materias sugeridas para cursar el n-ésimo cuatrimestre.

Cortamos 15 minutos

Ejercicio

Definir una función `materias_cuatrimestre(nombre_archivo, n)` que recorra el archivo indicado, conteniendo información de un cronograma sugerido de cursada, y devuelva una lista de diccionarios con la información de las materias sugeridas para cursar el n-ésimo cuatrimestre.
Debe funcionar así:

```
materias_cuatrimestre('cronograma_sugerido.csv', 3):

[{'Cuatrimestre': '3',
  'Asignatura': 'Álgebra I',
  'Correlatividad de Asignaturas': 'CBC'},
 {'Cuatrimestre': '3',
  'Asignatura': 'Algoritmos y Estructuras de Datos I',
  'Correlatividad de Asignaturas': 'CBC'}]
```

Numpy

Numpy (Numerical Python)

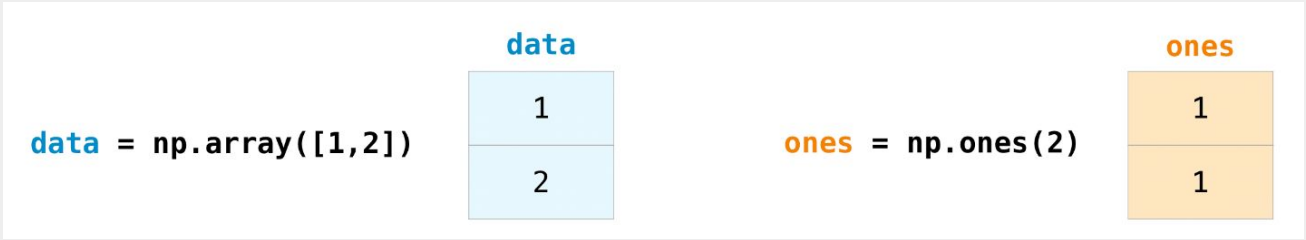
- ➔ Colección de módulos de código abierto que tiene aplicaciones en casi todos los campos de las ciencias y de la ingeniería.
- ➔ Estándar para trabajar con datos numéricos en Python.
- ➔ Muchas otras bibliotecas de Python (Pandas, SciPy, Matplotlib, scikit-learn, scikit-image, etc) usan numpy.
- ➔ Objetos: matrices multidimensionales por medio del tipo **ndarray** (un objeto n-dimensional homogéneo, es decir, con todas sus entradas del mismo tipo)
- ➔ Métodos para operar **eficientemente** sobre las mismas.

Se lo suele importar así:

```
import numpy as np
```

Numpy (Numerical Python)

```
import numpy as np

a = np.array([1, 2, 3, 4, 5, 6]) # 1 dimensión
b = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]) # 2 dimensiones
print(a[0])
print(b[0])
print(b[2][3])
print(b[2,3])
np.zeros(2) # matriz de ceros del tamaño indicado
np.zeros((2,3))
```

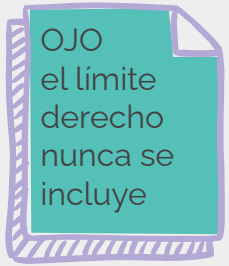


Numpy (Numerical Python)

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) # array([2, 4, 6, 8])
```

También podés usar np.linspace() para crear un vector de valores equiespaciados 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

Generá un vector que tenga los números impares entre el 1 y el 19 inclusive usando arange().

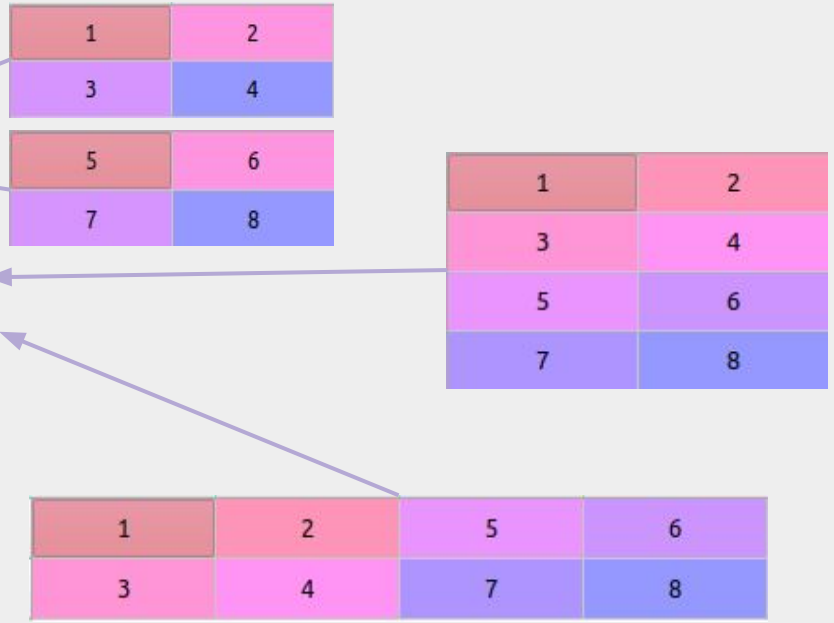
Repetí el ejercicio usando linspace(). ¿Qué diferencia hay en el resultado?

Ejemplos

```
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
np.concatenate((a, b))
```

```
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6], [7, 8]])
```

```
z = np.concatenate((x, y), axis = 0)
z = np.concatenate((x, y), axis = 1)
```

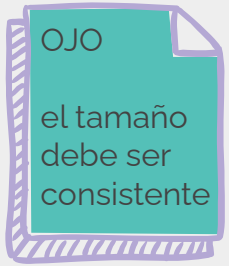


Ejemplos

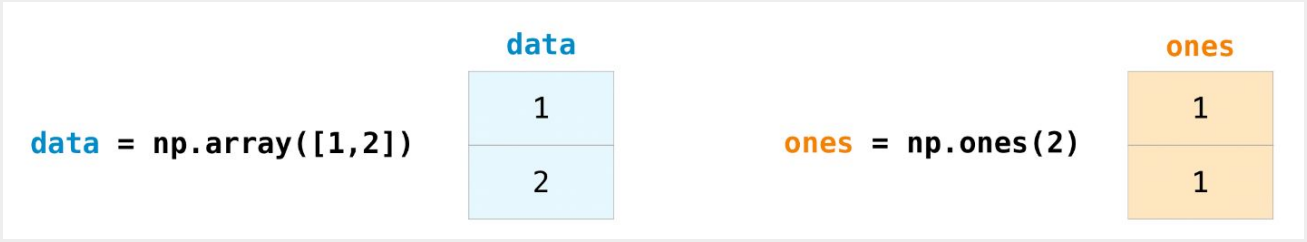
Un ejemplo de array de 3 dimensiones.

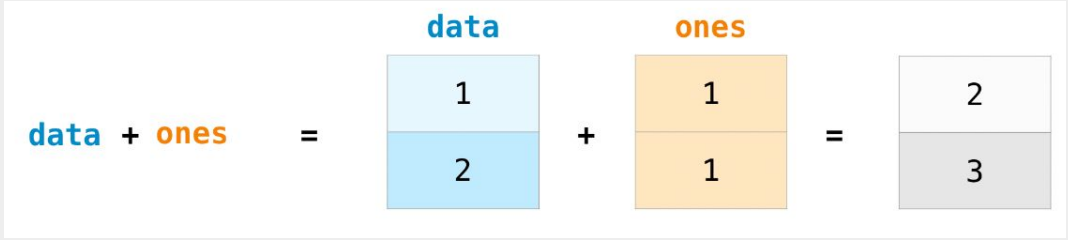
```
array_ejemplo = np.array([[[0, 1, 2, 3],
                           [4, 5, 6, 7]],
                           [[3, 8, 10, -1],
                           [0, 1, 1, 0]],
                           [[3 ,3 ,3, 3],
                           [5, 5, 5, 5]]])
```

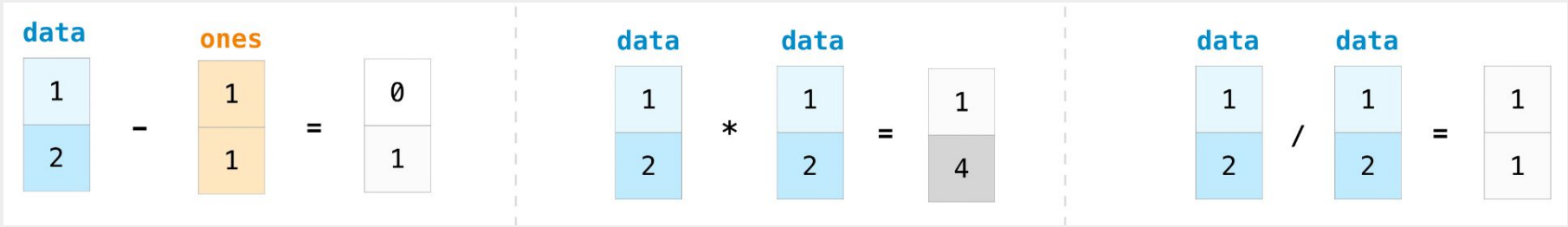
```
array_ejemplo.ndim # cantidad de dimensiones - 3
array_ejemplo.shape # cantidad de elementos en cada eje (3,2,4)
array_ejemplo.size # total de entradas 3*2*4

array_ejemplo.reshape((12,2)) # modifico la forma
array_ejemplo.reshape((4,6))
array_ejemplo.reshape((3,-1)) # 3 por lo que corresponda
```



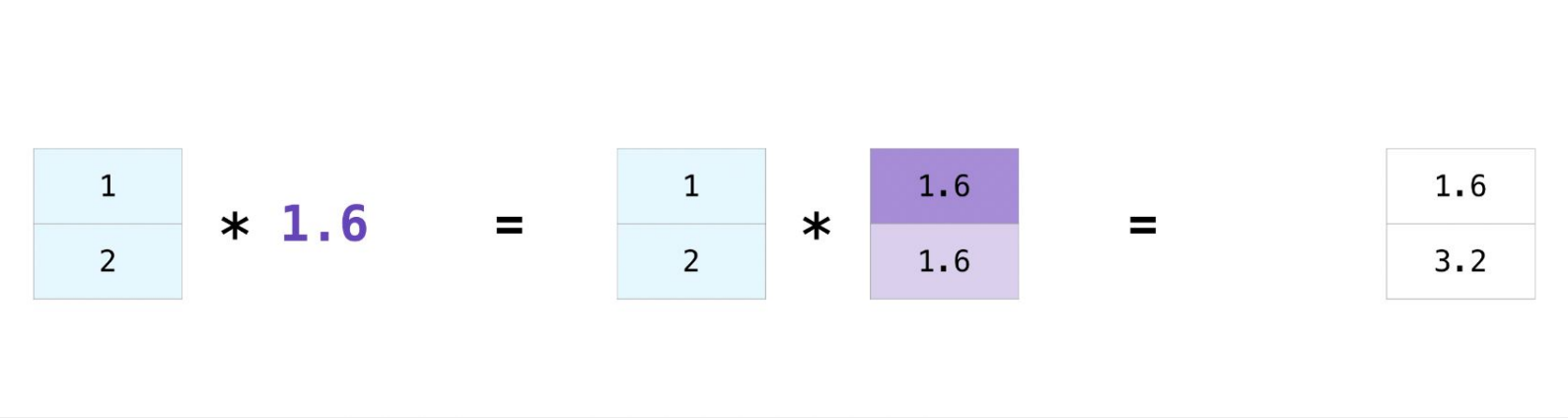
Operaciones

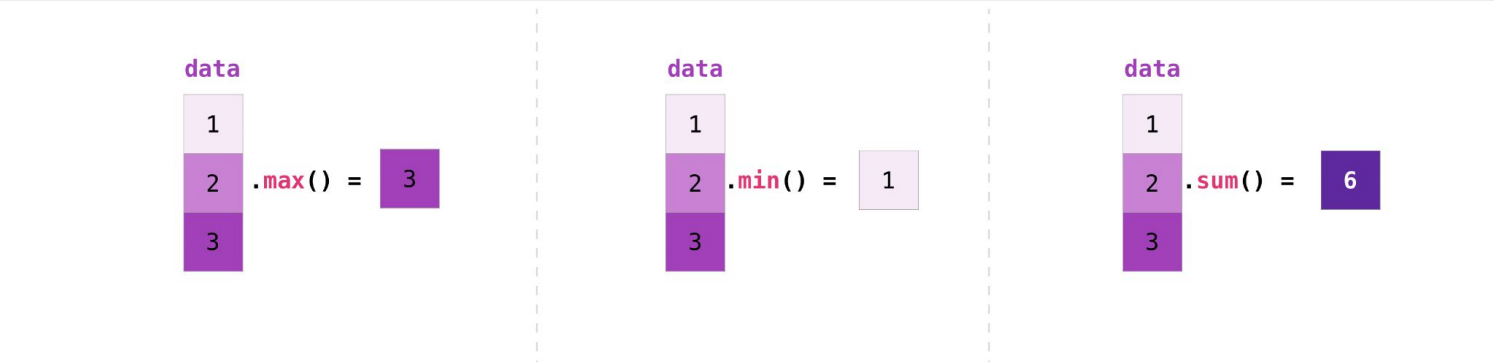




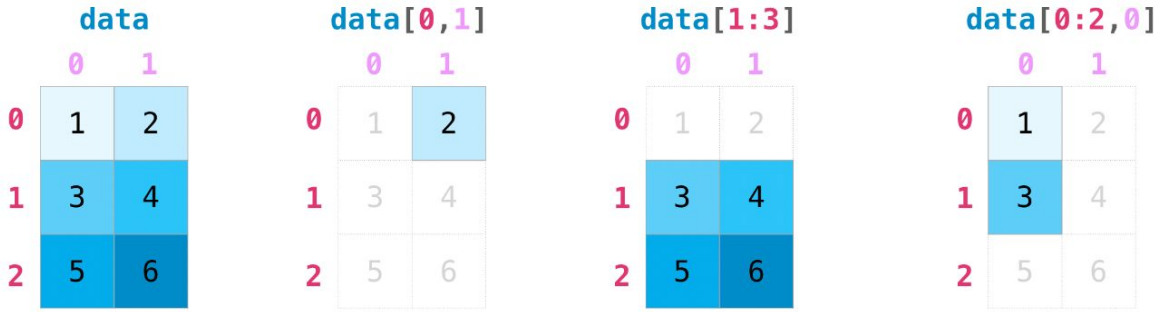


Operaciones

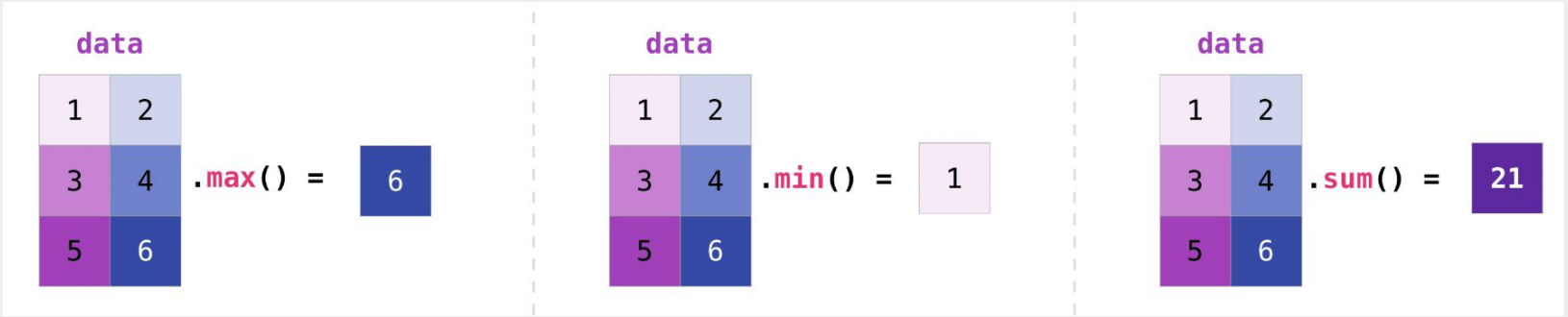


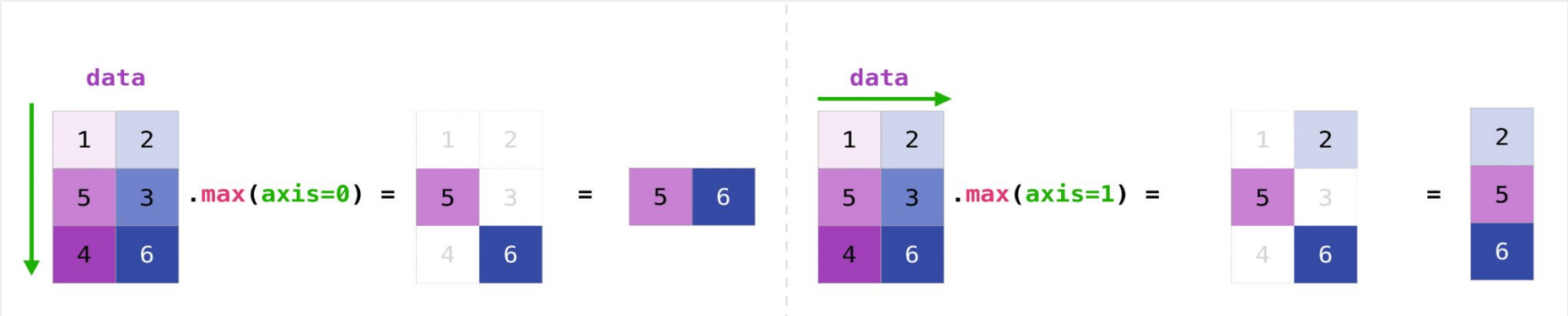


Operaciones



Operaciones





Ejercicio

Definir una función pisar_elemento(M,e) que tome una matriz de enteros M y un entero e y devuelva una matriz similar a M donde las entradas coincidentes con e fueron cambiadas por -1.

Por ejemplo si M = np.array([[0, 1, 2, 3], [4, 5, 6, 7]]) y e = 2, entonces la función debe devolver la matriz np.array([[0, 1, -1, 3], [4, 5, 6, 7]])

Pandas

Pandas

- Pandas es una extensión de NumPy para manipulación y análisis de datos.
- Ofrece estructuras de datos y operaciones para manipular tablas de datos (numéricos y de otros tipos) y series temporales.
- Tipos de datos fundamentales: **DataFrames** que almacenan tablas de datos y las **Series** que contienen secuencias de datos.

```
import pandas as pd
```

Gobierno Abierto

1 ÁRBOL CADA 7 PERSONAS

CIUDAD VERDE

Datos de árboles en CABA

HAY 372.687

ÁRBOLES EN LAS CALLES DE BUENOS AIRES

MAPA DE DENSIDAD

*PARQUE CHAS EL BARRIO CON MAYOR DENSIDAD

ESPECIES MÁS FRECUENTES

Fresno americano	39%	141820
Plátano	9%	34781
Paraíso	7%	24558
Ficus	7%	24076
Tilo	5%	17477
Jacarandá	3%	11044

Información basada en el censo de árboles de alineación 2011 realizado por el Gobierno de la Ciudad de Buenos Aires. La información pudo sufrir variaciones en periodos posteriores.

Buenos Aires Ciudad

EN TODO ESTÁS VOS

Datos de arbolado porteño

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

Pandas

```
import pandas as pd

import os

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.

Pandas

```
>>> df.head() # primeras líneas

   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
```

Pandas

```
>>> 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)
```

Pandas

```
>>> 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
```

Operaciones con DataFrames

Los **filtros** sirven para seleccionar porciones del dataframe. Para eso, utilizamos una condición. Como por ejemplo si queremos ver la información de los Ombúes, utilizamos un filtro con la condición de que el nombre sea “Ombú”.

```
>>> df['nombre_com'] == 'Ombú'

0        False

1        False

2        False

3         True

...
```

Esta instrucción devuelve una serie de valores de verdad. En cada caso, indica si se cumple o no la condición.

```
>>> (df['nombre_com'] == 'Ombú').sum()          # cantidad de filas de Ombú

590

>>> df[df['nombre_com'] == 'Ombú']
```

Esta instrucción devuelve un sub-dataframe, donde sólo están las filas de Ombúes.

Operaciones con DataFrames

```
df['nombre_com'].unique()                       # una vez cada nombre

cant_ejemplares = df['nombre_com'].value_counts()

cant_ejemplares.head(10)  # tabla con los 10 nombres más frecuentes
```

Operaciones con DataFrames

```
>>> df_jacarandas = df[df['nombre_com'] == 'Jacarandá']

>>> cols = ['altura_tot', 'diametro', 'inclinacio']

>>> df_jacarandas = df_jacarandas[cols]              # así seleccionamos un conjunto de columnas.

>>> 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
```

Operaciones con DataFrames

OJO! estos sub-dataframes son realmente pedazos del DF original. Es decir que si los manipulamos, los cambios que hagamos tendrán efecto también en el original.

Si vamos 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()
```

Filtros por índice y por posición

El índice de este 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.
Tiene los nombres de las especies como índice y sus respectivas cantidades como dato asociado.

Para acceder a una o varias filas de un dataframe, podemos `loc` e `iloc`.
iloc sirve para acceder por posición de las filas, es decir que `df.iloc[165]` nos devuelve la fila número 165.
En cambio loc sirve para acceder por el valor del índice. Es decir que `df.loc[165]` nos devuelve la fila cuyo índice tiene el valor 165.
En el caso de df es lo mismo, ya que como índice tiene simplemente el número de fila. Pero probemos con el sub-dataframe de Jacarandás.

```
In [35]: df_jacarandas.head()
Out[35]:
     altura_tot  diametro  inclinacio
165           5        10           0
166           5        10           0
167           5        10           0
168           5        10           0
169           5        10           0

In [36]: df_jacarandas.loc[165]
Out[36]:
altura_tot     5
diametro      10
inclinacio     0
Name: 165, dtype: int64
```

La primera fila es la 165

```
In [45]: df_jacarandas.tail()
Out[45]:
       altura_tot  diametro  inclinacio
51104           7        97           4
51172           8        28           8
51180           2        30           0
51207           3        10           0
51375          17        40          20

In [46]: df_jacarandas.loc[51172]
Out[46]:
altura_tot     8
diametro      28
inclinacio     8
Name: 51172, dtype: int64

In [47]: df_jacarandas.iloc[51172]
Traceback (most recent call last):
```

Tira error porque no hay tantas filas.

También podemos acceder a rebanadas (slices) usando iloc, y podemos seleccionar simultáneamente filas y columnas, si separamos con comas las respectivas selecciones.

```
In [49]: df_jacarandas.iloc[0:3]
Out[49]:
     altura_tot  diametro  inclinacio
165           5        10           0
166           5        10           0
167           5        10           0
```

Esto nos devuelve los datos correspondientes a las primeras 3 filas.

```
In [48]: df_jacarandas.iloc[-5:,2]
Out[48]:
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').

Siempre vienen acompañados del índice.

Si queremos seleccionar una sola columna podemos especificarla por medio de su nombre.Obtenemos una serie en lugar de un DataFrame. O podemos seleccionar una lista de columnas, y obtener un DataFrame.

```
In [51]: jac_diam = df_jacarandas['diametro']

In [52]: jac_diam
Out[52]:
165      10
166      10
167      10
168      10
169      10
        ..
51104    97
51172    28
51180    30
51207    10
51375    40
Name: diametro, Length: 3255, dtype: int64

In [53]: type(jac_diam)
Out[53]: pandas.core.series.Series
```

```
In [54]: jac_alt_diam = df_jacarandas[['altura_tot', 'diametro']]

In [55]: jac_alt_diam
Out[55]:
       altura_tot  diametro
165             5        10
166             5        10
167             5        10
168             5        10
169             5        10
...           ...       ...
51104           7        97
51172           8        28
51180           2        30
51207           3        10
51375          17        40

[3255 rows x 2 columns]

In [56]: type(jac_alt_diam)
Out[56]: pandas.core.frame.DataFrame
```

Ejercicios

Ejercicios

Utilizar el dataset de arbolado porteño en parques.

Cargar en un dataframe data_arboles_parques la información del archivo csv.

Armar un dataframe que contenga las filas de Jacarandás y otro con los Palos Borrachos.

Calcular para cada especie seleccionada:

Cantidad de árboles, altura máxima, mínima y promedio, diámetro máximo, mínimo y promedio.

Definir una función cantidad_arboles(parque) que, dado el nombre de un parque, calcule la cantidad de árboles que tiene.

Definir una función cantidad_nativos (parque) que calcule la cantidad de árboles nativos.

Ejercicios

Vamos a trabajar ahora con el archivo 'arbolado-publico-lineal-2017-2018.csv'.

Levantalo y armá un DataFrame data_arboles_veredas que tenga solamente las siguiente 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)
```

Ejercicios

Queremos estudiar si hay diferencias entre los ejemplares de una misma especie según si crecen en un sitio o en otro. Para eso tendremos que juntar datos de dos bases de datos diferentes.

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.

Te proponemos los siguientes pasos para comparar los diámetros a la altura del pecho de las tipas en ambos tipos de entornos.

Ejercicios

- 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 diametro 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.
- 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'.
- 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.