



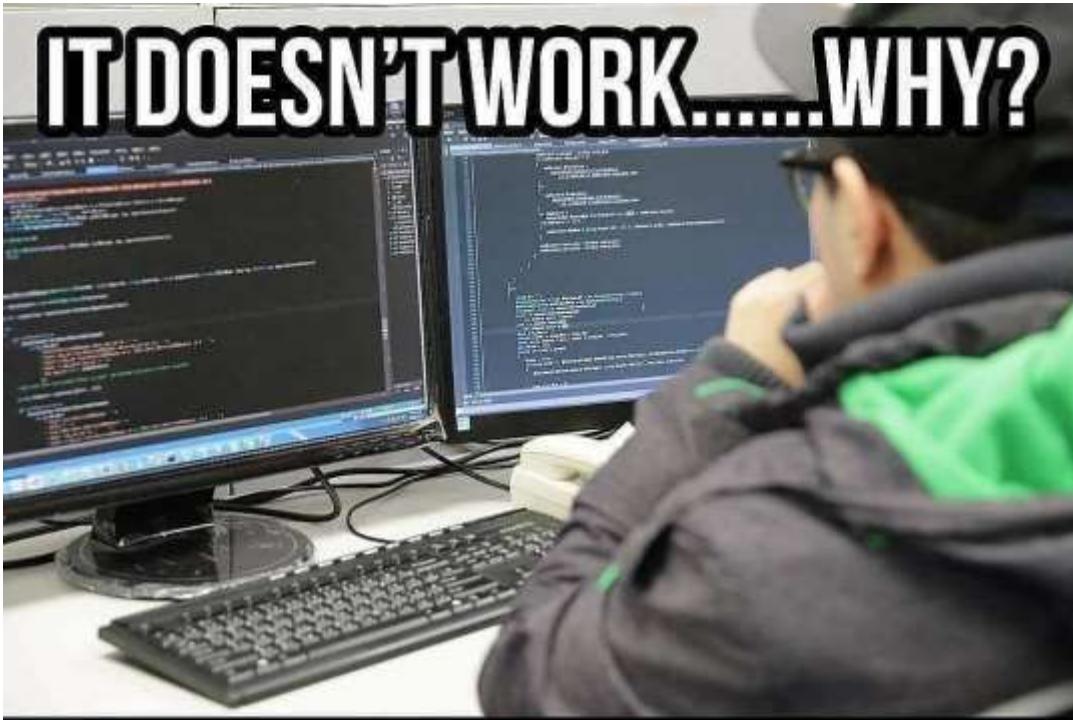
**Escuela Técnica
Roberto Rocca**

Python

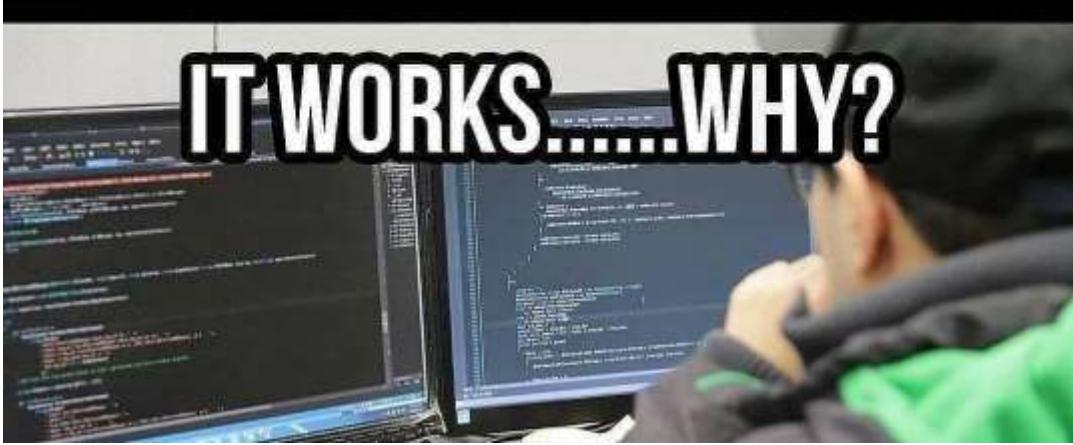


Python

```
36  
37     if path:  
38         self.file = open(os.path.join(path,  
39                             self.name))  
40         self.fingerprints.add(fp)  
41  
42     @classmethod  
43     def from_settings(cls, settings):  
44         debug = settings.get("debug", False)  
45         return cls(job_dir=settings["job_dir"],  
46                     request_seen=request_seen,  
47                     request_fingerprint=request_fingerprint,  
48                     fingerprints=fingerprints,  
49                     file=file,  
50                     fp=fp,  
51                     write=write,  
52                     add=add,  
53                     log=log,  
54                     request=request);  
55  
56     def request_fingerprint(self, request):  
57         request_fingerprint(request)
```



IT DOESN'T WORK.....WHY?



IT WORKS.....WHY?

¿Por qué Python?



-
- Lenguaje de alto nivel
 - Fácil de aprender
 - Tipado dinámico
 - Lenguaje interpretado no compilado
 - Puede extenderse y trabajar con datos implementados en C/C++
 - Es un lenguaje muy útil para scripting
 - Muy utilizado en ciencia e ingeniería
 - Orientado a objetos

¿Por qué Python?



Python permite escribir programas compactos y legibles. Los programas en Python son típicamente más cortos que sus programas equivalentes en C, C++ o Java por varios motivos:

- Los tipos de datos de alto nivel permiten expresar operaciones complejas en una sola instrucción
- La agrupación de instrucciones se hace por identacion (sangria) en vez de llaves de apertura y cierre
- NO es necesario declarar variables ni argumentos.



Origen del nombre

Monty Python (a veces conocidos como Los Pythons) fue un grupo británico de seis humoristas que sintetizó en clave de humor la idiosincrasia británica de los años 1960 y 1970, compuesto por Graham Chapman, John Cleese, Terry Gilliam, Eric Idle, Terry Jones y Michael Palin. Del grupo únicamente Terry Gilliam no era británico sino estadounidense.

El nombre del lenguaje no tiene nada que ver con reptiles

```
string sInput;
int iLength, iN;
double dblTemp;
bool again = true;

while (again) {
    iN = -1;
    again = false;
    getline(cin, sInput);
    system("cls");
    stringstream(sInput) >> dblTemp;
    iLength = sInput.length();
    if (iLength < 4) {
        again = true;
        continue;
    }
    if (sInput[iLength - 3] != '.') {
```

Interpretes

Anaconda, Spider, Jupyter

- Existen hoy en día muchos interpretes de Python, en el caso del trabajo en ingeniería es recomendable Spyder dentro del entorno Anaconda, o Jupyter Notebook para la creación scripts de diseño y texto



Visual Studio Code es un editor de código fuente desarrollado por Microsoft para Windows , Linux y macOS. Incluye soporte para la depuración, control integrado de Git, resaltado de sintaxis, finalización inteligente de código, fragmentos y refactorización de código. También es personalizable, por lo que los usuarios pueden cambiar el tema del editor, los atajos de teclado y las preferencias. Es gratuito y de código abierto, aunque la descarga oficial está bajo software privativo e incluye características personalizadas por Microsoft.



GDB Compiler

- El compilador online sencillo que nos permite compilar en múltiples lenguajes.



Google Colaboratory

Colaboratory, o "**Colab**" para abreviar, es un producto de **Google** Research. Permite a cualquier usuario escribir y ejecutar código arbitrario de Python en el navegador. Es especialmente adecuado para tareas de aprendizaje automático, análisis de datos y educación.





Thonny Python IDE

- Es el IDE que viene integrado en Raspberry, no contiene tantas herramientas como podemos encontrarnos en los anteriores entornos dado que se utiliza en sistemas de bajos recursos.

The screenshot shows the Thonny Python IDE interface. The code editor displays a script named 'prueba.py' with the following content:

```
def numeroCorrecto(numero):
    if numero == 5:
        print("Es el numero correcto")
        return 1
    else:
        if numero > 5:
            print("Se ha pasado")
            return 0
        else:
            print("Se ha quedado corto")
            return 0

numeroCorrecto(7)
```

The Variables pane shows a single variable entry:

Name	Value ID
numeroCorrecto	0x7F6EA8E556A8

The Heap pane lists memory locations and their values:

ID	Value
0x7F6EA8E556A8	<function numeroCorrecto at 0x7f6ea8e556a8>
0x7F6EB195C8F0	None
0x7F6EB1A63CF0	'prueba.py'
0x7F6EB1AC6430	'__main__'
0x7F6EB1AD3D30	<_frozen_importlib_external.SourceFile>
0x7F6EB1B17870	{}
0x7F6EB1B99638	<module 'builtins' (built-in)>

The Object inspector pane shows details for the 'numeroCorrecto' module:

id:	0x7F6EB1B99638
repr:	<module 'builtins' (built-in)>
type:	<class 'module'>

The AST pane shows the Abstract Syntax Tree for the script:

```
Node
└─ root=Module
    └─ body=[...]
        └─ 0=FunctionDef
            └─ name='numeroCorrecto'
            └─ args=arguments
                └─ args=[...]
```



Tipos de datos

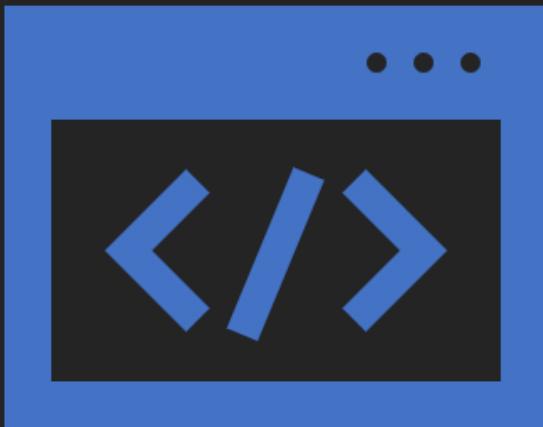
- En Python todos los datos son clases, en el momento de instanciarlo todos son objetos. Esto significa que además del dato en cuestión almacenado tendremos acceso a un montón de métodos asociados al tipo de dato manejando.
- Con el operador `type()` podemos ver que tipo de dato es cada variable y con el operador `dir()` podemos ver los métodos asociados a ese tipo de dato.

Tipos de datos numéricos

Tipo de dato	Descripción	Ejemplo
Int	Números enteros	3
Float	Números decimales en formato IEEE 754	2.6
Complex	Números complejos de la forma $a+bj$	$3+2j$
Bool	Valor booleano que puede tomar dos valores True o False	True False

IMPORTANTE!!!

- Las variables en Python **no contienen los datos, sino que apuntan a los datos.**
- Esta es la forma de trabajo de los **punteros**, lo que hace que el lenguaje sea más eficiente.



Tipos de datos de secuencias

Tipo de dato	Descripción	Ejemplo
string	Cadenas de caracteres	"Hola!" ; 'Hola': """Hola"""; ''Hola''
Tuplas	Una <i>tupla</i> es una secuencia de items ordenada e inmutable .	(200,100,5,4)
Listas	Una lista es una secuencia ordenada de elementos mutable .	[200,100,5,4]
diccionarios	Es una colección no-ordenada de valores que son accedidos a traves de una clave	diccionario = {'nombre' : 'Carlos', 'edad' : 22, 'cursos': 'Python' }
Conjuntos/ Sets y frozenset	<i>set</i> y <i>frozenset</i> representan conjuntos matématicos	{a,b,c,d}

Operaciones en Python

- Podemos utilizar el lenguaje realmente como una calculadora, de esta manera vemos el resultado de la suma de dos números

```
>>> 3 + 2  
5
```

Cuales son las operaciones que puedo realizar y cuales los operadores.

Operación	Operador	Ejemplo	Resultado	Descripción
Suma	+	1+2	3	Devuelve la suma de los operandos
Resta	-	3-2	1	Devuelve la diferencia entre los operandos
Multiplicación	*	2*3	6	Devuelve el producto de los operandos
Exponente	**	2**3	8	Devuelve el resultado de elevar el primer número al segundo (Exponente)
División	/	3.5/2	1.75	Realiza la división real de los números No obstante hay que tener en cuenta que si utilizamos dos operandos enteros, Python determinará que quiere que la variable resultado también sea un entero, por lo que el resultado de, por ejemplo, 3 / 2 y 3 // 2 sería el mismo: 1.
División entera	//	3.5 // 2	1.0	Devuelve la parte entera de la división de dos números
Operador modulo	%	32%10	2	Devuelve el resto de la división entre los dos operandos



Precedencia,
Establece como se
ejecutará una operación
en función de la
ubicación de los
operандos

Exponente: **

Negación: -

Multiplicación, División, División
entera, Módulo: *, /, //, %

Suma, Resta: +, -



Ejemplo

¿Cual será el resultado de la siguiente operación?

```
>>> 2**1/12
```

Resultado

```
>>> 2**1/12  
0.1666666666666666  
>>>
```

Se recomienda el uso de paréntesis para no caer en errores a la hora de esperar un resultado

Operadores de asignación

Asignaciones	
$a += b$	$a = a + b$
$a -= b$	$a = a - b$
$a *= b$	$a = a * b$
$a /= b$	$a = a / b$
$a //= b$	$a = a // b$
$a %= b$	$a = a \% b$
$a **= b$	$a = a ** b$

Operadores de comparación

Comparación	
$a == b$	a igual a b
$a != b$	a distinto de b
$a < b$	a menor a b
$a > b$	a mayor a b
$a <= b$	a menor o igual que b
$a >= b$	a mayor o igual que b

Operadores de identidad y pertenencia

Identidad y pertenencia	
a is b	a es el mismo objeto que b
a is not b	a no es el mismo objeto que b
a in b	a está contenido en b
a not in b	a no está contenido en b



Funciones integradas

- Python tiene algunas funciones integradas para poder realizar operaciones matemáticas

La función integrada divmod()

- La función integrada `divmod(x, y)` devuelve una tupla formada por el cociente y el resto de la división de x entre y .

```
>>> divmod(13, 4)  
(3, 1)
```

La función integrada round()

- Para redondear un número (por ejemplo, cuando se muestra al usuario el resultado final de un cálculo).

```
>>> round(4.35)  
4
```

La función integrada round()

- Si se escriben dos argumentos, siendo el segundo un número entero, la función integrada round() devuelve el primer argumento redondeado en la posición indicada por el segundo argumento.

```
>>> round(4.3527, 2)
```

```
4.35
```

```
>>> round(4.3527, 3)
```

```
4.353
```

Si se piden más decimales de los que tiene el número, se obtiene el primer argumento, sin cambios

La función integrada round()

- Si el segundo argumento es 0 y el primero es un número entero, el resultado es entero:

```
>>> round(4.3527, 0)  
4
```

La función integrada round()

- Si el segundo argumento es negativo, se redondea a decenas, centenas, etc.

```
>>> round(43527, -1)
43530
>>> round(43527, -2)
43500
>>> round(43527, -3)
44000
>>> round(43527, -4)
40000
>>> round(43527, -5)
0
```

Valor absoluto: abs()

- La función integrada `abs()` calcula el valor absoluto de un número, es decir, el valor sin signo.

```
>>> abs(-6)
```

```
6
```

```
>>> abs(7)
```

```
7
```

Máximo: max()

La función integrada `max()` calcula el valor máximo de un conjunto de valores (numéricos o alfabéticos). En el caso de cadenas, el valor máximo corresponde al último valor en orden alfabético, sin importar la longitud de la cadena.

```
>>> max(4, 5, -2, 8, 3.5, -10)
8
>>> max("David", "Alicia", "Tomás", "Emilio")
'Tomás'
```

Mínimo: min()

- La función integrada min() calcula el valor mínimo de un conjunto de valores (numéricos o alfabéticos). En el caso de cadenas, el valor mínimo corresponde al primer valor en orden alfabético, sin importar la longitud de la cadena.

```
>>> min(4, 5, -2, 8, 3.5, -10)
-10
>>> min("David", "Alicia", "Tomás", "Emilio")
'Alicia'
```

Las vocales acentuadas, la letra ñ o ç se consideran posteriores al resto de vocales y consonantes

Suma: sum()

- La función integrada sum() calcula la suma de un conjunto de valores. El conjunto de valores debe ser un tipo de datos iterable (tupla, rango, lista, conjunto o diccionario).

```
>>> sum((1, 2, 3, 4, 5))  
15
```

Ordenación: sorted()

- La función integrada sorted() ordena un conjunto de valores. El conjunto de valores debe ser un tipo de datos iterable (tupla, rango, lista, conjunto o diccionario). El conjunto de valores no se modifica, la función devuelve una lista con los elementos ordenados.

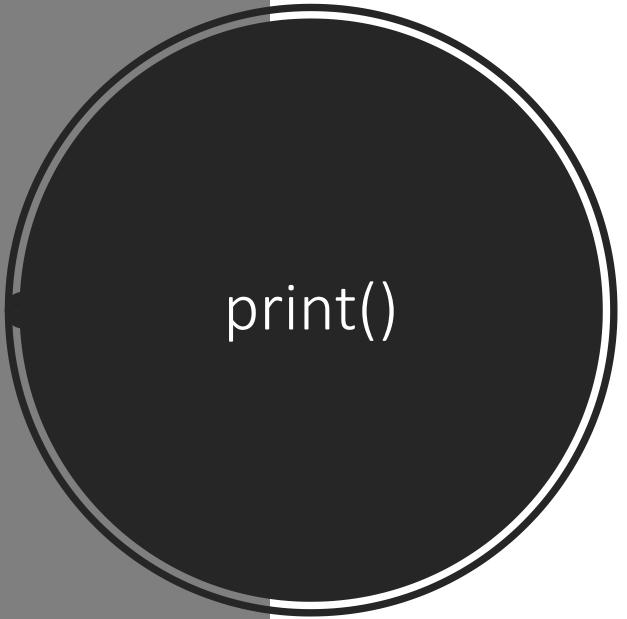
```
>>> sorted((10, 2, 8, -3, 6))  
[-3, 2, 6, 8, 10]
```

```
>>> sorted(("David", "Alicia", "Tomás", "Emilio"))  
['Alicia', 'David', 'Emilio', 'Tomás']
```

Cadenas de caracteres

- Además de números, Python puede manipular cadenas de texto, las cuales pueden ser expresadas de distintas formas.
- Pueden estar encerradas en comillas simples ('...') o dobles ("...") con el mismo resultado. \ puede ser usado para escapar comillas:

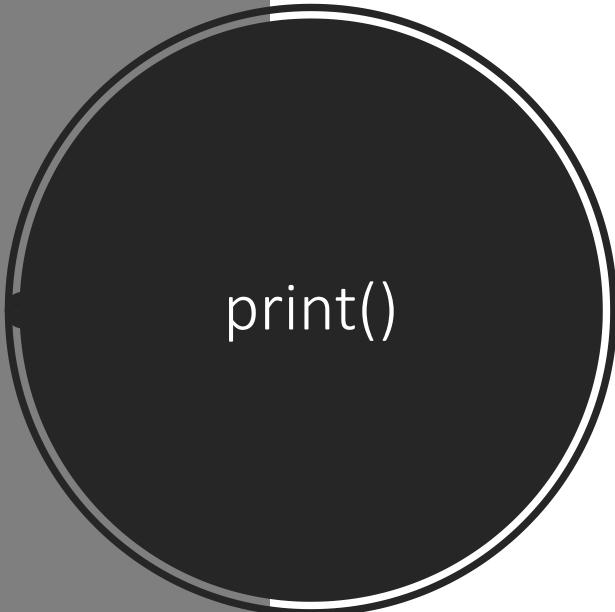
```
>>> 'huevos y pan' # comillas simples  
'huevos y pan'  
>>> 'doesn\'t' # usa \' para escapar  
comillas simples...  
"doesn't"
```



print()

- En el intérprete interactivo, la salida de cadenas está encerrada en comillas y los caracteres especiales son escapados con barras invertidas. Aunque esto a veces luzca diferente de la entrada (las comillas que encierran pueden cambiar), las dos cadenas son equivalentes. La cadena se encierra en comillas dobles si la cadena contiene una comilla simple y ninguna doble, de lo contrario es encerrada en comillas simples. La función **print()** produce una salida más legible, omitiendo las comillas que la encierran e imprimiendo caracteres especiales y escapados:

```
>>> '"Isn\'t," she said.'  
'Isn\'t," she said.'  
>>> print('"Isn\'t," she said.)  
"Isn't," she said.  
>>> s = 'Primera línea.\nSegunda línea.' # \n  
significa nueva línea  
>>> s # sin print(), \n es incluido en la salida  
'Primera línea.\nSegunda línea.'  
>>> print(s) # con print(), \n produce una nueva  
línea  
Primera línea.  
Segunda línea.
```



print()

- Si no querés que los caracteres antepuestos por \ sean interpretados como caracteres especiales, podés usar *cadenas crudas* agregando una r antes de la primera comilla:

```
>>> print('C:\algun\nombre') # aquí \n significa nueva linea!
```

```
C:\algun  
ombre
```

```
>>> print(r'C:\algun\nombre') # nota la r  
antes de la comilla
```

```
C:\algun\nombre
```

Cadenas de textos

- Las cadenas de texto se pueden *indexar* (subíndices), el primer carácter de la cadena tiene el índice 0. No hay un tipo de dato para los caracteres; un carácter es simplemente una cadena de longitud uno:

```
>>> palabra = 'Python'  
>>> palabra[0] # carácter en la posición 0  
'P'  
>>> palabra[5] # carácter en la posición 5  
'n'
```

Cadenas de textos

- Los índices quizás sean números negativos, para empezar a contar desde la derecha:

```
>>> palabra[-1] # último carácter  
'n'  
>>> palabra[-2] # ante último carácter  
'o'  
>>> palabra[-6]  
'P'
```

Cadenas de texto

- Además de los índices, las *rebanadas* también están soportadas. Mientras que los índices son usados para obtener caracteres individuales, las *rebanadas* te permiten obtener sub-cadenas:

```
>>> palabra[0:2] # caracteres desde la  
posición 0 (incluída) hasta la 2 (excluída)  
'Py'  
>>> palabra[2:5] # caracteres desde la  
posición 2 (incluída) hasta la 5 (excluída)  
'tho'
```

Cadena de textos

- Las cadenas de texto pueden ser concatenadas (pegadas juntas) con el operador + y repetidas con *:

```
>>> # 3 veces 'un', seguido de 'ium'  
>>> 3 * 'un' + 'ium'  
'unununium'
```

Esto solo funciona con dos literales, no con variables ni expresiones:

```
>>> prefix = 'Py'  
>>> prefix 'thon' # no se puede concatenar  
una variable y una cadena literal  
...  
SyntaxError: invalid syntax  
>>> ('un' * 3) 'ium'  
...  
SyntaxError: invalid syntax
```

Listas

- Python tiene varios tipos de datos *compuestos*, usados para agrupar otros valores. El más versátil es la *lista*, la cual puede ser escrita como una lista de valores separados por coma (ítems) entre corchetes. Las listas pueden contener ítems de diferentes tipos, pero usualmente los ítems son del mismo tipo:

```
>>> cuadrados = [1, 4, 9, 16, 25]
>>> cuadrados
[1, 4, 9, 16, 25]
```

Listas

- Como las cadenas de caracteres (y todos los otros tipos *sequence* integrados), las listas pueden ser indexadas y rebanadas:

```
>>> cuadrados[0] # indices  
retornan un ítem  
1  
>>> cuadrados[-1]  
25  
>>> cuadrados[-3:] # rebanadas  
retornan una nueva lista  
[9, 16, 25]
```

Listas

- Las listas también soportan operaciones como concatenación:

```
>>> cuadrados + [36, 49, 64, 81, 100]  
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Listas

- A diferencia de las cadenas de texto, que son *immutable*, las listas son un tipo *mutable*, es posible cambiar un su contenido:

```
>>> cubos = [1, 8, 27, 65, 125] # hay algo  
mal aquí  
>>> 4 ** 3 # el cubo de 4 es 64, no 65!  
64  
  
>>> cubos[3] = 64 # reemplazar el valor  
incorrecto  
>>> cubos  
[1, 8, 27, 64, 125]
```

Listas

- También podés agregar nuevos ítems al final de la lista, usando el *método append()*:

```
>>> cubos.append(216) # agregar el cubo de  
6  
>>> cubos.append(7 ** 3) # y el cubo de 7  
>>> cubos  
[1, 8, 27, 64, 125, 216, 343]
```

Función len()

- Devuelve la longitud de una cadena de texto o de una lista

```
>>> letras = ['a', 'b', 'c', 'd']
>>> len(letras)
4
```

Es posible anidar listas (crear listas que contengan otras listas), por ejemplo:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

Bucles y ciclos

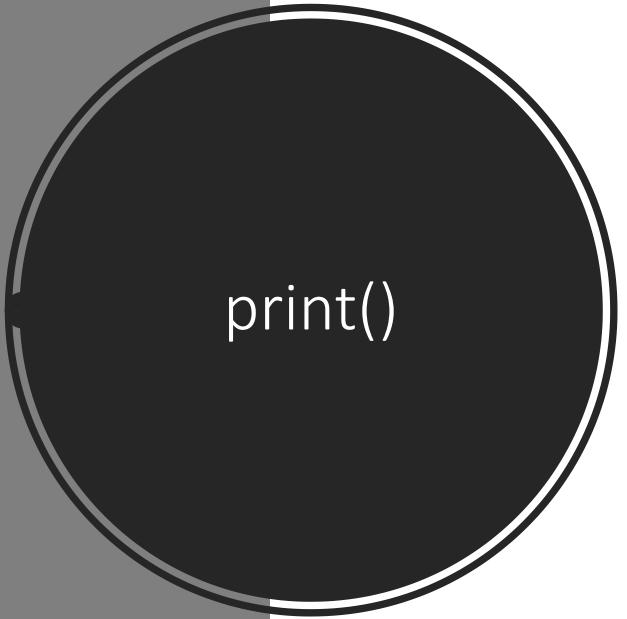
- Por supuesto, podemos usar Python para tareas más complicadas que sumar dos y dos. Por ejemplo, podemos escribir una subsecuencia inicial de la serie de *Fibonacci* así:

```
>>> # Series de Fibonacci:  
... # la suma de dos elementos define el  
siguiente  
... a, b = 0, 1  
>>> while b < 10:  
...     print(b)  
...     a, b = b, a+b
```

Este ejemplo introduce varias características nuevas:

- La primer línea contiene una *asignación múltiple*: las variables a y b toman en forma simultanea los nuevos valores 0 y 1. En la última línea esto se vuelve a usar, demostrando que las expresiones a la derecha son evaluadas antes de que suceda cualquier asignación. Las expresiones a la derecha son evaluadas de izquierda a derecha.
- El bucle **while** se ejecuta mientras la condición (aquí: $b < 10$) sea verdadera. En Python, como en C, cualquier entero distinto de cero es verdadero; cero es falso. La condición también puede ser una cadena de texto o una lista, de hecho cualquier secuencia; cualquier cosa con longitud distinta de cero es verdadero, las secuencias vacías son falsas. La prueba usada en el ejemplo es una comparación simple. Los operadores estándar de comparación se escriben igual que en C: < (menor qué), > (mayor qué), == (igual a), <= (menor o igual qué), >= (mayor o igual qué) y != (distinto a).

- El *cuerpo* del bucle está *indentado*: la sangría es la forma que usa Python para agrupar declaraciones. En el intérprete interactivo debés teclear un tab o espacio(s) para cada línea indentada. En la práctica vas a preparar entradas más complicadas para Python con un editor de texto; todos los editores de texto decentes tienen la facilidad de agregar la sangría automáticamente.



print()

- La función **print()** escribe el valor de el o los argumentos que se le pasan. Difiere de simplemente escribir la expresión que se quiere mostrar (como hicimos antes en los ejemplos de la calculadora) en la forma en que maneja múltiples argumentos, cantidades en punto flotante, y cadenas. Las cadenas de texto son impresas sin comillas, y un espacio en blanco es insertado entre los elementos, así podés formatear cosas de una forma agradable:

```
>>> i = 256*256
>>> print('El valor de i es', i)
El valor de i es 65536
```

Tomando decisiones

- Tal vez el tipo más conocido de sentencia sea el **if**. Por ejemplo:

Puede haber cero o más bloques **elif**, y el bloque **else** es opcional. La palabra reservada '**elif**' es una abreviación de 'else if', y es útil para evitar un sangrado excesivo. Una secuencia **if** ... **elif** ... **elif** ... sustituye las sentencias **switch** o **case** encontradas en otros lenguajes.

```
>>> x = int(input("Ingresa un entero,  
por favor: "))  
Ingresa un entero, por favor: 42  
>>> if x < 0:  
...     x = 0  
...     print('Negativo cambiado a cero')  
... elif x == 0:  
...     print('Cero')  
... elif x == 1:  
...     print('Simple')  
... else:  
...     print('Más')  
...  
'Mas'
```

Sentencia for

- La sentencia **for** en Python difiere un poco de lo que uno puede estar acostumbrado en lenguajes como C. En lugar de darle al usuario la posibilidad de definir tanto el paso de la iteración como la condición de fin (como en C), la sentencia **for** de Python itera sobre los ítems de cualquier secuencia (una lista o una cadena de texto), en el orden que aparecen en la secuencia. Por ejemplo

```
>>> # Midiendo cadenas de texto
... palabras = ['gato', 'ventana',
'defenestrado']
>>> for p in palabras:
...     print(p, len(p))
...
gato 4
ventana 7
defenestrado 12
```

Recomendaciones

- Si necesitás modificar la secuencia sobre la que estás iterando mientras estás adentro del ciclo (por ejemplo para borrar algunos ítems), se recomienda que hagas primero una copia. Iterar sobre una secuencia no hace implícitamente una copia.
- La notación de rebanada es especialmente conveniente para esto:

```
>>> for p in palabras[:]: # hace una copia  
    por rebanada de toda la lista  
    ... if len(p) > 6:  
    ... palabras.insert(0, p)  
    ...  
  
>>> palabras  
['defenestrado', 'ventana', 'gato',  
'ventana', 'defenestrado']
```

Iteraciones sobre secuencias números

- Si se necesita iterar sobre una secuencia de números, es apropiado utilizar la función integrada **range()**, la cual genera progresiones aritméticas:

```
>>> for i in range(5):  
...     print(i)  
...  
0  
1  
2  
3  
4
```

- El valor final dado nunca es parte de la secuencia; `range(10)` genera 10 valores, los índices correspondientes para los ítems de una secuencia de longitud 10. Es posible hacer que el rango empiece con otro número, o especificar un incremento diferente (incluso negativo; algunas veces se lo llama 'paso'):

```
range(5, 10)
5 through 9
range(0, 10, 3)
0, 3, 6, 9
range(-10, -100, -30)
-10, -40, -70
```

Para iterar sobre los índices de una secuencia, podés combinar `range()` y `len()` así:

```
>>> a = ['Mary', 'tenia', 'un',
'corderito']
>>> for i in range(len(a)):
... print(i, a[i])
...
0 Mary
1 tenia
2 un
3 corderito
```

Las sentencias , , y en lazos

- La sentencia **break**, como en C, termina el lazo **for** o **while** más anidado.
- Las sentencias de lazo pueden tener una cláusula **else** que es ejecutada cuando el lazo termina, luego de agotar la lista (con **for**) o cuando la condición se hace falsa (con **while**), pero no cuando el lazo es terminado con la sentencia **break**. Se exemplifica en el siguiente lazo, que busca números primos:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'es igual a', x, '*', n/x)
...             break
... else:
...     # sigue el bucle sin encontrar un factor
...     print(n, 'es un numero primo')
...
2 es un numero primo
3 es un numero primo
4 es igual a 2 * 2
5 es un numero primo
6 es igual a 2 * 3
7 es un numero primo
8 es igual a 2 * 4
9 es igual a 3 * 3
```

(Sí, este es el código correcto. Fijate bien: el `else` pertenece al ciclo `for`, **no** al `if`.)

Declaración continue:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Encontré un número par", num)
...         continue
...     print("Encontré un número", num)

Encontré un número par 2
Encontré un número 3
Encontré un número par 4
Encontré un número 5
Encontré un número par 6
Encontré un número 7
Encontré un número par 8
Encontré un número 9
```

Definiendo funciones

- Podemos crear una función que escriba la serie de Fibonacci hasta un límite determinado:

```
>>> def fib(n): # escribe la serie de Fibonacci hasta n
...     """Escribe la serie de Fibonacci hasta n."""
...     a=1
...     a,b=2,a
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Ahora llamamos a la funcion que acabamos de definir:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
La
```

Funciones

- La palabra reservada **def** se usa para *definir* funciones. Debe seguirle el nombre de la función y la lista de parámetros formales entre paréntesis. Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente, y deben estar indentado.
- La primer sentencia del cuerpo de la función puede ser opcionalmente una cadena de texto literal; esta es la cadena de texto de documentación de la función.

Más sobre listas

`list.append (x)`

- Agrega un ítem al final de la lista. Equivale a `a[len(a):] = [x]`.

`list.extend (iterable)`

- Extiende la lista agregándole todos los ítems del iterable. Equivale a `a[len(a):] = iterable`.

`list.insert (i, x)`

- Inserta un ítem en una posición dada. El primer argumento es el índice del ítem delante del cual se insertará, por lo tanto `a.insert(0, x)` inserta al principio de la lista, y `a.insert(len(a), x)` equivale a `a.append(x)`.

`list.remove (x)`

- Quita el primer ítem de la lista cuyo valor sea `x`. Es un error si no existe tal ítem.

`list.pop ([, i])`

- Quita el ítem en la posición dada de la lista, y lo devuelve. Si no se especifica un índice, `a.pop()` quita y devuelve el último ítem de la lista. (Los corchetes que encierran a `i` en la firma del método denotan que el parámetro es opcional, no que deberías escribir corchetes en esa posición. Verás esta notación con frecuencia en la Referencia de la Biblioteca de Python.)

Más sobre listas

`list.clear ()`

- Quita todos los elementos de la lista. Equivalente a `del a[:]`.

`list.index (x[, start[, end]])`

- Devuelve un índice basado en cero en la lista del primer ítem cuyo valor sea `x`. Levanta una excepción `ValueError` si no existe tal ítem. Los argumentos opcionales `start` y `end` son interpretados como la notación de rebanadas y se usan para limitar la búsqueda a una subsecuencia particular de la lista. El index returned se calcula de manera relativa al inicio de la secuencia completa en lugar de con respecto al argumento `start`.

`list.count (x)`

- Devuelve el número de veces que `x` aparece en la lista.

`list.sort (key=None, reverse=False)`

- Ordena los ítems de la lista in situ (los argumentos pueden ser usados para personalizar el orden de la lista, ve `sorted()` para su explicación).

`list.reverse ()`

- Invierte los elementos de la lista in situ.

`list.copy ()`

- Devuelve una copia superficial de la lista. Equivalente a `a[:]`.

Ejemplos

```
>>> frutas = ['naranja', 'manzana', 'pera', 'banana', 'kiwi', 'manzana',
'banana']
>>> frutas.count('manzana')
2
>>> frutas.count('mandarina')
0
>>> frutas.index('banana')
3
>>> frutas.index('banana', 4) # Find next banana starting a position 4
6
>>> frutas.reverse()
>>> frutas
['banana', 'manzana', 'kiwi', 'banana', 'pera', 'manzana', 'naranja']
>>> frutas.append('uva')
>>> frutas
['banana', 'manzana', 'kiwi', 'banana', 'pera', 'manzana', 'naranja', 'uva']
>>> frutas.sort()
>>> frutas
['manzana', 'manzana', 'banana', 'banana', 'uva', 'kiwi', 'naranja', 'pera']
>>> frutas.pop()
'pera'
```

Usando listas como pilas (LIFO)

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Usando listas como colas (FIFO)

```
>>> from collections import deque
>>> queue = deque(["Eric", "John",
"Michael"])
>>> queue.append("Terry") # llega Terry
>>> queue.append("Graham") # llega Graham
>>> queue.popleft() # el primero en llegar
ahora se va
'Eric'
>>> queue.popleft() # el segundo en llegar
ahora se va
'John'
>>> queue # el resto de la cola en orden de
llegada
['Michael', 'Terry', 'Graham']
```

Tuplas y secuencias

- Vimos que las listas y cadenas tienen propiedades en común, como el indizado y las operaciones de seccionado. Estas son dos ejemplos de datos de tipo *secuencia*. Como Python es un lenguaje en evolución, otros datos de tipo secuencia pueden agregarse. Existe otro dato de tipo secuencia estándar: la *tupla*.

```
>>> t = 12345, 54321, 'hola!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hola!')
>>> # Las tuplas pueden anidarse:
```

```
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hola!'), (1, 2, 3, 4, 5))
>>> # Las tuplas son inmutables:
... t[0] = 88888
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
item assignment
>>> # pero pueden contener objetos
       mutables:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Conjuntos

- Python también incluye un tipo de dato para *conjuntos*. Un conjunto es una colección no ordenada y sin elementos repetidos.
- Los usos básicos de éstos incluyen verificación de pertenencia y eliminación de entradas duplicadas. Los conjuntos también soportan operaciones matemáticas como la unión, intersección, diferencia, y diferencia simétrica. Las llaves o la función **set()** pueden usarse para crear conjuntos. Notá que para crear un conjunto vacío tenés que usar **set()**, no **{}**; esto último crea un diccionario vacío, una estructura de datos que discutiremos en la sección siguiente.

```
>>> canasta = {'manzana', 'naranja', 'manzana', 'pera', 'naranja', 'banana'}
>>> print fruta # muestra que se removieron los duplicados
{'pera', 'manzana', 'banana', 'naranja'}
>>> 'naranja' in canasta # verificación de pertenencia rápida
True
>>> 'yerba' in canasta
False
>>> # veamos las operaciones para las letras únicas de dos palabras
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a # letras únicas en a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b # letras en a pero no en b
{'r', 'b', 'd'}
>>> a | b # letras en a o en b o en ambas
{'a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'}
>>> a & b # letras en a y en b
{'a', 'c'}
>>> a ^ b # letras en a o b pero no en ambos
{'b', 'd', 'm', 'l', 'r', 'z'}
```

Diccionarios

- Otro tipo de dato útil incluído en Python es el *diccionario*. Los diccionarios se encuentran a veces en otros lenguajes como "memorias asociativas" o "arreglos asociativos". A diferencia de las secuencias, que se indexan mediante un rango numérico, los diccionarios se indexan con *claves*, que pueden ser cualquier tipo inmutable; las cadenas y números siempre pueden ser claves. Las tuplas pueden usarse como claves si solamente contienen cadenas, números o tuplas; si una tupla contiene cualquier objeto mutable directa o indirectamente, no puede usarse como clave. No podés usar listas como claves, ya que las listas pueden modificarse usando asignación por índice, asignación por sección, o métodos como **append()** y **extend()**.

Ejemplo de diccionario

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'irv': 4127, 'guido': 4127}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Clases

- Las clases introducen un poquito de sintaxis nueva, tres nuevos tipos de objetos y algo de semántica nueva.

```
class Clase:  
<declaracin-1>  
.  
.  
.  
<declaracin-N>
```

Clases métodos constructores

- Por supuesto, el método `__init__()` puede tener argumentos para mayor flexibilidad. En ese caso, los argumentos que se pasen al operador de instanciación de la clase van a parar al método `__init__()`. Por ejemplo:

```
>>> class Complejo:  
...     def __init__(self, partereal,  
parteimaginaria):  
...         self.r = partereal  
...         self.i = parteimaginaria  
...  
>>> x = Complejo(3.0, -4.5)  
>>> x.r, x.i  
(3.0, -4.5)
```

Clases ejemplos

```
class Perro:  
    tipo = 'canino' # variable de clase compartida por todas las instancias  
    def __init__(self, nombre):  
        self.nombre = nombre # variable de instancia única para la instancia  
>>> d = Perro('Fido')  
>>> e = Perro('Buddy')  
>>> d.tipo # compartido por todos los perros  
'canino'  
>>> e.tipo # compartido por todos los perros  
'canino'  
>>> d.nombre # único para d  
'Fido'  
>>> e.nombre # único para e  
'Buddy'
```

Clases ejemplos

```
class Perro:  
    def __init__(self, nombre):  
        self.nombre = nombre  
        self.trucos = [] # crea una nueva lista vacía para cada perro  
    def agregar_truco(self, truco):  
        self.trucos.append(truco)  
>>> d = Perro('Fido')  
>>> e = Perro('Buddy')  
>>> d.agregar_truco('girar')  
>>> e.agregar_truco('hacerse el muerto')  
>>> d.trucos  
['girar']  
>>> e.trucos  
['hacerse el muerto']
```

Inputs

Nombre = 'Isi'

Edad = 36

```
Nombre=input("Ingrese su nombre: ")
```

```
Edad=input("Ingrese su edad:")
```

```
print('La edad de %s es: %s años ' % (Nombre, Edad))
```

```
print('La edad de {} es: {} años '.format(Nombre, Edad))
```

```
print(f'La edad de {Nombre} es: {Edad} años')
```

Inputs

bolsas = 3

manzanas_por_bolsa= 12

```
print(f'Hay un total de {bolsas * manzanas_por_bolsa} manzanas entre  
todas las bolsas')
```

Inputs

```
usuario = {'nombre': 'Israel Pavelek', 'ocupacion': 'Docente'}
```

```
print(f"{{usuario['nombre']}} es {{usuario['ocupacion']}}")
```

Inputs

bolsas = 3

manzanas_por_bolsa= 12

```
print(f'Hay un total de {bolsas * manzanas_por_bolsa} manzanas entre  
todas las bolsas')
```

Inputs

```
Nombre = 'Isi'
```

```
Edad = 36
```

```
Ocupacion = 'Docente'
```

```
msg = (  
    f'Nombre: {Nombre}\n'  
    f'Edad: {Edad}\n'  
    f'Ocupacion: {Ocupacion}'  
)
```

```
print(msg)
```

Modulos Matemáticos

- El módulo **math** permite el acceso a las funciones de la biblioteca C subyacente para la matemática de punto flotante:

```
>>> import math  
>>> math.cos(math.pi / 4)  
0.70710678118654757  
>>> math.log(1024, 2)  
10.0
```

Modulo para enviar correo

```
import smtplib

gmail_user = 'ipavelek@gmail.com'
gmail_password = 'xxxxxxxxxx'

sent_from = gmail_user
to = ['ipavelek@etrr.edu.ar']
subject = 'Hola'
body = 'Que onda?'

email_text = """\
From: %s
To: %s
Subject: %s

%s
""" % (sent_from, ", ".join(to), subject, body)
```

```
try:  
    server = smtplib.SMTP_SSL('smtp.gmail.com', 465)  
    server.ehlo()  
    server.login(gmail_user, gmail_password)  
    server.sendmail(sent_from, to, email_text)  
    server.close()  
  
    print ('Email enviado')  
except:  
    print ('Hubo algun error')
```

Python en ingeniería

Numpy SciPy Matplotlib

```
mirror_mod = modifier_obj
# mirror object to mirror
mirror_mod.mirror_object
operation = "MIRROR_X"
mirror_mod.use_x = True
mirror_mod.use_y = False
mirror_mod.use_z = False
operation == "MIRROR_Y"
mirror_mod.use_x = False
mirror_mod.use_y = True
mirror_mod.use_z = False
operation == "MIRROR_Z"
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True

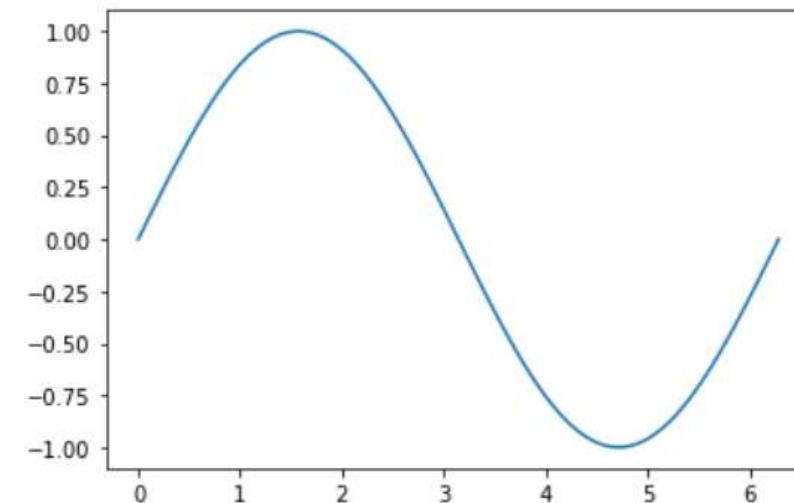
selection at the end -add
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active
("Selected" + str(modifier))
mirror_ob.select = 0
bpy.context.selected_objects
data.objects[one.name].select
int("please select exactly one
- OPERATOR CLASSES ---
```

```
types.Operator):
    X mirror to the selected
    object.mirror_mirror_x"
    or X"
```

```
context):
    ext.active_object is not
```

Python en ciencia

```
import numpy  
from matplotlib import pyplot  
x = numpy.linspace(0, 2*numpy.pi, 100)  
y = numpy.sin(x)  
  
pyplot.plot(x, y)  
pyplot.show()
```



Matplotlib

- **Matplotlib** es una biblioteca para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy. Proporciona una API, **pylab**, diseñada para recordar a la de MATLAB.

Numpy

- Los arrays Numpy son una excelente alternativa a las listas de Python. Algunas de las ventajas clave de los arrays Numpy es que son rápidos, fáciles de trabajar con ellos, y ofrece a los usuarios la oportunidad de realizar cálculos a través de arrays completos.

SciPy

- **SciPy** es una biblioteca libre y de código abierto para Python. Se compone de herramientas y algoritmos matemáticos. Se creó a partir de la colección original de Travis Oliphant, que se componía de módulos de extensión para Python y fue lanzada en 1999 bajo el nombre de Multipack, llamada así por los paquetes *netlib* que reunían a ODEPACK, QUADPACK, y MINPACK.
- SciPy contiene módulos para optimización, álgebra lineal, integración, interpolación, funciones especiales, FFT, procesamiento de señales y de imagen, resolución de ODES y otras tareas para la ciencia e ingeniería.

Manejo de pines en Raspberry

BOARD	GPIO	GPIO	BOARD
01	3.3v DC Power	DC Power 5v	02
03	GPIO02 (SDA1 , I ² C)	DC Power 5v	04
05	GPIO03 (SCL1 , I ² C)	Ground	06
07	GPIO04 (GPIO_GCLK)	(TXD0) GPIO14	08
09	Ground	(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)	(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)	Ground	14
15	GPIO22 (GPIO_GEN3)	(GPIO_GEN4) GPIO23	16
17	3.3v DC Power	(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)	Ground	20
21	GPIO09 (SPI_MISO)	(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)	(SPI_CE0_N) GPIO08	24
25	Ground	(SPI_CE1_N) GPIO07	26
27	ID_SD (I ² C ID EEPROM)	(I ² C ID EEPROM) ID_SC	28
29	GPIO05	Ground	30
31	GPIO06	GPIO12	32
33	GPIO13	Ground	34
35	GPIO19	GPIO16	36
37	GPIO26	GPIO20	38
39	Ground	GPIO21	40

- **Amarillo (2)**: Alimentación a 3.3V.
- **Rojo (2)**: Alimentación a 5V.
- **Naranja (26)**: Entradas / salidas de propósito general. Pueden configurarse como entradas o salidas. Ten presente que el nivel alto es de 3.3V y no son tolerantes a tensiones de 5V.
- **Gris (2)**: Reservados.
- Negro (8)**: Conexión a GND o masa.
- **Azul (2)**: Comunicación mediante el protocolo I2C para comunicarse con periféricos que siguen este protocolo.
- **Verde (2)**: Destinados a conexión para UART para puerto serie convencional.
- **Morado (5)**: Comunicación mediante el protocolo SPI para comunicarse con periféricos que siguen este protocolo.

Los pines GPIO ofrecen una corriente de 3mA.

BOARD	GPIO		GPIO	BOARD
01	3.3v DC Power		DC Power 5v	02
03	GPIO02 (SDA1 , I ² C)		DC Power 5v	04
05	GPIO03 (SCL1 , I ² C)		Ground	06
07	GPIO04 (GPIO_GCLK)		(TXD0) GPIO14	08
09	Ground		(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)		(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)		Ground	14
15	GPIO22 (GPIO_GEN3)		(GPIO_GEN4) GPIO23	16
17	3.3v DC Power		(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)		Ground	20
21	GPIO09 (SPI_MISO)		(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)		(SPI_CE0_N) GPIO08	24
25	Ground		(SPI_CE1_N) GPIO07	26
27	ID_SD (I ² C ID EEPROM)		(I ² C ID EEPROM) ID_SC	28
29	GPIO05		Ground	30
31	GPIO06		GPIO12	32
33	GPIO13		Ground	34
35	GPIO19		GPIO16	36
37	GPIO26		GPIO20	38
39	Ground		GPIO21	40

Modos de trabajo

Existen 2 formas de numerar los pines de la Raspberry Pi, en modo GPIO o en modo BCM.

En el **modo GPIO**, los pines se numeran de forma física por el lugar que ocupan en la placa (representados por el color gris) viene siendo igual para todas las versiones (comenzamos a contar desde arriba a la izquierda y finalizamos abajo a la derecha).

En el **modo BCM**, los pines se numeran por la correspondencia en el chip Broadcom (que es la CPU de la Raspberry Pi).

Biblioteca para el manejo de pines

Salidas

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BOARD)
GPIO.setup(7, GPIO.OUT)

while True:
    GPIO.output(7, True)
    time.sleep(1)
    GPIO.output(7, False)
    time.sleep(1)
```

Biblioteca para el manejo de pines

Entradas

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BOARD)
GPIO.setup(3, GPIO.IN)
GPIO.setup(7, GPIO.OUT)

while True:
    if GPIO.input(3):
        GPIO.output(7, False)
    else:
        GPIO.output(7, True)
```

Biblioteca para el manejo de pines PWM

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BOARD)
GPIO.setup(7, GPIO.OUT)

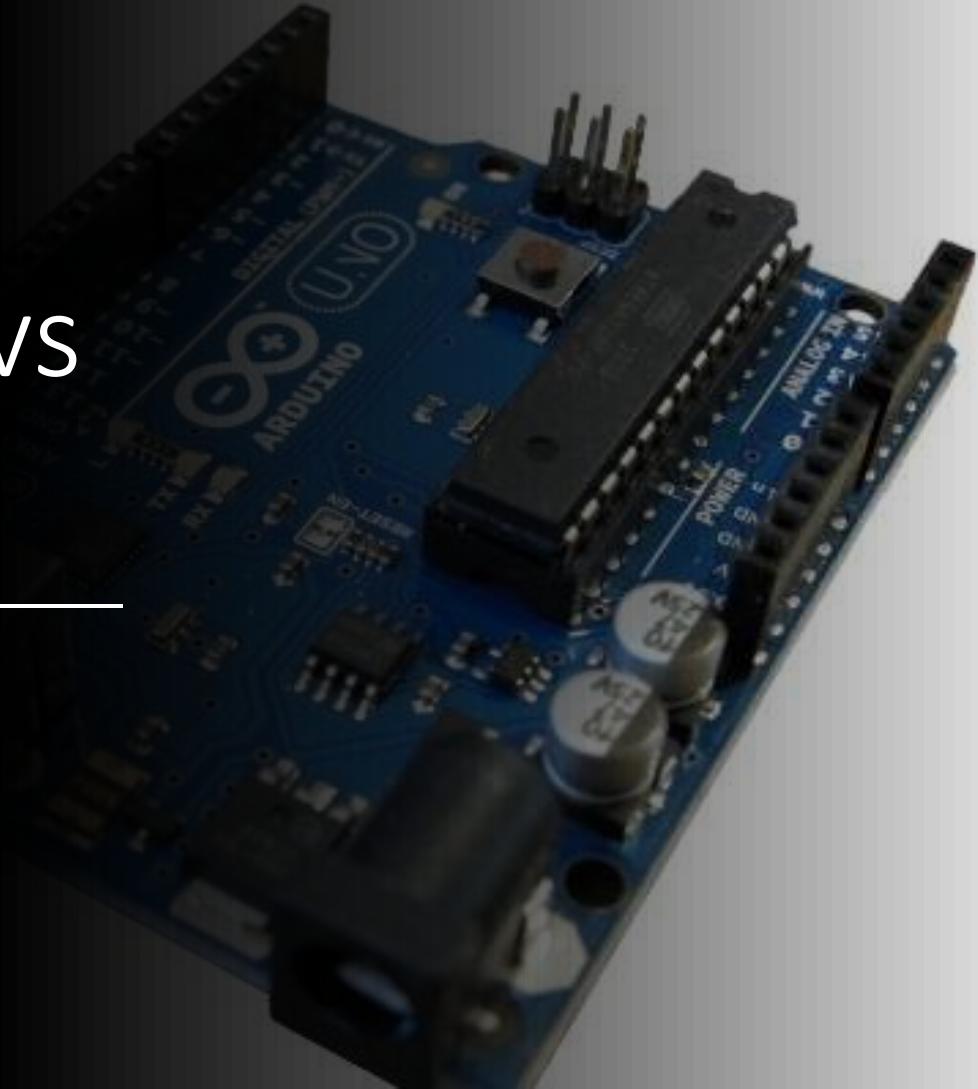
led = GPIO.PWM(7, 100)

while True:
    led.start(0)
    for i in range(0, 100, 25):
        led.ChangeDutyCycle(i)
        time.sleep(0.5)
```

Funciones PWM en Python

- **pi_pwm = GPIO.PWM (Pin no., frequency)**
- **start (Duty Cycle)**
- **ChangeDutyCycle(Duty Cycle)**
- **ChangeFrequency(frequency)**
- **stop()**

Raspberry Pi vs Arduino



¿Qué es Arduino?

- Arduino es una plataforma de creación de electrónica de código abierto basada en hardware y software libre, lo que permite que cualquiera pueda utilizarlos y adaptarlos.

¿Qué es Raspberry?

- **Raspberry Pi es un ordenador de placa simple y bajo costo** desarrollado en Reino Unido por la Raspberry Pi Foundation. Es lo suficientemente potente como para facilitar el aprendizaje y realizar tareas básicas, y también permite programar y compilar programas que se ejecuten en él.

Raspberry Pi vs Arduino

- **Arduino y Raspberry Pi son dos conceptos totalmente diferentes**, por lo que es un poco difícil hacer una comparación entre ambos. Son dos productos con diferentes finalidades, aunque por su versatilidad la imaginación de la comunidad maker haya hecho que ambos sean utilizados para crear todo tipo de proyectos de electrónica.

Prestaciones	Arduino UNO	Lilypad Arduino	Arduino Mega 2560	Arduino Fio	Arduino ADK	Arduino PRO	Arduino Nano
Microcontroller	ATmega328V	ATmega168V ATmega328V	ATmega256	ATmega328P	ATmega2560	ATmega328V	ATmega168 ATmega328
Operating Voltage	5 V	2.7-5.5 V	5V	3.3V	5V	5 V	5 V
Input Voltage (recommended)	7-12V	2.7-5.5 V	7-12V	3.35-12 V	7-12V	7-12V	7-12 V
Input Voltage (limits)	6-20V		6-20V		6-20V	6-20V	6-20 V
Input Voltage for Charge				3.7 - 7 V			
Digital I/O Pins	14 (of which 6 provide PWM output)	14 (of which 6 provide PWM output)	54 (of which 15 provide PWM output)	14 (of which 6 provide PWM output)	54 (of which 15 provide PWM output)	14 (of which 6 provide PWM output)	14 (of which 6 provide PWM output)
Analog Input Pins	6	6	16	8	16	6	8
DC Current per I/O Pin	40 mA	40 mA	40 mA	40 mA	40 mA	40 mA	40 mA
DC Current for 3.3V Pin	50 mA		50 mA		50 mA	50 mA	
Flash Memory	32 KB (ATmega328) of which 0.5 KB used by bootloader	16 KB (of which 2 KB used by bootloader)	256 KB of which 8 KB used by bootloader	32 KB (of which 2 KB used by bootloader)	256 KB of which 8 KB used by bootloader	32 KB (ATmega328) of which 0.5 KB used by bootloader	16 KB (ATmega168) or 32 KB (ATmega328) of which 2 KB used by bootloader
SRAM	2 KB	1 KB	8 KB	2 KB	8 KB	2 KB	1 KB (ATmega168) or 2 KB (ATmega328)
EEPROM	1 KB	512 bytes	4 KB	1 KB	4 KB	1 KB	512 bytes (ATmega168) or 1 KB (ATmega328)
Clock Speed	16 MHz	8 MHz	16 MHz	8 MHz	16 MHz	16 MHz	16 MHz



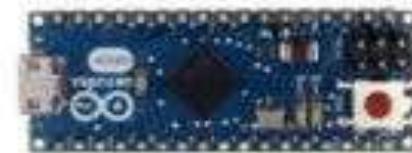
Arduino Uno



Arduino Mega 2560



Arduino Micro



Price Points	\$19.99-\$23.00	\$36.61 - \$39.00	\$19.80 - \$24.38
Dimension	2.7 in x 2.1 in	4 in x 2.1 in	0.7 in x 1.9 in
Processor	Atmega328P	ATmega2560	ATmega32U4
Clock Speed	16MHz	16MHz	16MHz
Flash Memory (kB)	32	256	32
EEPROM (kB)	1	4	1
SRAM (kB)	2	8	2.5
Voltage Level	5V	5V	5V
Digital I/O Pins	14	54	20
Digital I/O with PWM Pins	6	15	7
Analog Pins	6	16	12
USB Connectivity	Standard A/B USB	Standard A/B USB	Micro-USB
Shield Compatibility	Yes	Yes	No
Ethernet/Wi-Fi/Bluetooth	No (a Shield/module can enable it)	No (a Shield/module can enable it)	No

Raspberry Modelos



Model B+



Model B



Model A



Compute Mo



Model A+



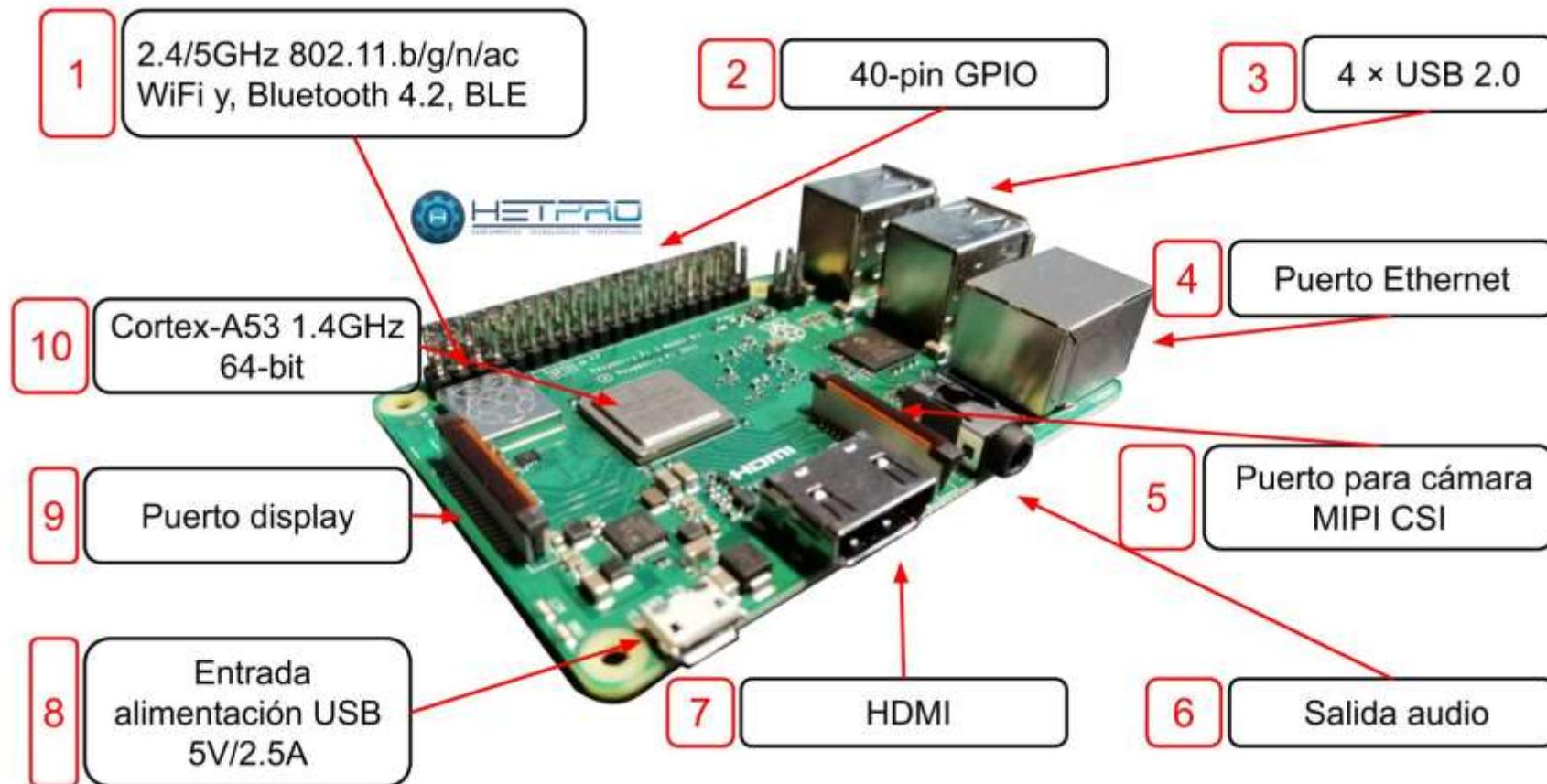
2 Model B

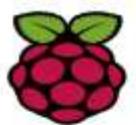


Zero



Raspberry Pi 3B+ características





Comparativa Raspberry Pi

	SoC	CPU	GPU	RAM	USB	V/A	Boot	Red	Alimentación	Tamaño	Fecha	Precio
Model A	Broadcom BCM2835	700MHz ARM1176JZF-S	VideoCore IV	256MB	1	RCA Jack HDMI	SD	No	300mA 1,5w / 5v MicroUSB GPIO	85,6 x 53,98 mm	04/12	25\$
Model A+	Broadcom BCM2835	700MHz ARM1176JZF-S	VideoCore IV	256MB	1	Jack HDMI	uSD	No	400mA 2w / 5v MicroUSB GPIO	65 x 56 mm	11/14	20\$
3 Model A+	Broadcom BCM2837B0	1,4GHz QUAD ARM Cortex-A53	VideoCore IV	512MB	1	Jack HDMI	uSD	Dual-band WiFi, BT	2,5A 12,5w / 5v MicroUSB GPIO	65 x 56 mm	11/18	20\$
Model B	Broadcom BCM2835	700MHz ARM1176JZF-S	VideoCore IV	512MB	2	RCA Jack HDMI	SD	ETH 10/100	700mA 3,5w / 5v MicroUSB GPIO	85,6 x 53,98 mm	04/12	35\$
Model B+	Broadcom BCM2835	700MHz ARM1176JZF-S	VideoCore IV	512MB	4	Jack HDMI	uSD	ETH 10/100	500mA 2,5w / 5v MicroUSB GPIO	85 x 56 mm	07/14	35\$
2 Model B	Broadcom BCM2836	900MHz QUAD ARM Cortex-A7	VideoCore IV	1GB	4	Jack HDMI	uSD	ETH 10/100	800mA 4w / 5v MicroUSB GPIO	85 x 56 mm	02/15	35\$
3 Model B	Broadcom BCM2837	1,2GHz QUAD ARM Cortex-A53	VideoCore IV	1GB	4	Jack HDMI	uSD	ETH 10/100 WiFi, BT	2,5A 12,5w / 5v MicroUSB GPIO	85 x 56 mm	02/16	35\$
3 Model B+	Broadcom BCM2837B0	1,4GHz QUAD ARM Cortex-A53	VideoCore IV	1GB	4	Jack HDMI	uSD	ETH 10/100/300 (USB) Dual-band WiFi BT	2,5A 12,5w / 5v MicroUSB GPIO PoE (HAT)	85 x 56 mm	03/18	35\$
4 Model B	Broadcom BCM2711	1,5GHz QUAD ARM Cortex-A72	VideoCore IV	1,2 o 4GB	2 (2.0) 2 (3.0)	Jack 2 micro HDMI	uSD	ETH 1000 Dual-band WiFi BT	2,5A 12,5w / 5v USB-C GPIO PoE (HAT)	85 x 56 mm	06/19	35\$
Zero	Broadcom BCM2835	1GHz ARM1176JZF-S	VideoCore IV	512MB	1 Micro	Mini HDMI	uSD	No	160mA 0,8w / 5v MicroUSB GPIO	65 x 30 mm	11/15	5\$
Zero W	Broadcom BCM2835	1GHz ARM1176JZF-S	VideoCore IV	512MB	1 Micro	Mini HDMI	uSD	Wifi, BT	160mA 0,8w / 5v MicroUSB GPIO	65 x 30 mm	02/17	10\$

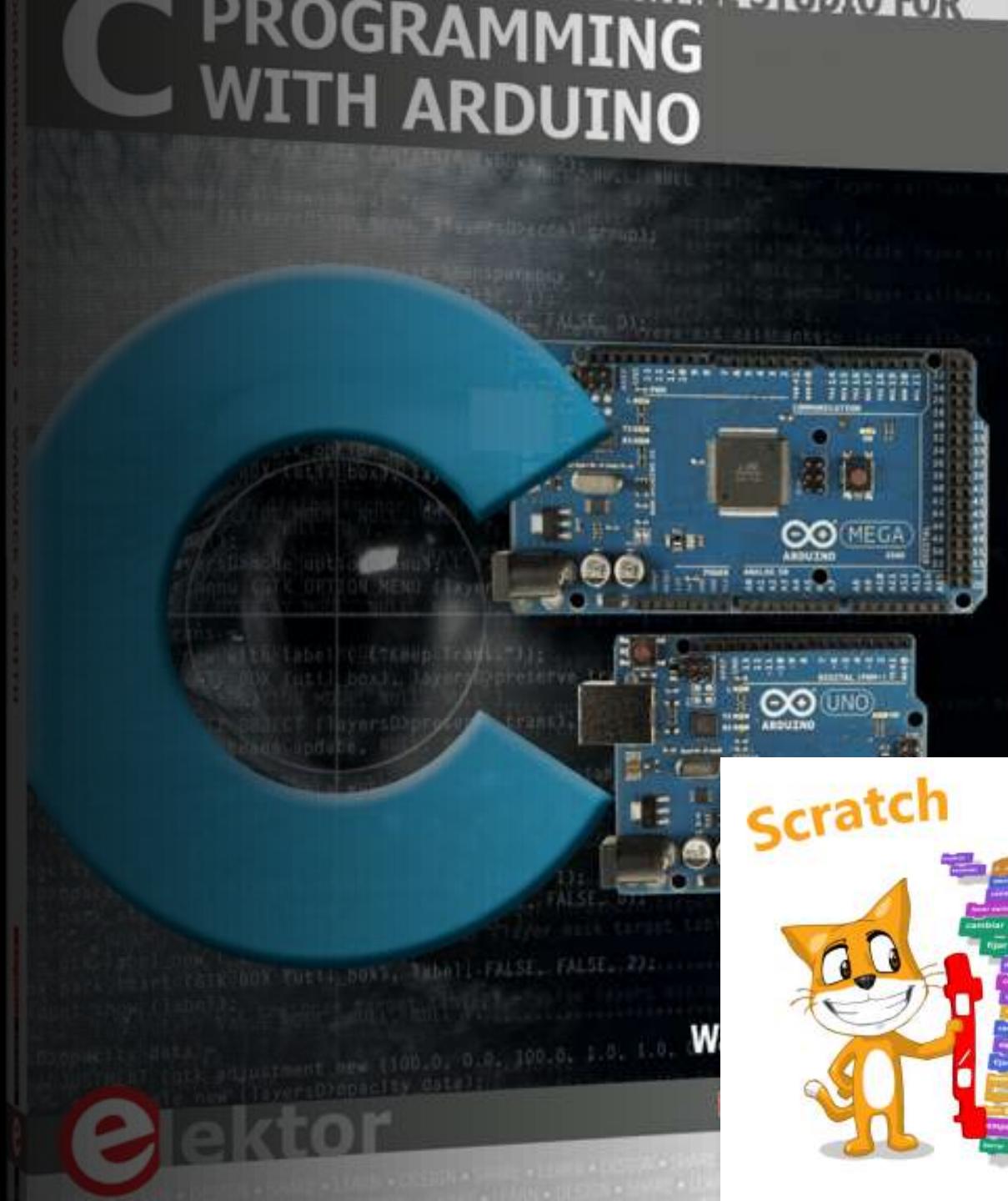


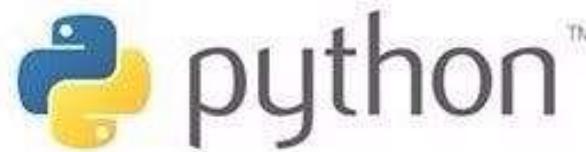
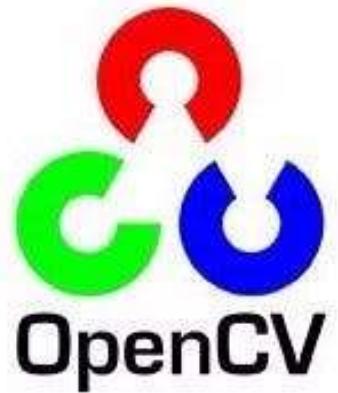
- Fuente de al menos 2,5A micro usb
- Memoria Micro SD clase 10 de al menos 16GB
- Disipadores/coolers
- Gabinete (opcional)



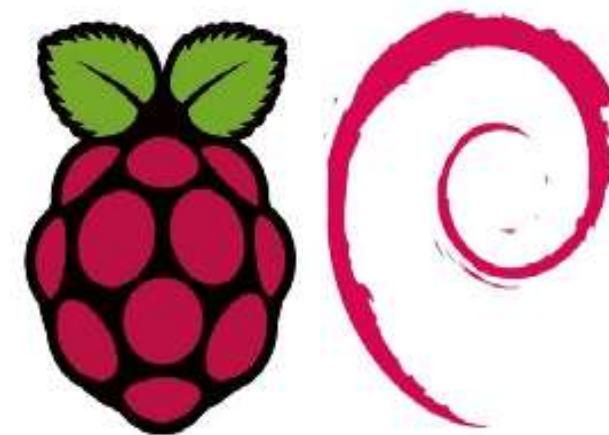
La ingeniería es
una relación de
compromiso

Lenguajes de programación arduino

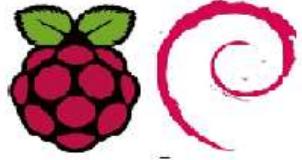




+



Raspbian



Raspbian

- **Raspbian** es una distribución del sistema operativo GNU/Linux basado en Debian, y por lo tanto libre para Raspberry Pi.
- El lanzamiento inicial fue en junio de 2012. Desde 2015, la Raspberry Pi Foundation lo ha proporcionado de forma oficial como el sistema operativo primario para la familia de placas Raspberry Pi. Hay varias versiones de Raspbian, siendo la actual Raspbian Buster.
- Raspbian fue creado por Mike Thompson y Peter Green como un proyecto independiente. El lanzamiento inicial fue en junio de 2012. El sistema operativo aún está en desarrollo activo. Raspbian está altamente optimizado para la Raspberry Pi.

Descarga de Raspbian

<https://www.raspberrypi.org/downloads/raspberry-pi-os/>



balenaEtcher

Descargar de
Balenaetcher

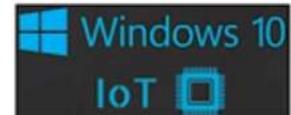
<https://www.balena.io/etcher/>

Raspberry Pi



Raspbian

The Raspberry Pi is a capable little computer running its operating system on a SD card and is powered by a USB phone charger.



Windows IoT



Ubuntu MATE



LibreELEC



RISC OS



FreeBSD



RetroPie

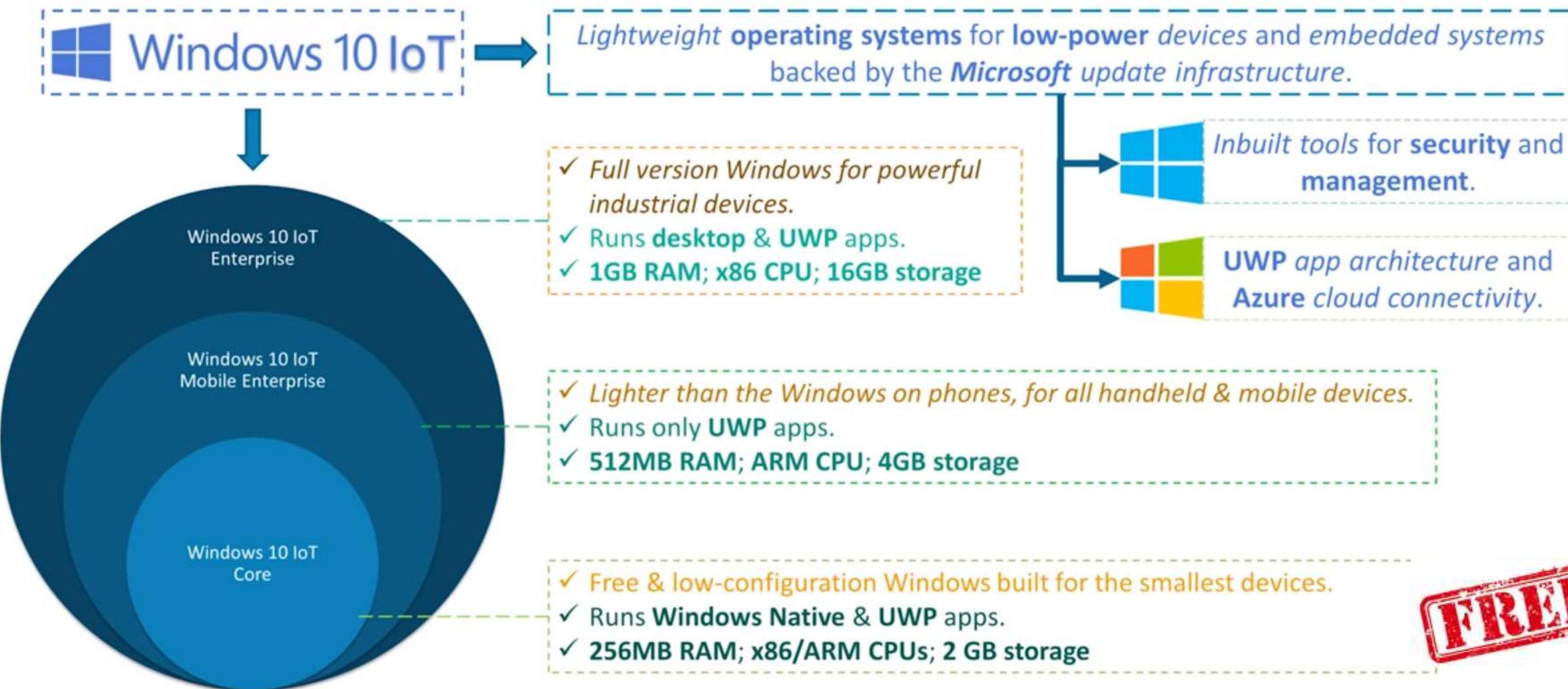


Plan 9

Device Hardware

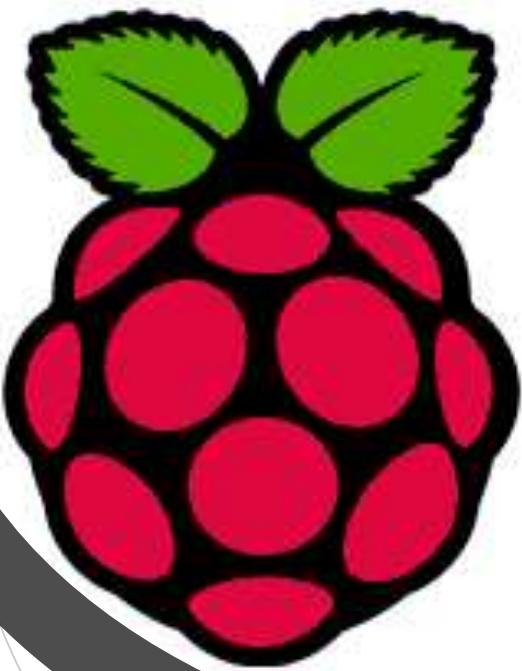
Device Software

Windows 10 IoT



Download Windows IoT Core

- <https://docs.microsoft.com/en-us/windows/iot-core/downloads>



Raspberry en
Virtual Box

Comandos:

- sudo apt-get update
 - sudo apt-get upgrade
 - sudo apt-get install build-essential
module-assistant
 - sudo m-a prepare
- Instalar cd de guest addtiones
- sudo mount /media/cdrom
 - sudo sh
/media/cdrom/VBoxLinuxAdditions.run



Raspberry en C como un Arduino

<https://azure-samples.github.io/raspberry-pi-web-simulator/>

```
In [ ]: x = 1  
        type(x)
```

```
In [ ]: print(x)  
        x="holo"  
        print(x)
```

```
In [ ]: x = [1, 2, 3]  
        y = x  
        print(y,x)
```

```
In [ ]: x.append(4)  
        print(y)
```

```
In [ ]: x=5  
        print(x,y)
```

PRÁCTICA CON FUNCIONES

Formato: Entregar un archivo con formato .ipynb. Debe tener el nombre “Funciones+Apellido”.

Sugerencia: Preparar el código y probar los resultados con distintas entradas.

Desafío
entregable



>> Consigna:

1. Escribir una función para calcular el factorial de un número cualquiera.
2. Escribir una función para calcular la suma de una serie comenzando por un número cualquiera y terminando en otro número que debe ser mayor al primero.

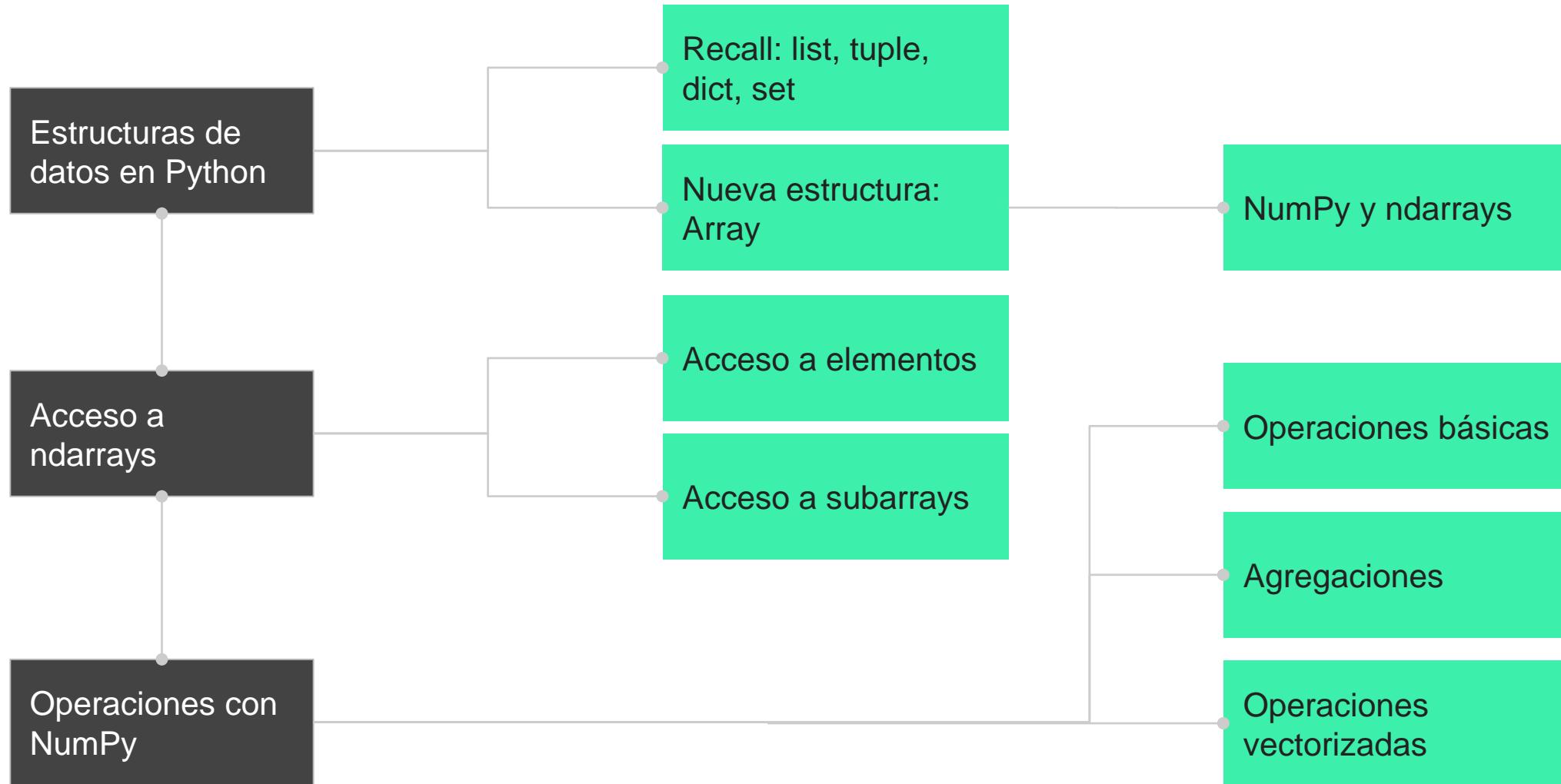
>>Aspectos a incluir en el entregable:

El código debe estar hecho en un notebook y debe estar probado.
Incluir una prueba que demuestre que el código funciona.

>>Ejemplo a continuación:

MAPA DE CONCEPTOS CLASE 3

¡Para recordar!



*Recall: estructuras de
datos*

Recall: estructuras de

- Anteriormente vimos las estructuras **list, tuple, dict y set**.

Tipo	Ejemplo	Definición
list	[1, 2, 3]	Lista ordenada
tuple	(1, 2, 3)	Lista ordenada inmutable
dict	{'a':1, 'b':2, 'c':3}	Diccionario: conjunto de pares clave:valor
set	{1, 2, 3}	Conjunto, a la manera de un conjunto matemático



Recall: estructura *list*

- Anteriormente trabajamos con estructuras **list**, que nos permitían almacenar datos ordenados de distinto tipo.
- Siempre mantenían el **orden** de sus elementos
- Eran **mutables**

```
L = list(range(10))  
L
```



```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

8



Recall: estructura tuple

- Trabajamos también con las estructuras **tuple**.
- Al igual que las listas, siempre mantenían el **orden** de sus elementos
- Eran **inmutables**. Una vez inicializadas, no era posible reasignar elementos.

```
T = tuple(range(10))  
T
```



```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Sin embargo... □

Estas estructuras **no llegan a cubrir**
las necesidades del Data Scientist!

NumPy y ndarrays

El array como estructura de

- Extenderemos la aplicación de los tipos de estructura de datos, agregando el tipo de dato **array**.
- Tanto array como list sirven para guardar conjuntos de datos **ordenados** en memoria.
- Mientras que el tipo de dato list puede guardar datos de diferentes tipos, el tipo de dato array guarda datos de un **único tipo**.
- Esto le permite ser **más eficiente**, especialmente al trabajar con conjuntos de datos grandes.

Numpy arrays

... como los lists, pero con vitaminas 🚀

- La librería Numpy provee una forma particular de array llamada **ndarray** o **Numpy Array**.
- Recordar: los **ndarrays**, al ser un tipo de array, sólo pueden almacenar datos de un mismo tipo.

```
import numpy as np
```

```
Npa = np.array(range(10))
```

```
Npa
```



```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Construyendo ndarrays

Veamos algunos ejemplos

```
Np_cero = np.zeros(10)  
Np_cero
```

```
↳ array([0., 0., 0., ..., 0.])
```

```
Np_cero_int = np.zeros(10, dtype=int)  
Np_cero_int
```

```
↳ array([0, 0, 0, ..., 0])
```

```
Np_uno = np.ones(10)  
Np_uno
```

```
↳ array([1., 1., 1., ..., 1.])
```

```
Np_relleno = np.full(10,256)  
Np_relleno
```

```
↳ array([256, 256, 256, ..., 256])
```

Construyendo ndarrays

- Numpy provee objetos **rango**:

```
Np_rango = np.arange(10)  
Np_rango
```



```
array([0, 1, 2, ..., 9])
```

- Ndarrays con **valores aleatorios** y de **dos dimensiones**:

```
Np_random_dimensiones = np.random.randint(10, size=(3, 4))  
Np_random_dimensiones
```

```
array([[6, 5, 4, 7],  
       [0, 1, 1, 2],  
       [0, 0, 3, 8]])
```

Propiedades de los

Inspectaremos **arrays** nuestros Numpy arrays 🔎

Podemos acceder a distintas **propiedades** de los arreglos:

- Dimensión:

```
Np_cero.ndim
```



1

- Forma:

```
Np_relleno_dimensiones.shape
```



(2, 3)

- Tamaño:

```
Np_relleno_dimensiones.size
```



6

Propiedades de los

Inspeccionemos un poco nuestros Numpy arrays ↗
Arrays

Podemos acceder a distintas **propiedades** de los arreglos:

- *Tipo de dato:*

N_cero.dtype	→ float64
Np_cero_int.dtype	→ int64
N_relleno_dimensiones.itemsize	→ 8

- *Tamaño de elemento:*

Np_cero.nbytes	→ 80
Np_cero_int.nbytes	

- *Tamaño total:*

Indexado y acceso

Accediendo elementos

Veamos cómo consultar los arreglos

- Al igual que las listas, los elementos del arreglo se acceden mediante su índice, comenzando desde 0.

```
rango = range(1,11)  
Np_diez_numeros = np.array(rango)  
Np_diez_numeros
```

↳ array([1, 2, 3, ..., 10])

- Primer elemento:
- Quinto elemento:

```
Np_diez_numeros[0]  
Np_diez_numeros[4]
```

↳ 1
↳ 5

Accediendo elementos

- Podemos seleccionar elementos desde atrás para adelante mediante índices negativos, **comenzando desde -1**.
- Último elemento:
- Penúltimo elemento:
- Para acceder a un elemento de una matriz, indicar fila y columna:

```
Np diez numeros[-1]
```

☞ 10

```
Np_diez_numeros[-2]
```

☞ 9

```
Np_random_dimensiones
```



```
array([[6, 5, 4, 7],  
       [0, 1, 1, 2],  
       [0, 0, 3, 8]])
```

```
Np_random_dimensiones[2, 1]
```



☞ 0

Accediendo subarrays

- Podemos seleccionar una rebanadas del arreglo de la siguiente manera:

Objeto[desde:hasta:tamaño_de_paso]

- El parámetro tamaño_de_paso permite, por ejemplo, seleccionar elementos de dos en dos
- Atención a estos detalles:
 - El índice "desde" **es inclusivo.** 
 - El índice "hasta" **es exclusivo.** 

Accediendo subarrays

Veamos algunos ejemplos...

- Primeros cuatro:
- Desde el cuarto:
- Desde el quinto al séptimo:
- De dos en dos:
- Desde atrás, de dos en dos:

```
Np_diez_numeros[:4]
```

```
[1 2 3 4]
```

```
Np_diez_numeros[3:]
```

```
[ 4 5 ... 10]
```

```
Np_diez_numeros[4:7]
```

```
[5 6 7]
```

```
Np_diez_numeros[::-2]
```

```
[1 3 5 7 9]
```

```
Np_diez_numeros[::-2]
```

```
[10 8 6 4 2]
```

Accediendo subarrays

- Para **arreglos multidimensionales**, especificar los índices de manera ordenada:

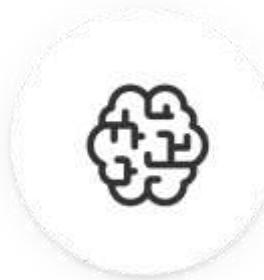
Objeto[dimensión1, dimensión2,...]

Veamos algunos ejemplos...

- Tercera fila, todas las columnas: Np_random_dimensiones[2,:]
- Primeras dos filas, primeras dos columnas Np_random_dimensiones[:2, :2]
- Tercera fila, cuarta columna: Np_random_dimensiones[2, 3]

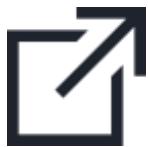
OPERACIONES BÁSICAS

*Reshape,
concatenación,
splitting*



PENSANDO EN AJEDREZ





Reshape

- Permite modificar la dimensión de un arreglo, retornando otro con distinta dimensión y forma pero manteniendo los mismos elementos.

```
np.arange(1,65)
```

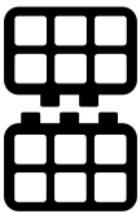
```
array([ 1,  2,  3, ..., 64])
```

```
Ajedrez_64 = np.arange(1,65).reshape(8,8)
```

```
Ajedrez_64
```

Sencillo, ¿verdad? 😊

```
array([[ 1,  2,  3,  4,  5,  6,  7,  8],
       [ 9, 10, 11, 12, 13, 14, 15, 16],
       [17, 18, 19, 20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29, 30, 31, 32],
       [33, 34, 35, 36, 37, 38, 39, 40],
       [41, 42, 43, 44, 45, 46, 47, 48],
       [49, 50, 51, 52, 53, 54, 55, 56],
       [57, 58, 59, 60, 61, 62, 63, 64]])
```

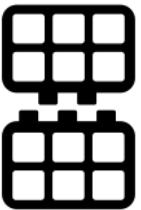


Concatenación

- Consiste en formar un nuevo arreglo a partir de “enganchar” o “apilar” otros.
- Python ofrece dos métodos:
 - Con la operación **concatenate**.
 - Con las operaciones **vstack** y **hstack**

```
Array_1 = np.random.randint(10, size=5)
Array_2 = np.random.randint(10, size=5)
Arrays_concatenados = np.concatenate([Array_1, Array_2])
```

```
array([0, 6, 5, 4, 3, 1, 8, 0, 0, 3])
```



Concatenación

- El método **vstack** apila verticalmente:

```
Array_extra = np.array([[10],[20]])
```

```
Array_extra
```

```
array([[10],  
       [20]])
```

```
Array_apilados_v = np.vstack([Array_extra, Array_extra])
```

```
Array_apilados_v
```

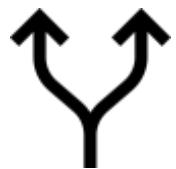
```
array([[10],  
       [20],  
       [10],  
       [20]])
```

- El método **hstack** apila horizontalmente:

```
Array_apilados_h = np.hstack([Array_extra, Array_extra])
```

```
Array_apilados_h
```

```
array([[10, 10],  
       [20, 20]])
```



Splitting

- Consiste en **desarmar** o **partir** los arreglos.
- Puede pensarse como la **operación inversa a la concatenación**

```
Arrays_concatenados
```

```
array([0, 6, 5, 4, 3, 1, 8, 0, 0, 3])
```

```
Array_partido = np.split(Arrays_concatenados, [2])
Array_partido
```



```
[array([0, 6]), array([5, 4, 3, 1, 8, 0, 0, 3])]
```

Especificamos los **puntos de corte** con un arreglo. En este caso queremos un *único corte entre el segundo y tercer elemento*



Splitting

- *Dos puntos de corte*

```
Array_partido_2 = np.split(ARRAYS_concatenados, [2, 8])
```

```
Array_partido_2
```

↳

```
[array([0, 6]), array([5, 4, 3, 1, 8, 0]), array([0, 3])]
```

- Podemos *desarmar* el arreglo y guardarlo en variables distintas



```
Parte_1, Parte_2, Parte_3 = Array_partido_2
```

Parte_1

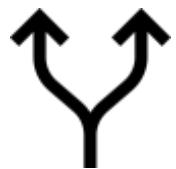
```
array([0, 6])
```

Parte_2

```
array([5, 4, 3, 1, 8, 0])
```

Parte_3

```
array([0, 3])
```



Splitting

hsplit realiza cortes verticales:

```
Ajedrez_partido_1 = np.hsplit(Ajedrez_64, [4])  
Ajedrez_partido_1
```



```
[array([[ 1,  2,  3,  4],  
       [ 9, 10, 11, 12],  
       [17, 18, 19, 20],  
       [25, 26, 27, 28],  
       [33, 34, 35, 36],  
       [41, 42, 43, 44],  
       [49, 50, 51, 52],  
       [57, 58, 59, 60]]),  
 array([[ 5,  6,  7,  8],  
       [13, 14, 15, 16],  
       [21, 22, 23, 24],  
       [29, 30, 31, 32],  
       [37, 38, 39, 40],  
       [45, 46, 47, 48],  
       [53, 54, 55, 56],  
       [61, 62, 63, 64]]])
```

vsplit realiza cortes horizontales:

```
Ajedrez_partido_2 = np.vsplit(Ajedrez_64, [4])  
Ajedrez_partido_2
```



```
[array([[ 1,  2,  3,  4,  5,  6,  7,  8],  
       [ 9, 10, 11, 12, 13, 14, 15, 16],  
       [17, 18, 19, 20, 21, 22, 23, 24],  
       [25, 26, 27, 28, 29, 30, 31, 32]]),  
 array([[33, 34, 35, 36, 37, 38, 39, 40],  
       [41, 42, 43, 44, 45, 46, 47, 48],  
       [49, 50, 51, 52, 53, 54, 55, 56],  
       [57, 58, 59, 60, 61, 62, 63, 64]]])
```

Agregaciones y Operaciones vectorizadas



Cálculos sobre Numpy

Como futuros Data Scientists, **definitivamente** nos encontraremos con la tarea de efectuar **cálculos a partir de arrays**

Numpy está para darnos una mano en esto



Calculando el promedio

Una solución *tradicional* al problema es la siguiente:

```
Array_aleatorio = np.random.randint(10, size=10)  
print(Array_aleatorio)  
summa = 0  
  
for i in Array_aleatorio:  
    summa += i  
  
promedio = summa / np.size(Array_aleatorio)
```

Si bien esta resolución es elegante y cumple con su tarea, Numpy nos provee de opciones más **cómodas y eficientes** 🚀



Agregaciones

Demos un vistazo a las agregaciones principales ➔

- Suma:
- Promedio:
- Valor máximo:
- Mediana:
- Desvío estándar:
- Varianza:

Array_aleatorio.sum()

Array_aleatorio.mean()

Array_aleatorio.max()

np.median(Array_aleatorio)

np.std(Array_aleatorio)

np.var(Array_aleatorio)

*Estas funciones están optimizadas para grandes volúmenes de datos y además nos ahoran **mucho código...***





Operaciones vectorizadas

¿Por qué son tan importantes?

- Incluso las operaciones más sencillas pueden resultar muy lentas si las llevamos a cabo elemento a elemento.
- Las computadoras son especialmente buenas para realizar cálculos en **paralelo** □
- Las **operaciones vectorizadas** o **funciones universales (ufuncs)** nos permiten operar entre arreglos de la manera más rápida posible.



Operaciones vectorizadas

Operemos arreglos, pero de manera eficiente 😊

Recordemos los arreglos de prueba:

Array_1

↳ `array([7, 0, 6, 7, 5])`

Array_2

↳ `array([3, 7, 9, 9, 0])`

- Sumas vectorizadas:

Array_1 + 5

↳ `array([12, 5, 11, 12, 10])`

Array_1 + Array_2

`np.add(Array_1, Array_2)`

↳ `array([10, 7, 15, 16, 5])`

Ambas formas son equivalentes!



Producto Vectorial

- El **producto vectorial** sobre arreglos *unidimensionales* se calcula sumando los resultados de multiplicar los elementos que tienen la misma posición.
- En Numpy, la versión vectorizada se implementa en el método np.matmul

```
np.matmul(Array_1, Array_2)
```



138



PRÁCTICA CON ARRAYS EN

NUMPY

Profundizaremos el uso de los arrays en NumPy con los
siguientes ejercicios

Visualización de datos

Una herramienta clave en el
proceso de data science

Diplomatura en Data Science Aplicada

Introducción

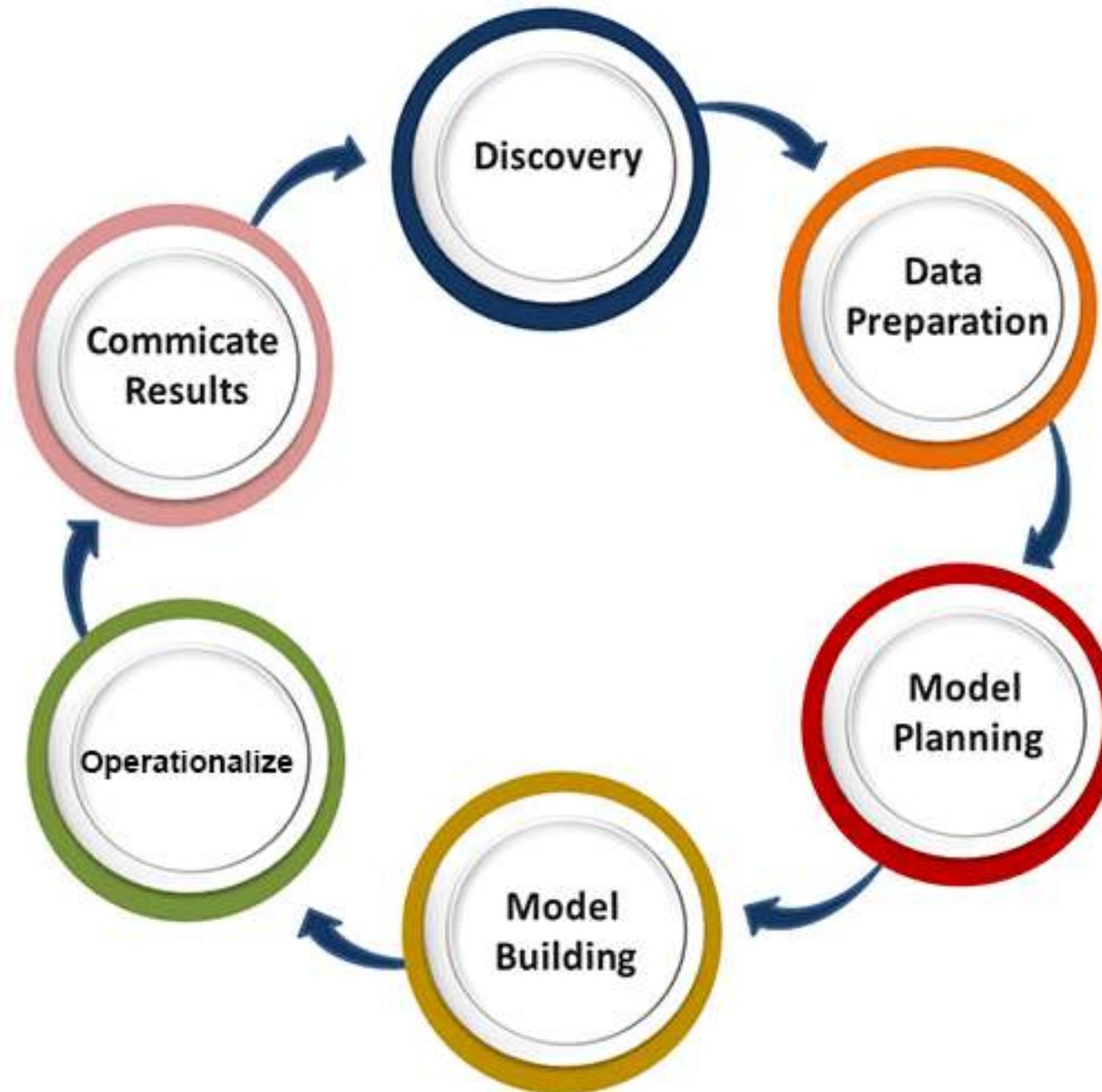
El objetivo de esta clase será brindarles herramientas para la visualización eficiente de datos con python. Ya sea para el propio entendimiento de los mismos o su comunicación a terceros.



La Importancia de la Visualización de Datos

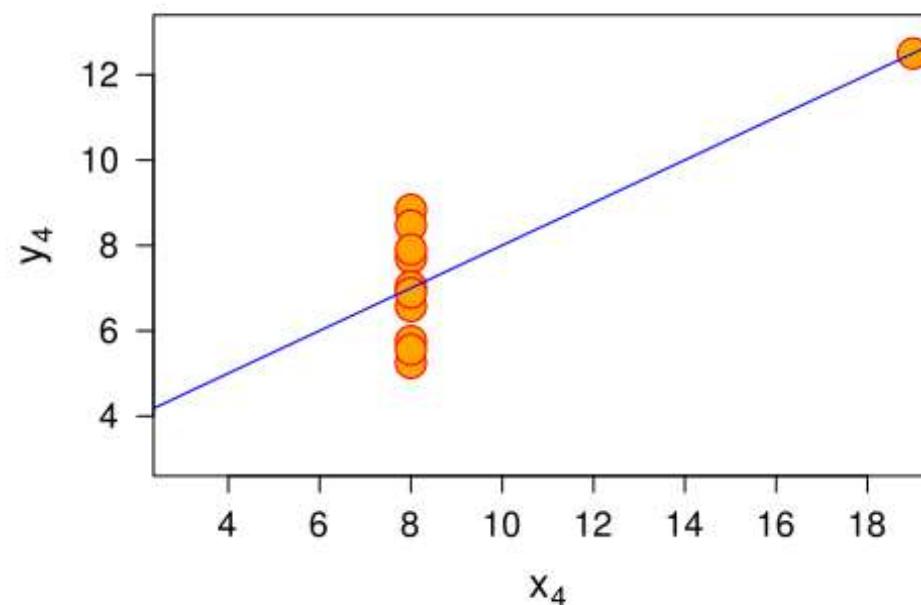
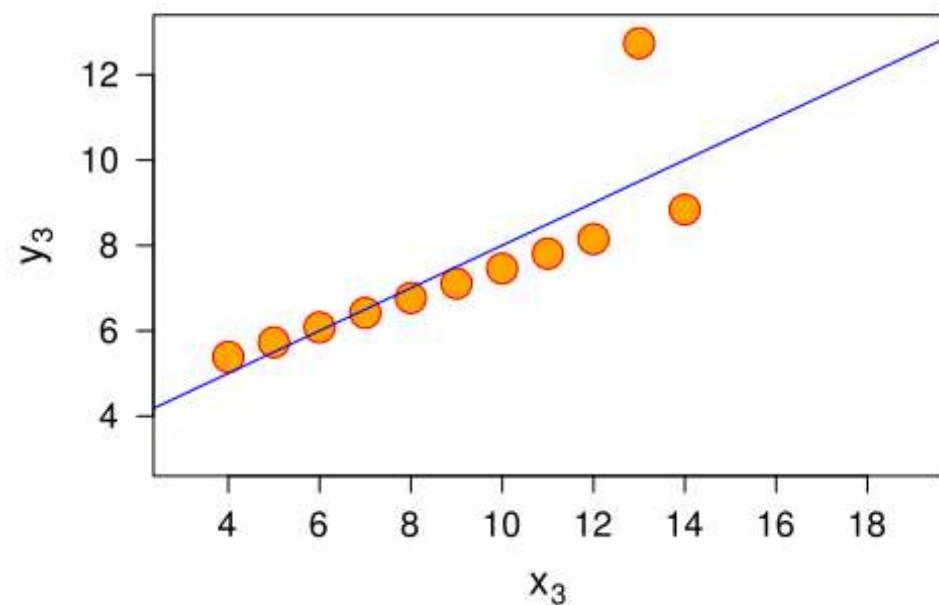
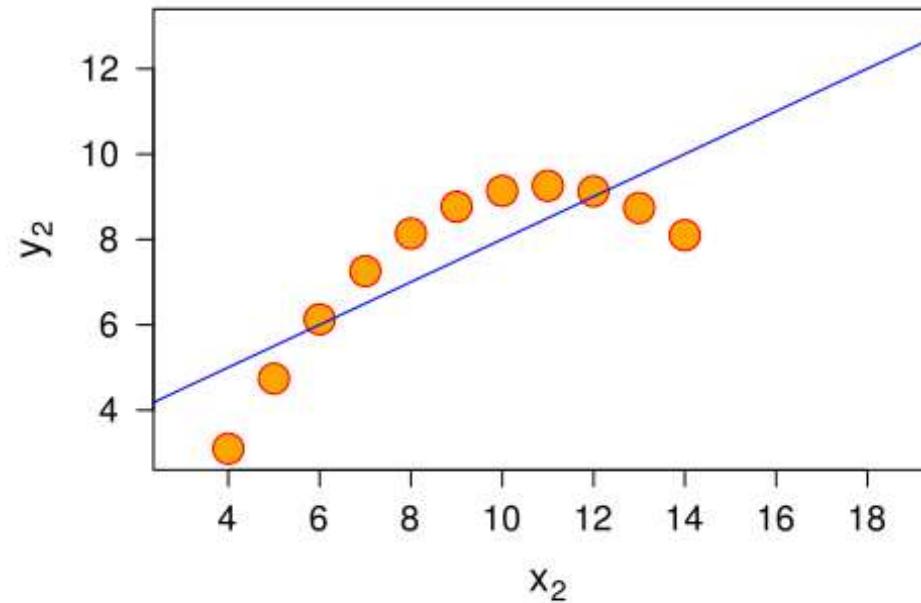
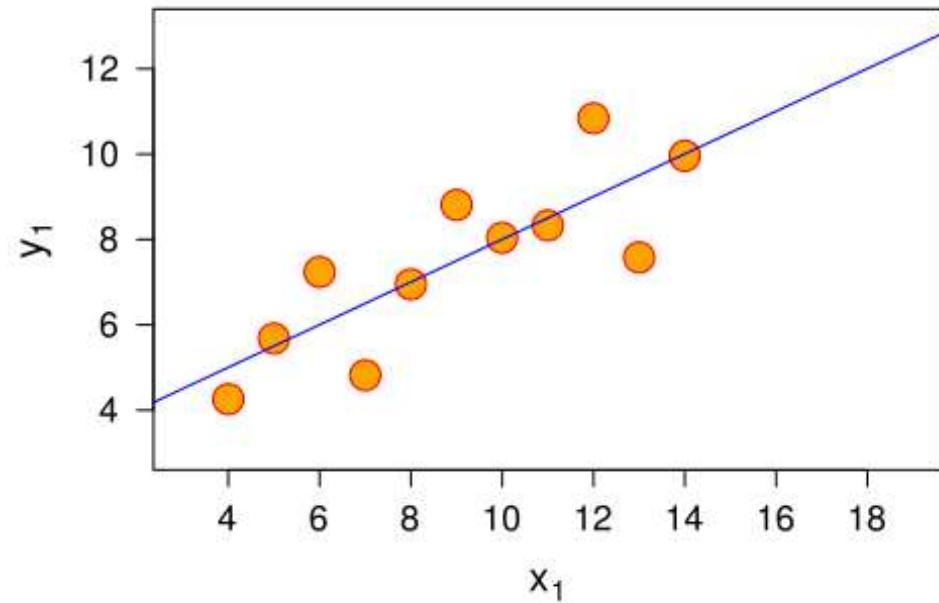
- Utilizar gráficos para visualizar grandes volúmenes de datos, es más fácil que examinar datasets.
- Ellas nos permitirán cubrir el gap entre nuestra capacidad cognitiva y el volumen de datos que manejamos actualmente
- Permite detectar patrones, tendencias y correlaciones que de otro modo pueden pasar desapercibidos.
- Comunicación de resultados.
- Valor para terceros a la hora de tomar decisiones, auditar procesos etc.

Data science life-cycle



Anscombe's quartet

Anscombe's Data									
Observation	x1	y1	x2	y2	x3	y3	x4	y4	
1	10	8,04	10	9,14	10	7,46	8	6,58	
2	8	6,95	8	8,14	8	6,77	8	5,76	
3	13	7,58	13	8,74	13	12,74	8	7,71	
4	9	8,81	9	8,77	9	7,11	8	8,84	
5	11	8,33	11	9,26	11	7,81	8	8,47	
6	14	9,96	14	8,1	14	8,84	8	7,04	
7	6	7,24	6	6,13	6	6,08	8	5,25	
8	4	4,26	4	3,1	4	5,39	19	12,5	
9	12	10,84	12	9,13	12	8,15	8	5,56	
10	7	4,82	7	7,26	7	6,42	8	7,91	
11	5	5,68	5	4,74	5	5,73	8	6,89	
Summary Statistics									
N	11	11	11	11	11	11	11	11	11
MEDIA	9,00	7,50	9,00	7,50091	9,00	7,50	9,00	7,50	
DESVÍO S	3,16	1,94	3,16	1,94	3,16	1,94	3,16	1,94	
r	0,82		0,82		0,82		0,82		



¿Que es EDA?

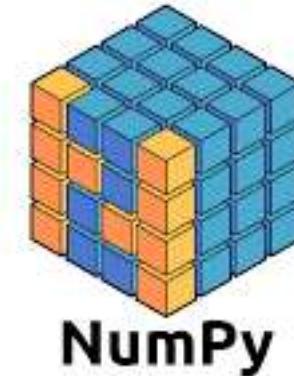
El EDA (Exploratory Data Analysis) es un proceso crítico en la investigación inicial de los datos. El principal objetivo es comprender los datos y descubrir si existe relación entre las variables.

Implica:

- Descubrir patrones
- Limpieza de datos
- Detectar anomalías
- Probar hipótesis
- Verificar supuestos con la ayuda de estadísticas y gráficos

Librerías

- Numpy: Algebra lineal
- Pandas: Análisis de datos
- Matplotlib: Visualización
- Seaborn: Visualización enfocada en datasets
- Plotly: Visualizaciones interactivas
- Dash: Tableros (introductorio)



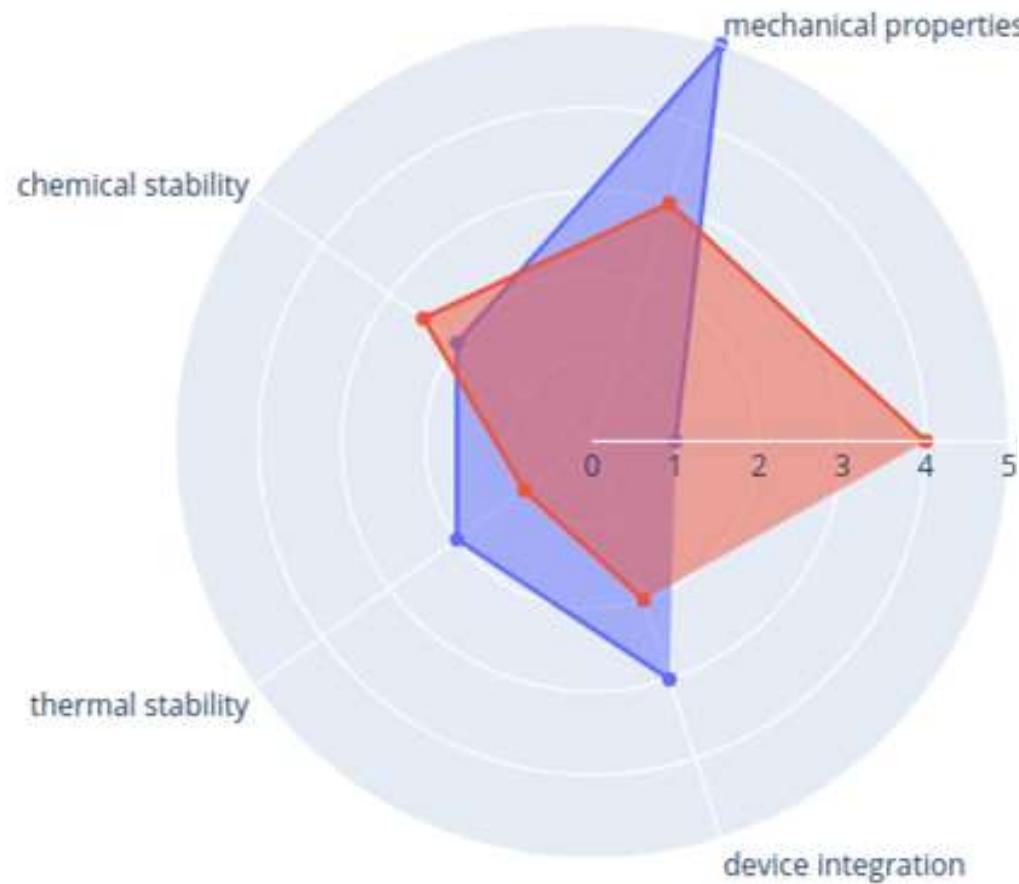
¿Cuál librería utilizar?

¿Qué quiero visualizar?

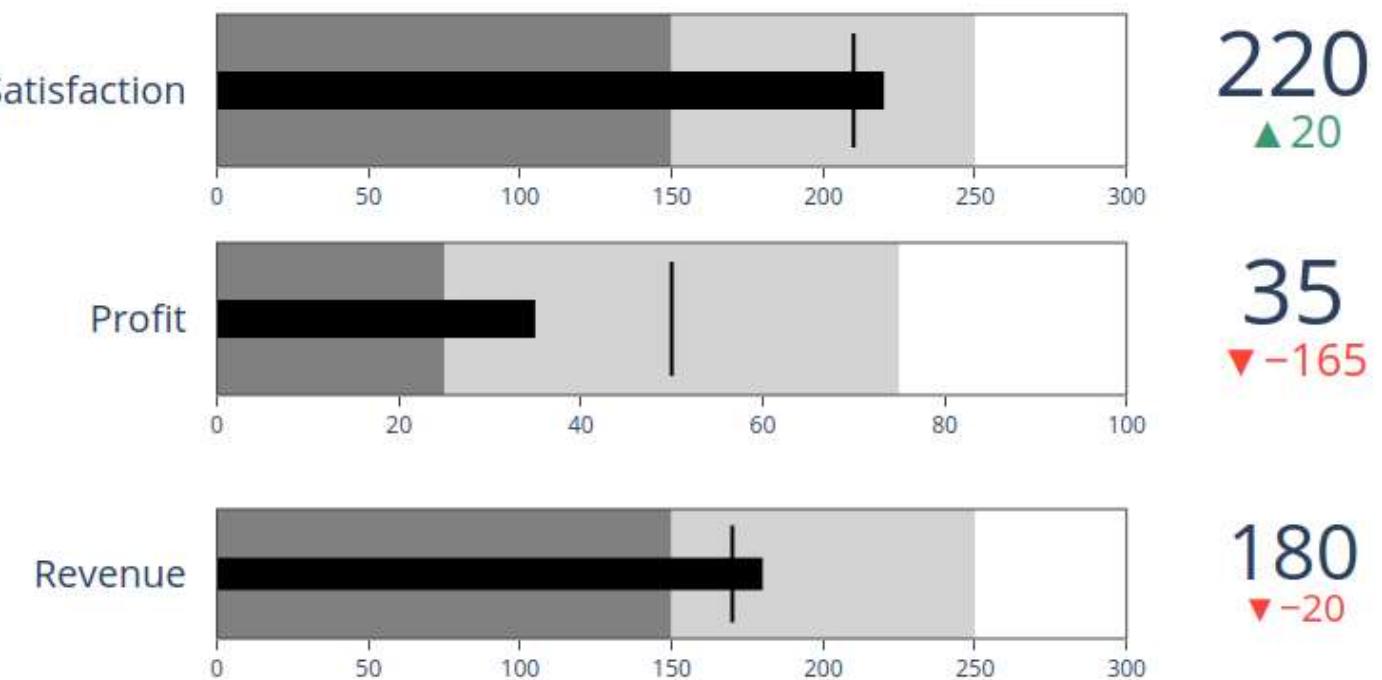
¿Quién va a ver este gráfico?

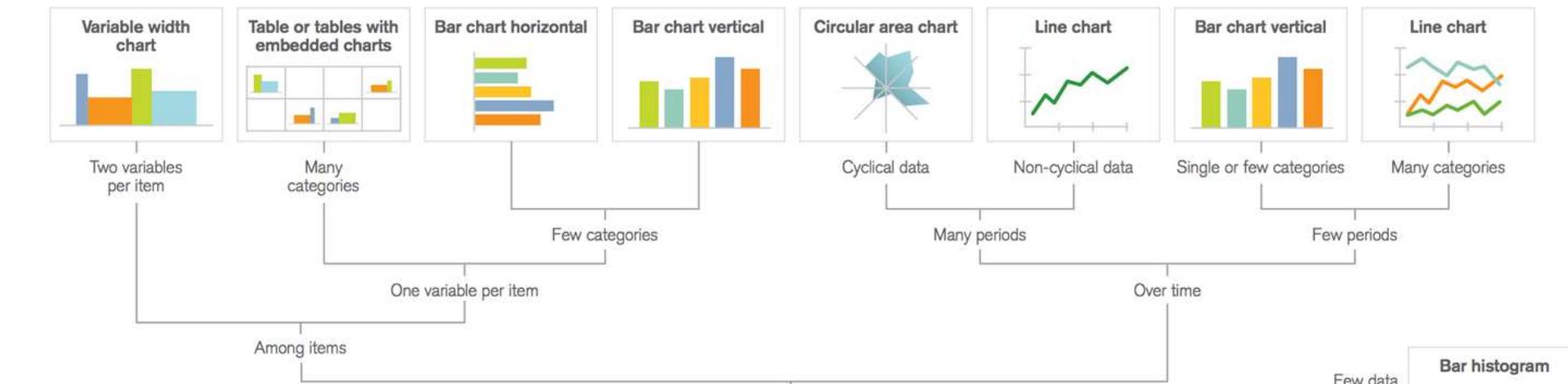


Propuesta



Requerimiento

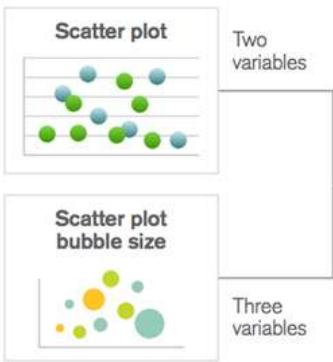




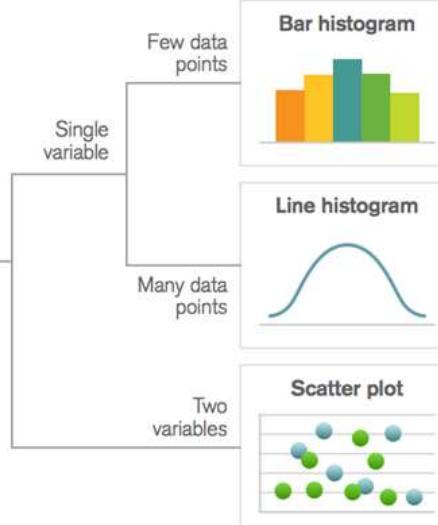
Comparison

What would you
like to show?

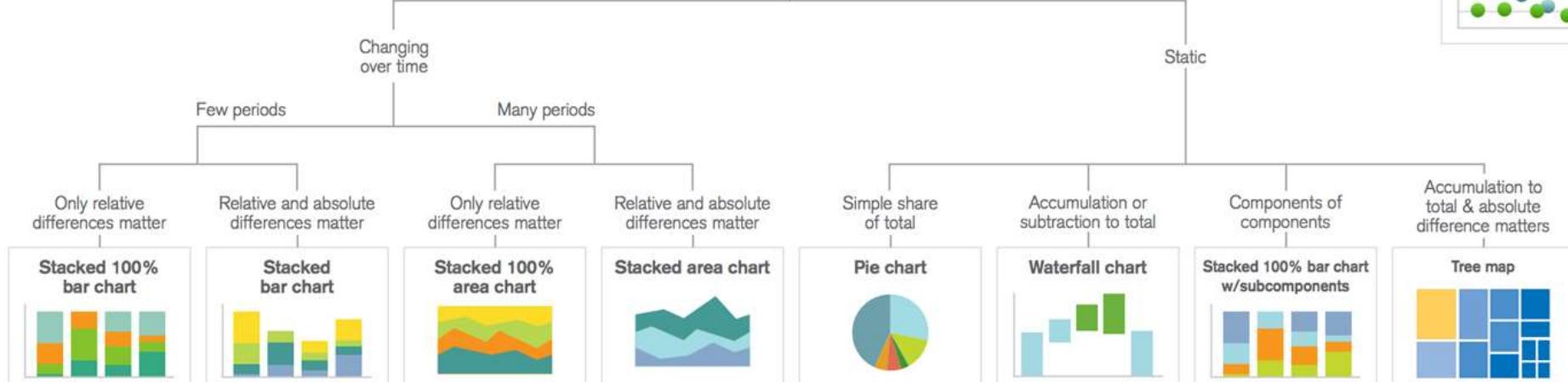
Relationship



Distribution



Composition



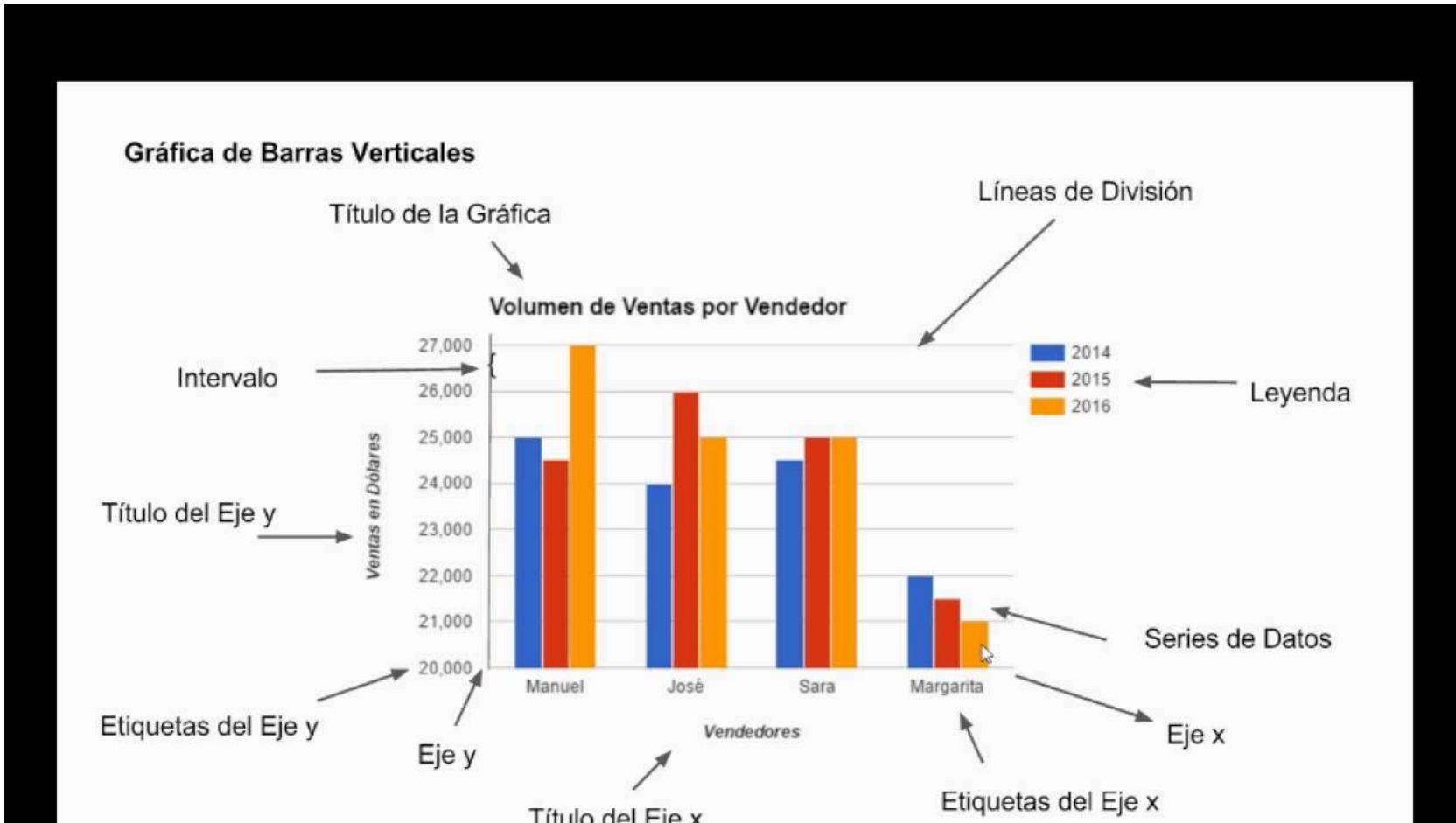
La pirámide de las librerías
para visualizar



plotly



Partes de un gráfico



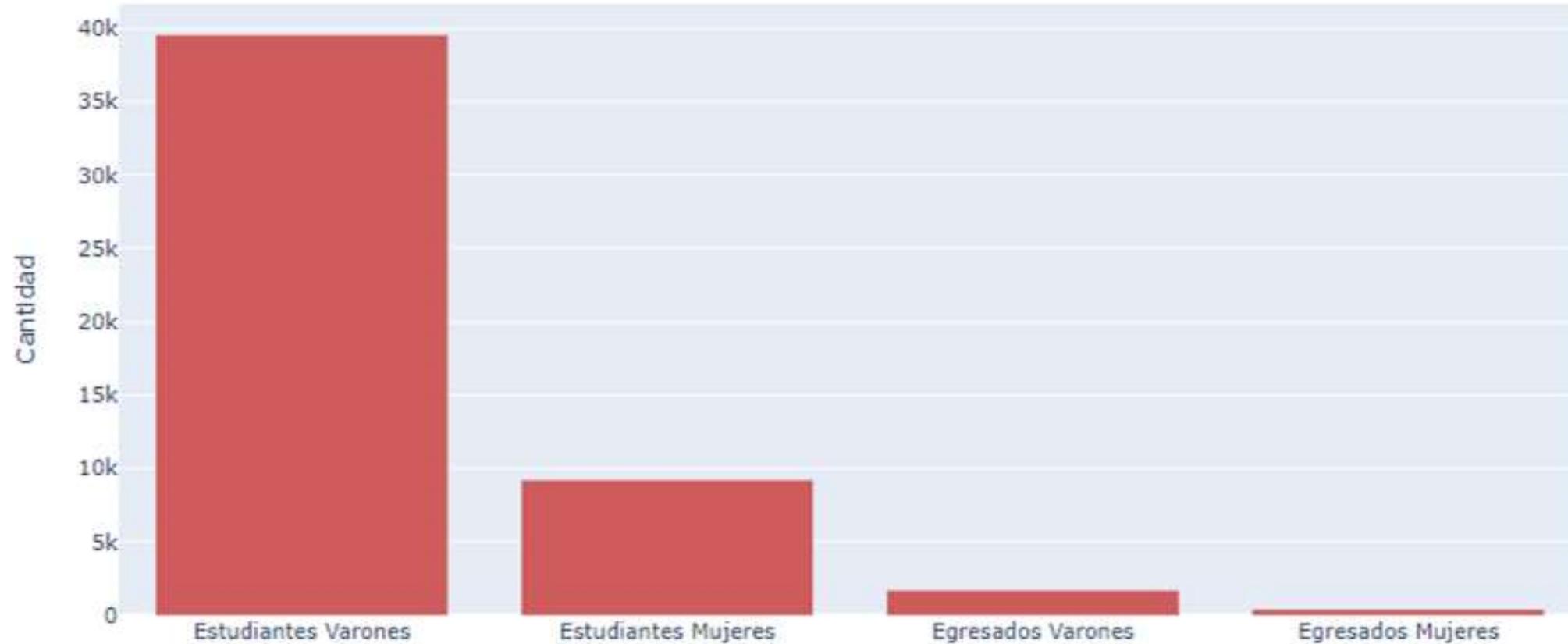
TIPOS DE GRÁFICOS Y SU UTILIDAD

Los datos utilizados para esta presentación se encuentran disponibles en
<http://mujeresprogramadoras.com.ar/>

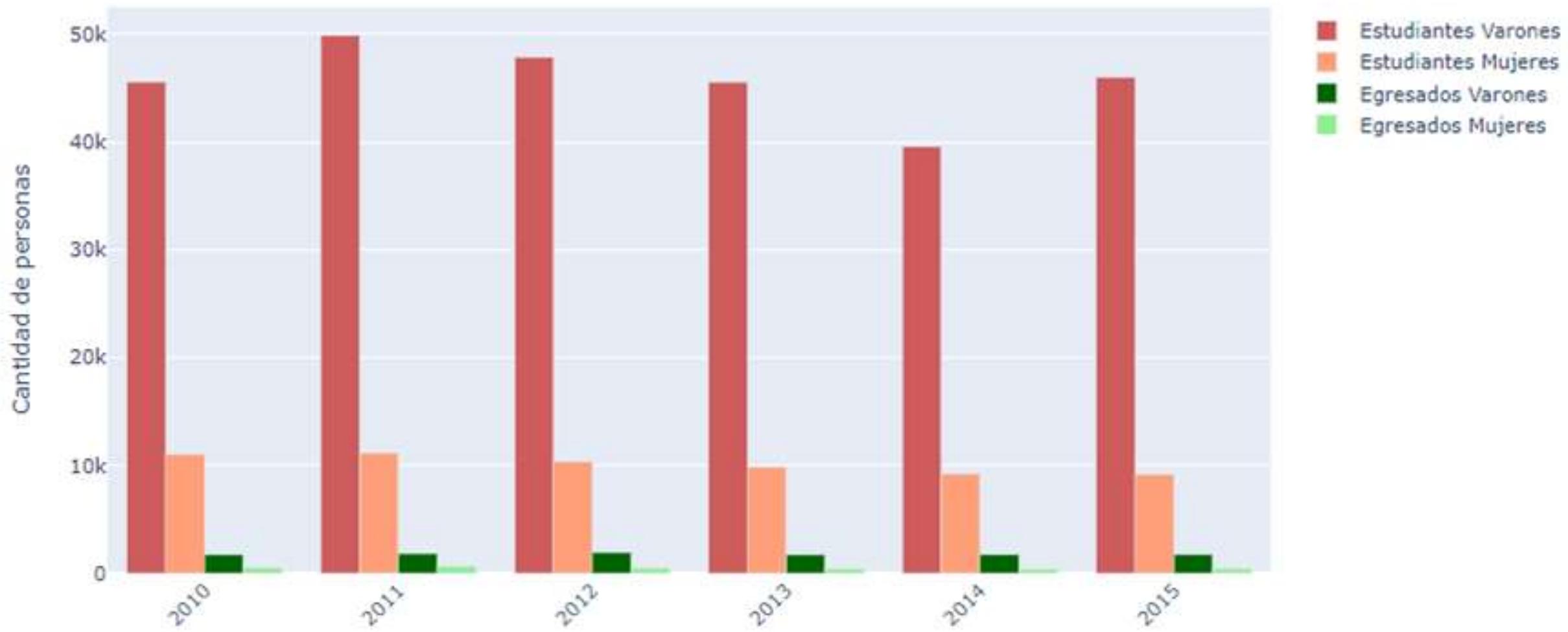
Esta es una base de datos abierta y pública sobre Mujeres en carreras relacionadas a programación de la Argentina.

Bar chart o gráfico de barra

Estudiantes y egresos año 2014

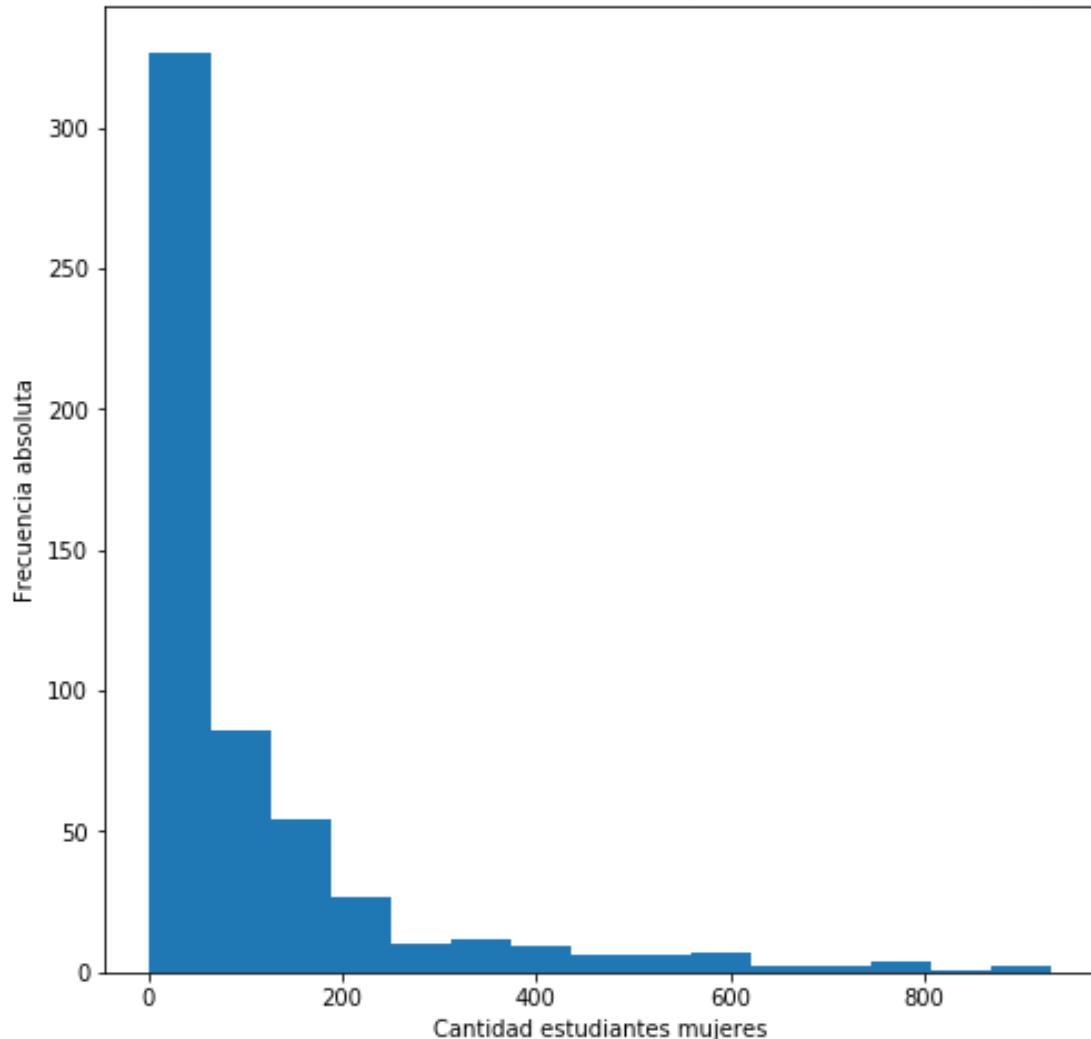


Bar chart o gráfico de barra

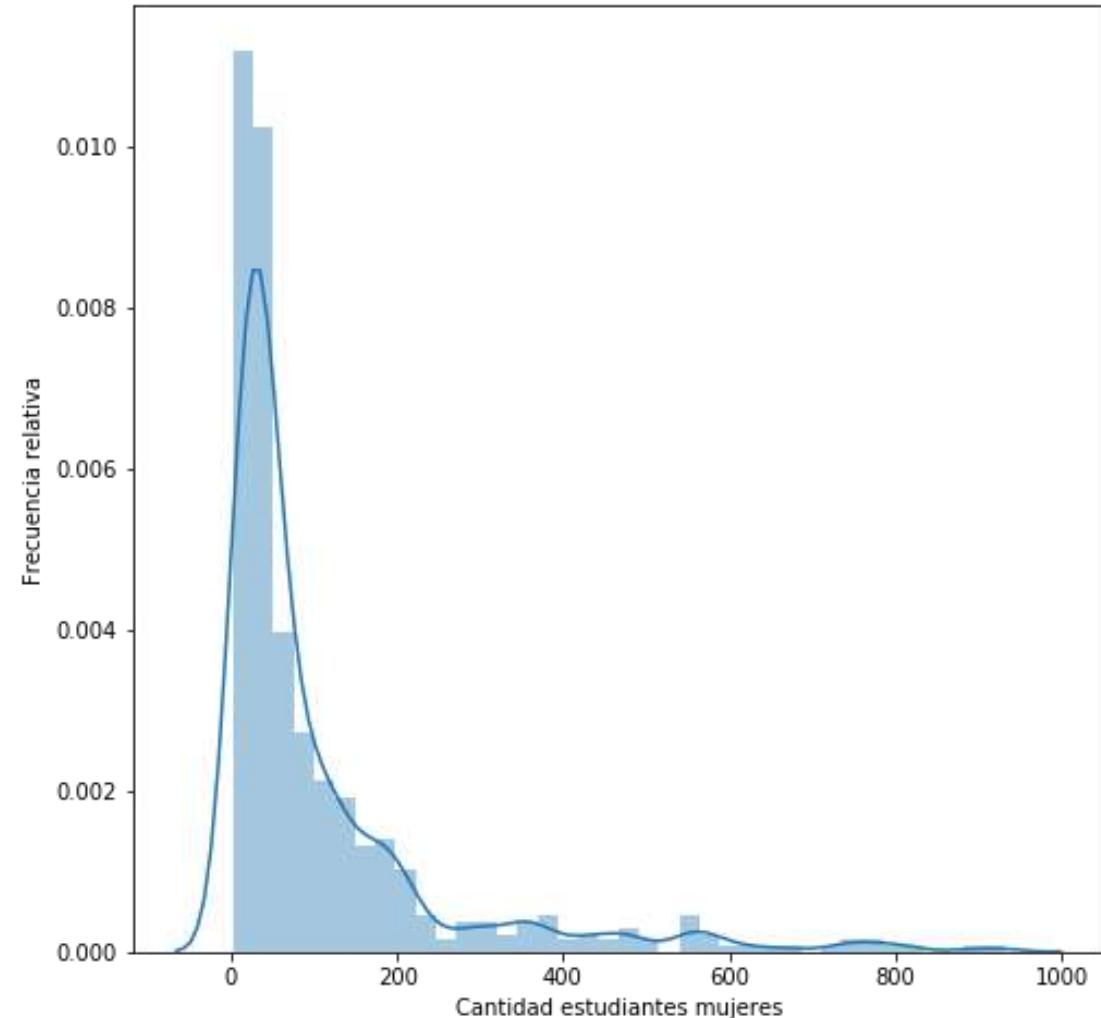


Histogram plot o histograma

Distribución del número de estudiantes mujeres en carreras tecnologicas

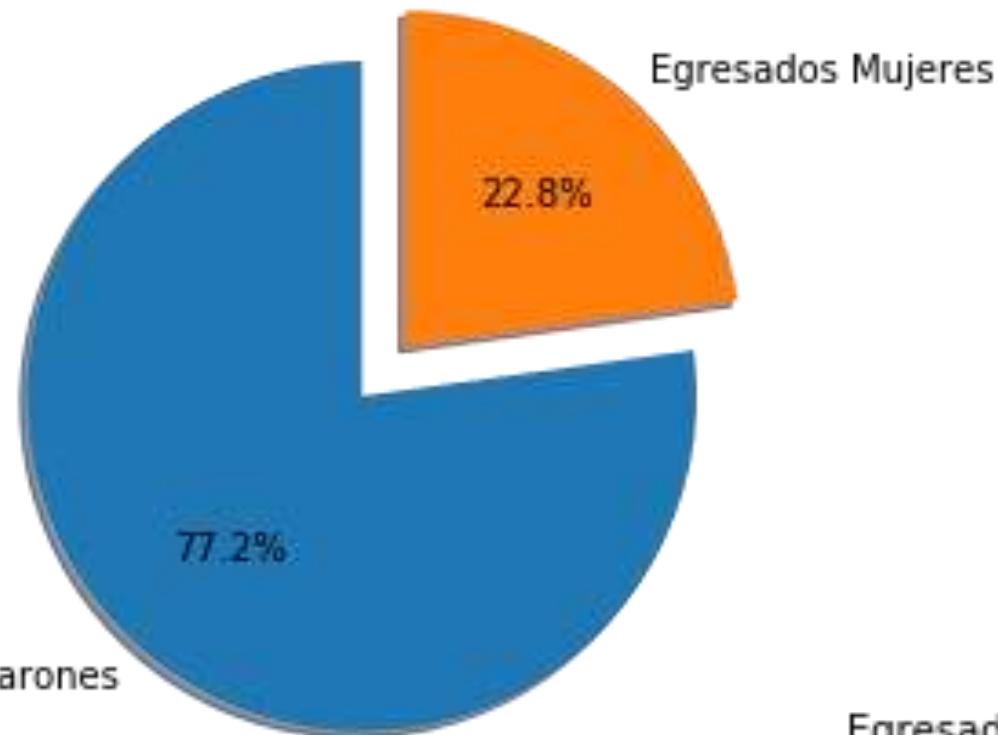


Distribución del número de estudiantes mujeres en carreras tecnologicas

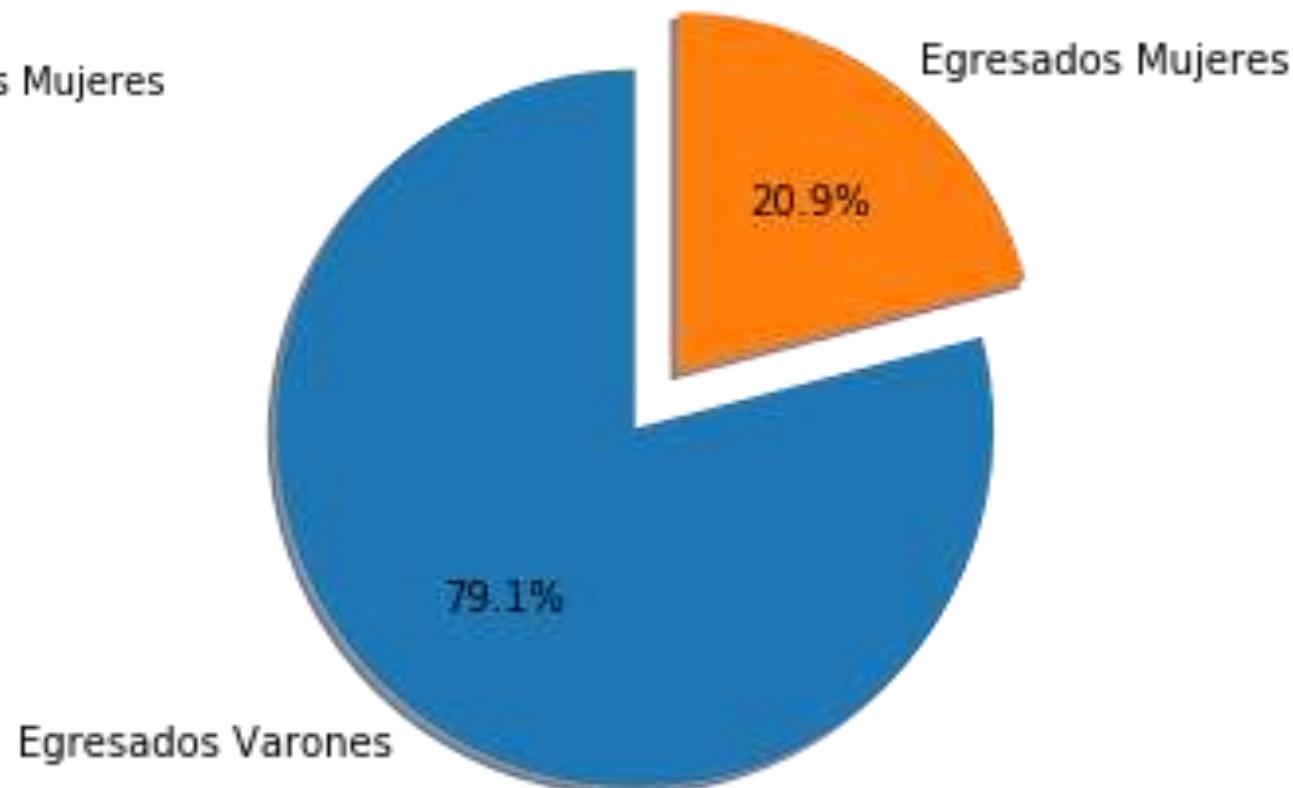


Pie chart o gráfico de torta

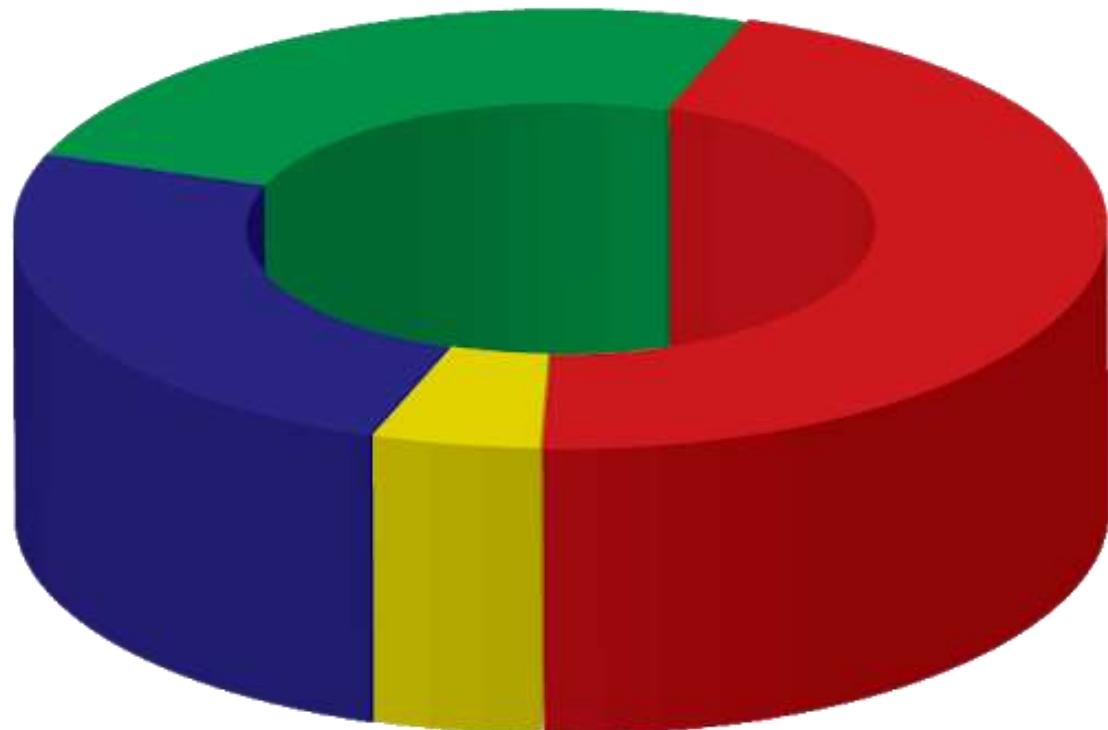
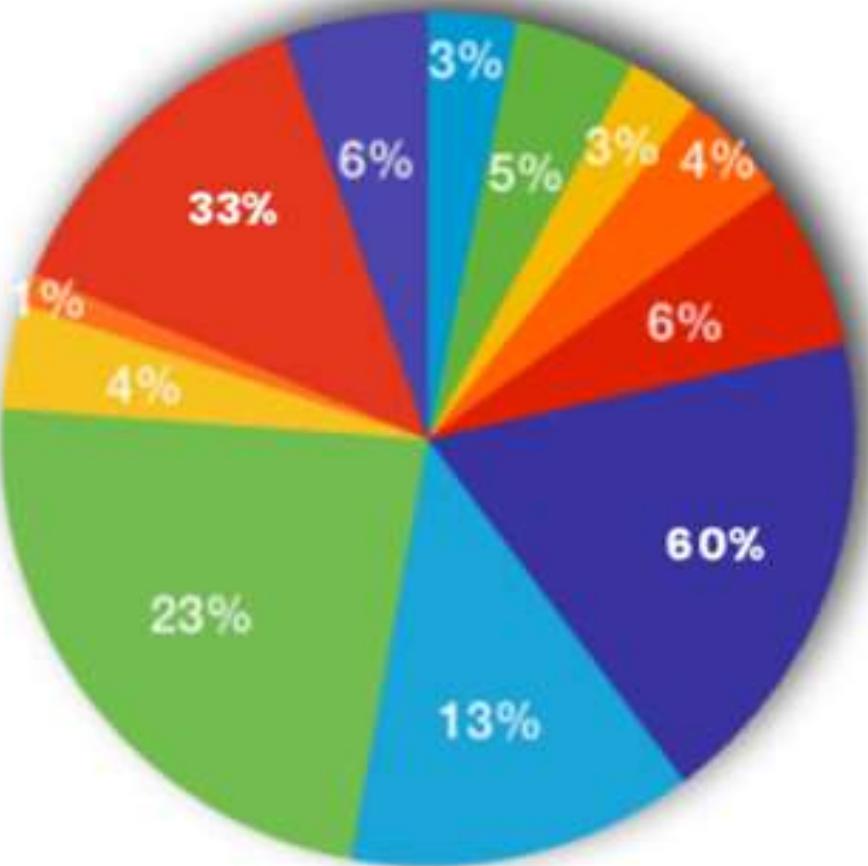
Egresos 2010



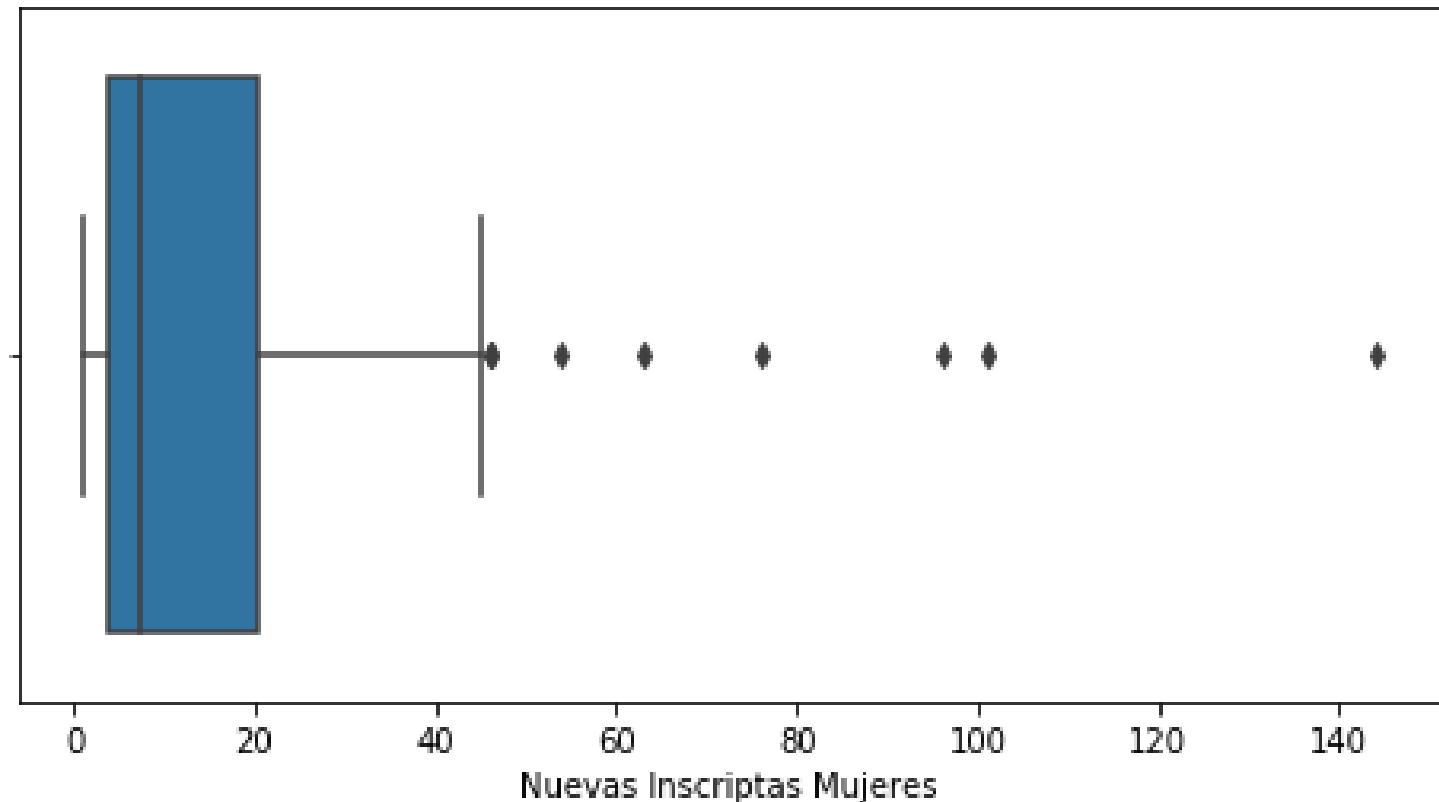
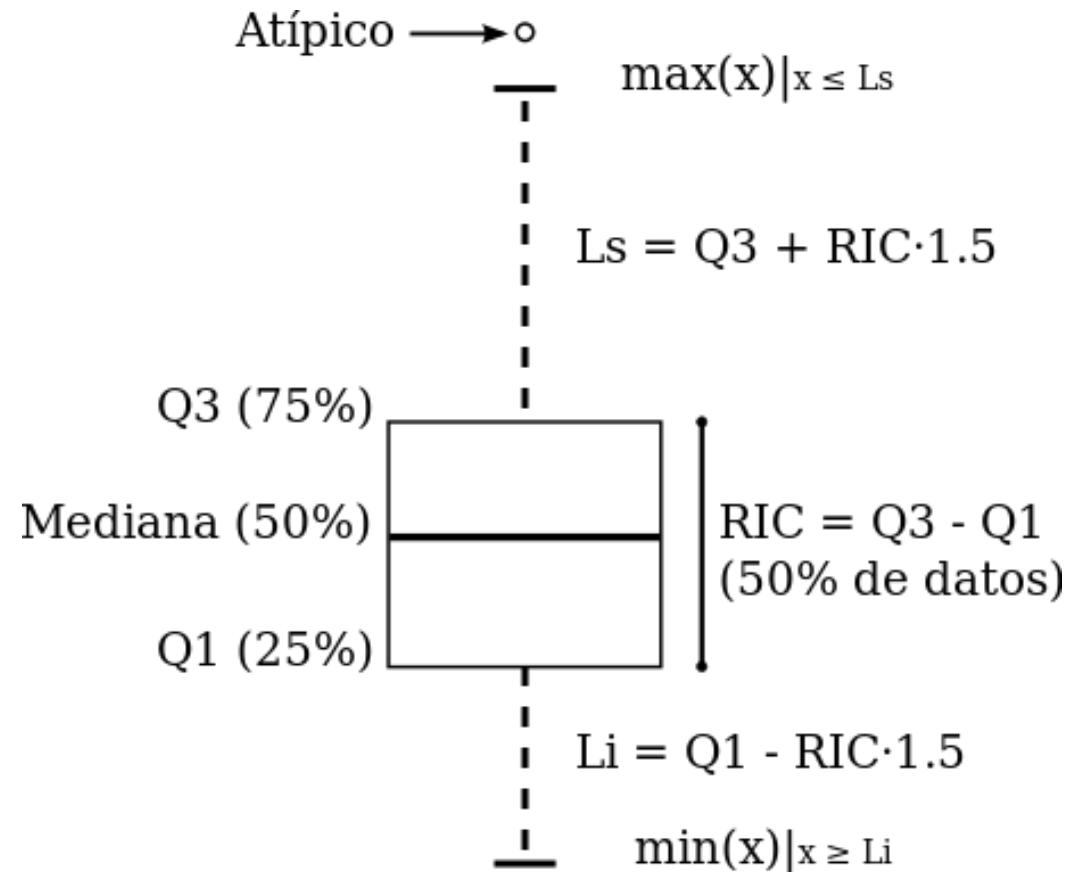
Egresos 2015



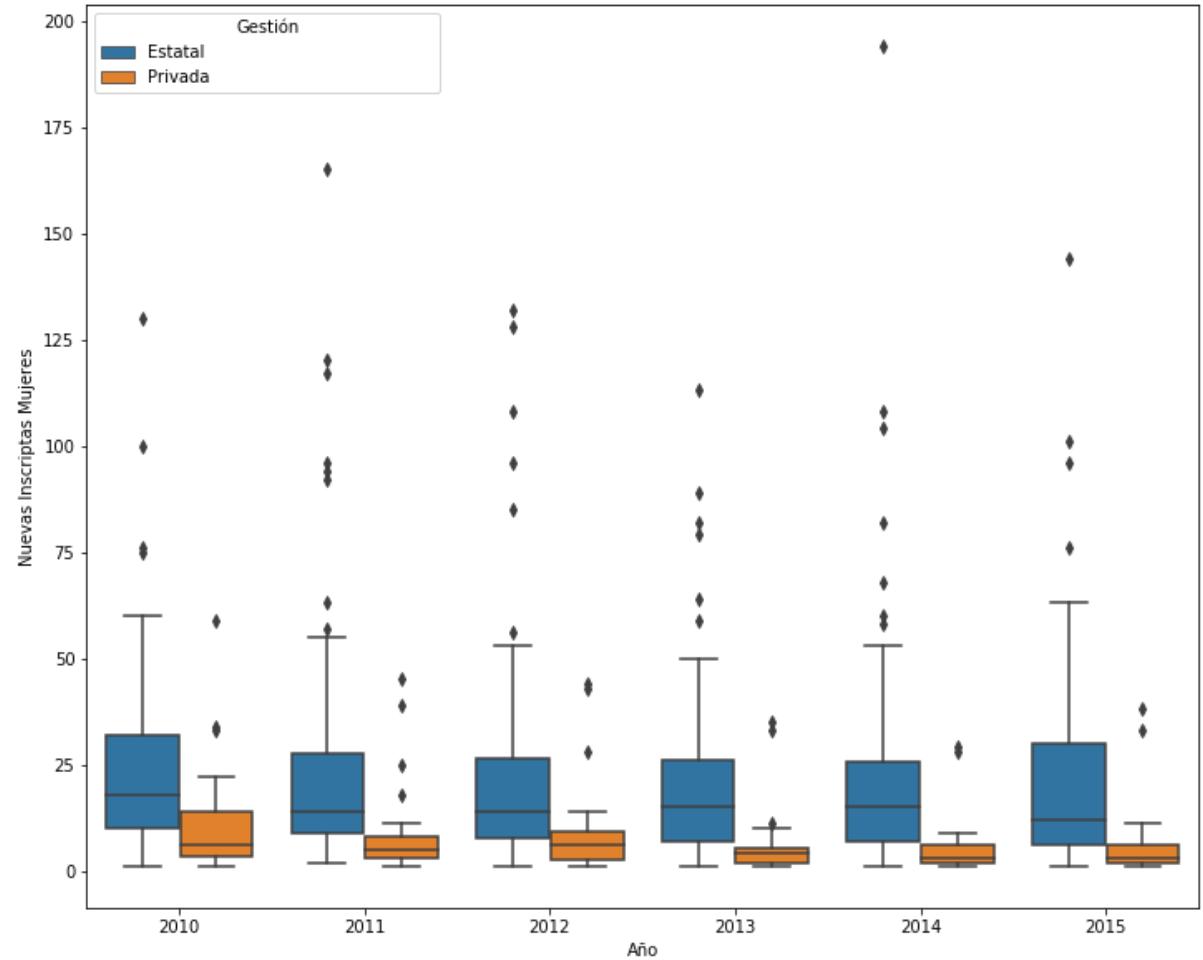
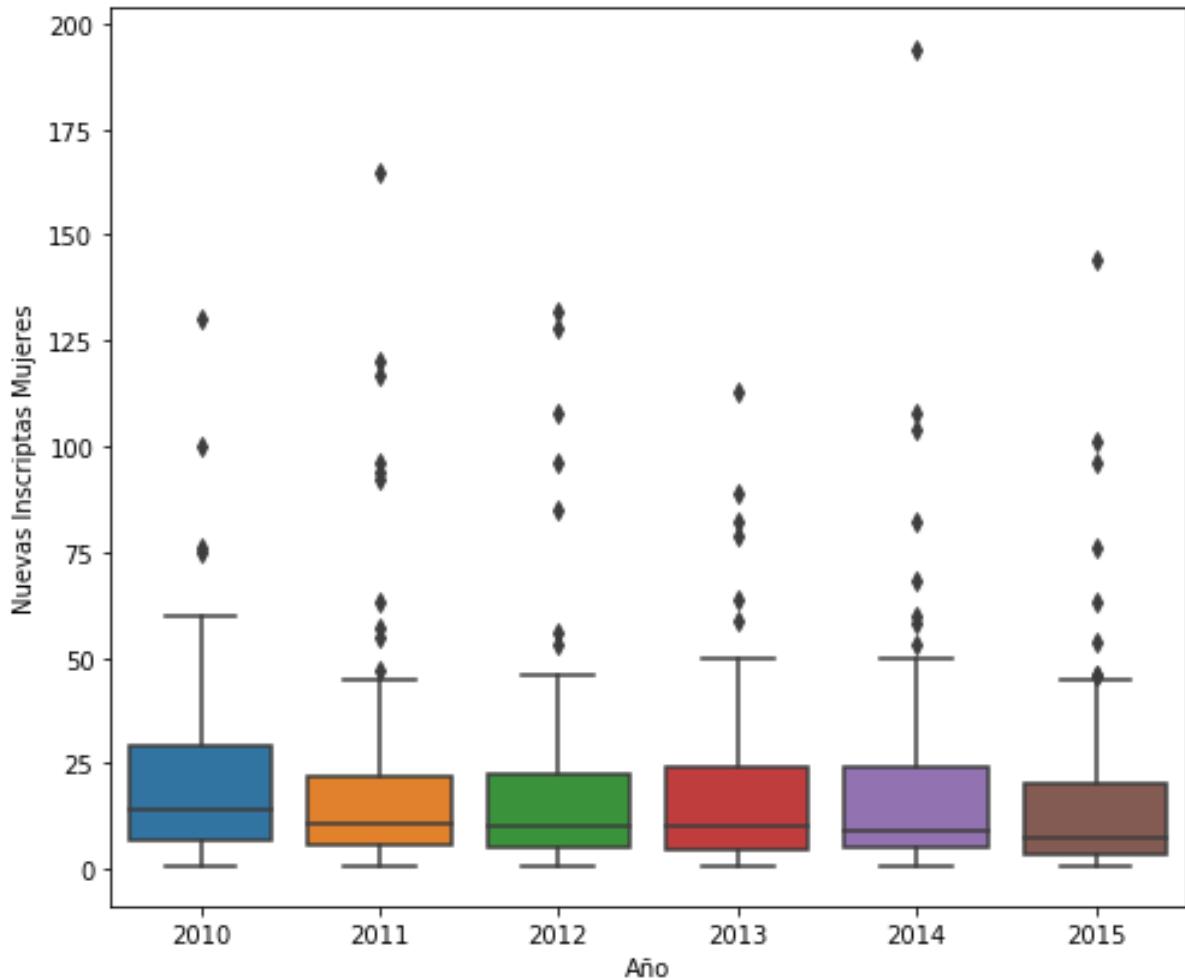
Los don't



Box plot o gráfico de cajas

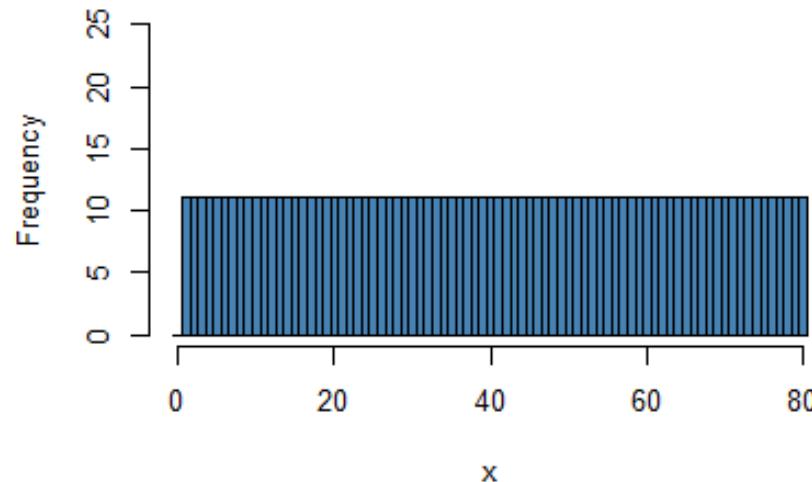


Box plot o gráfico de cajas

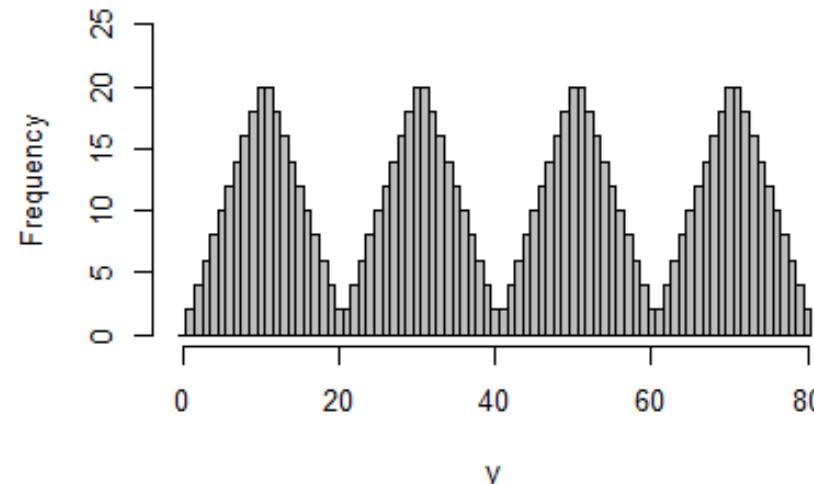


Boxplot VS histogram

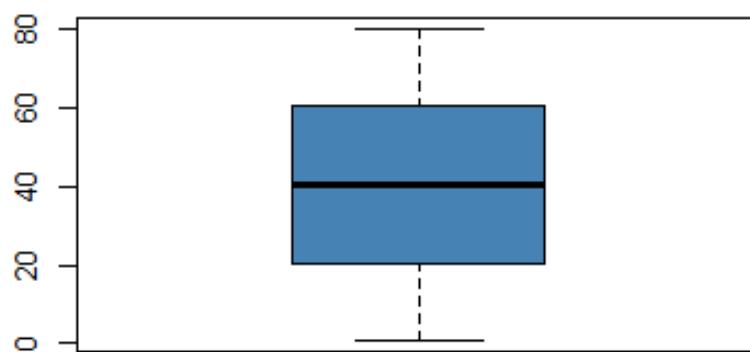
Histogram of x



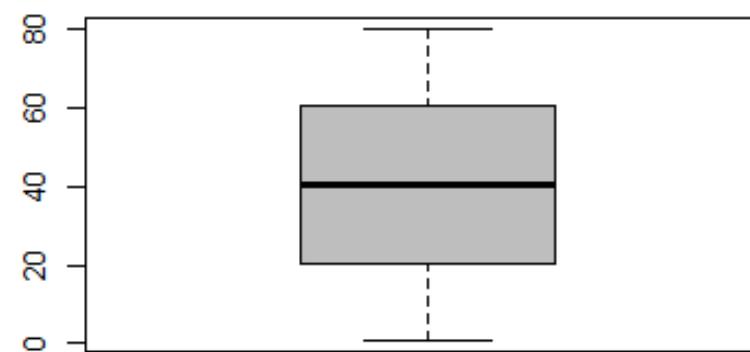
Histogram of y



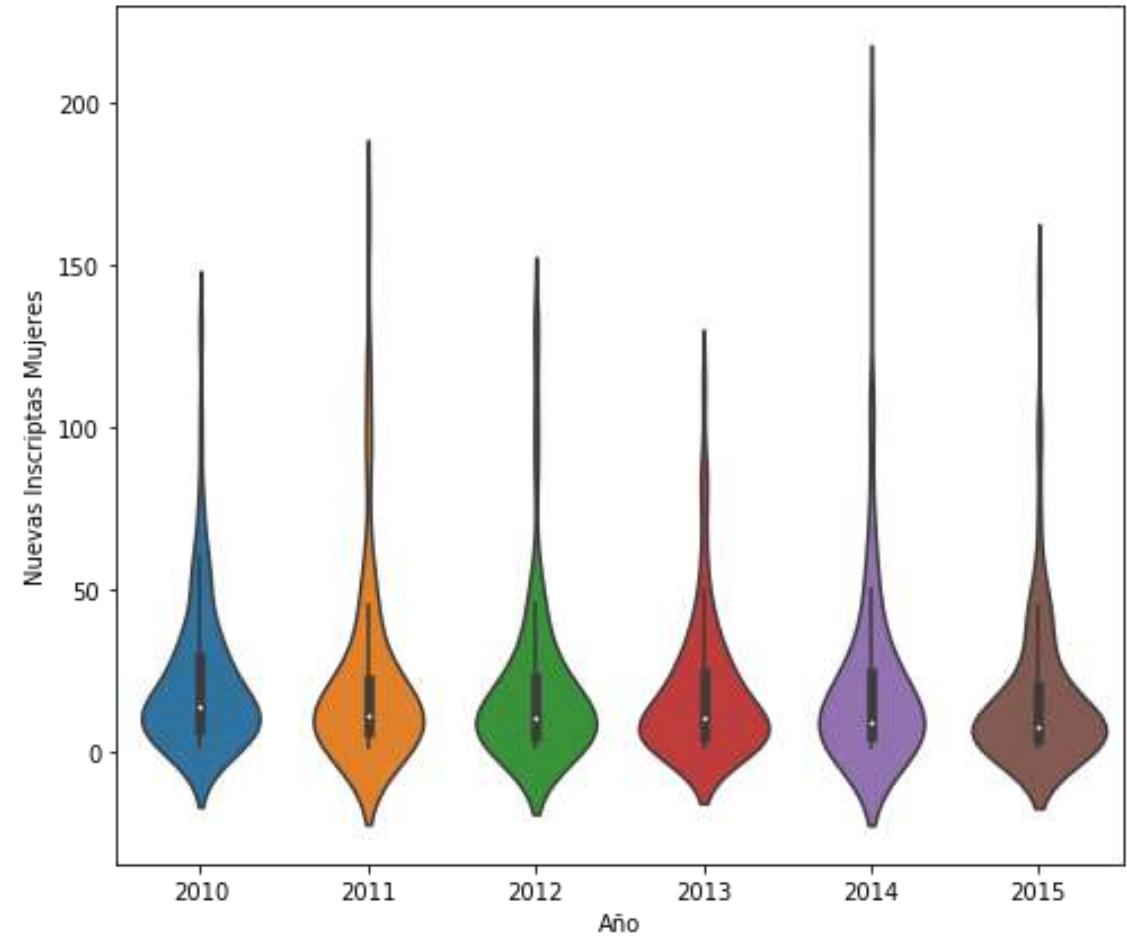
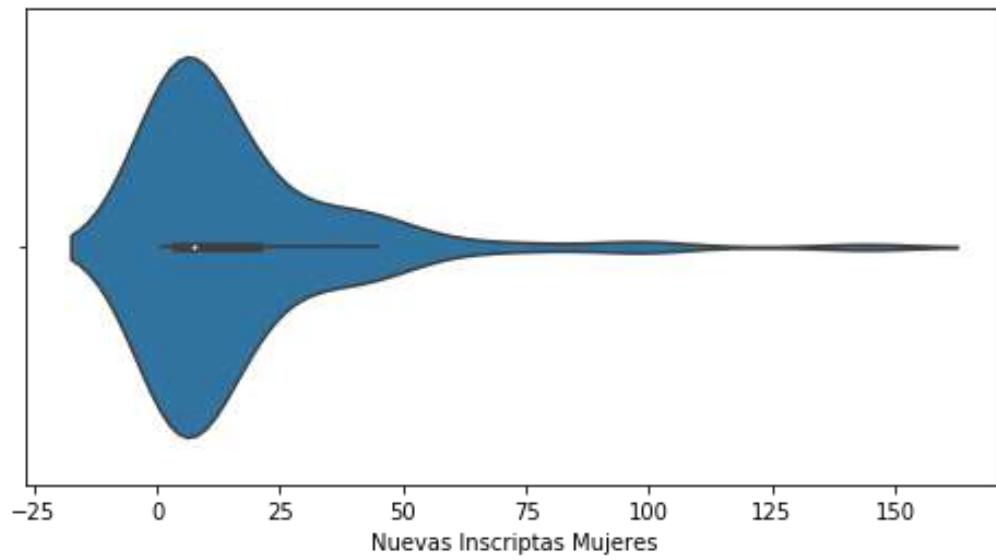
Boxplot of x



Boxplot of y

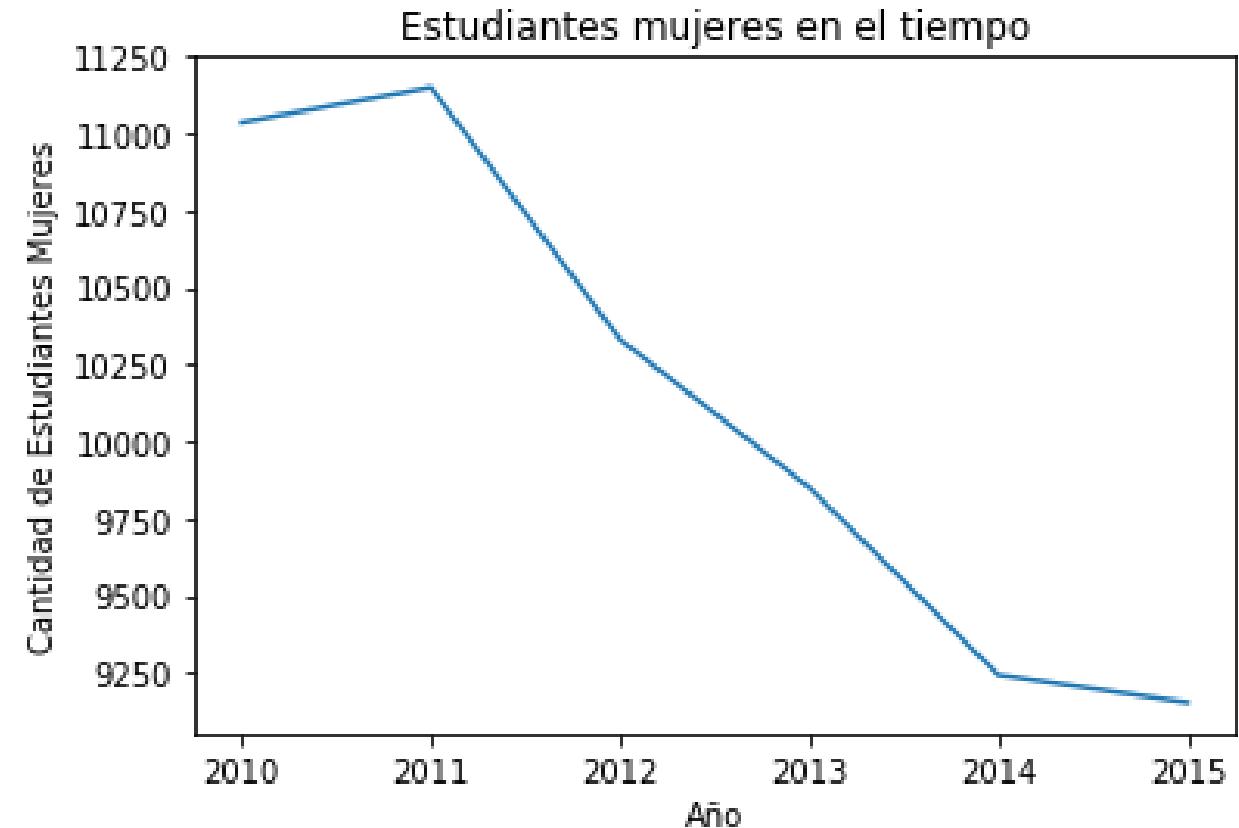
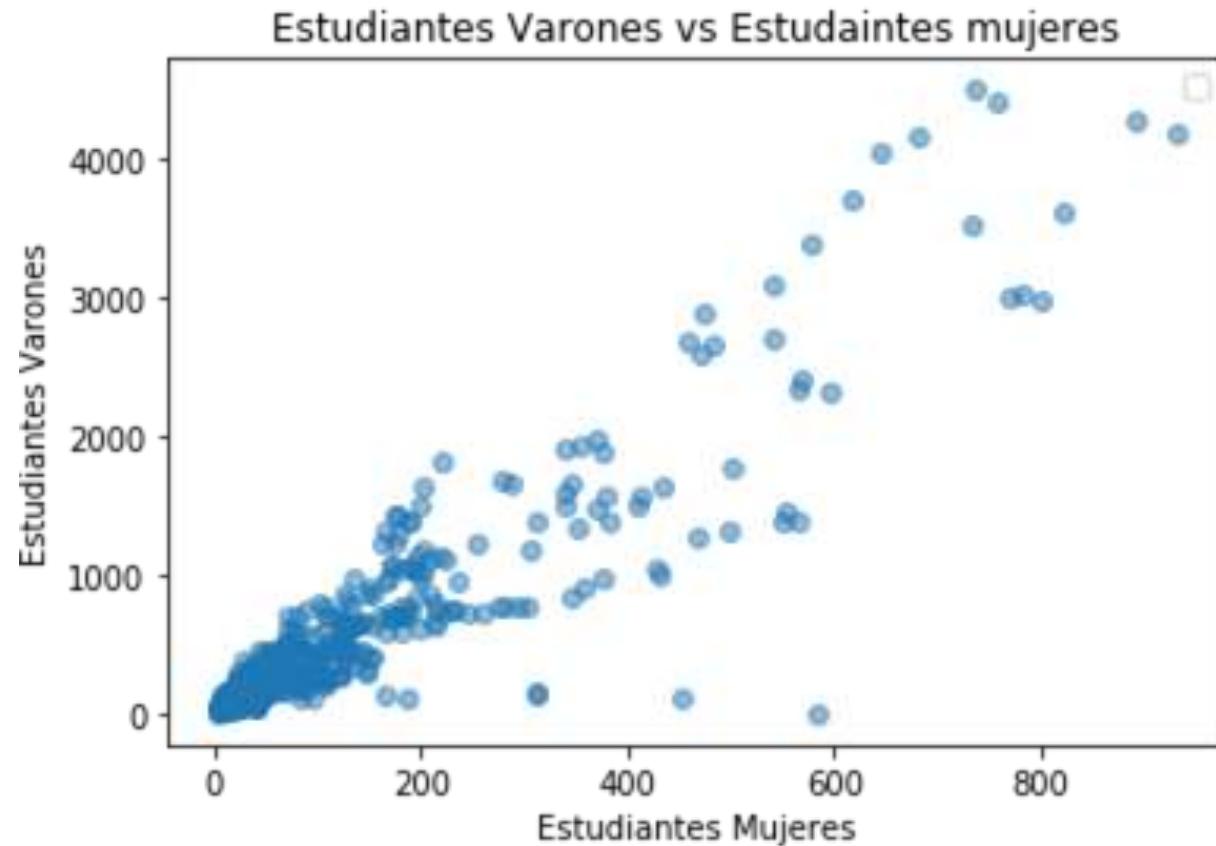


Boxplot + histogram = violinplot

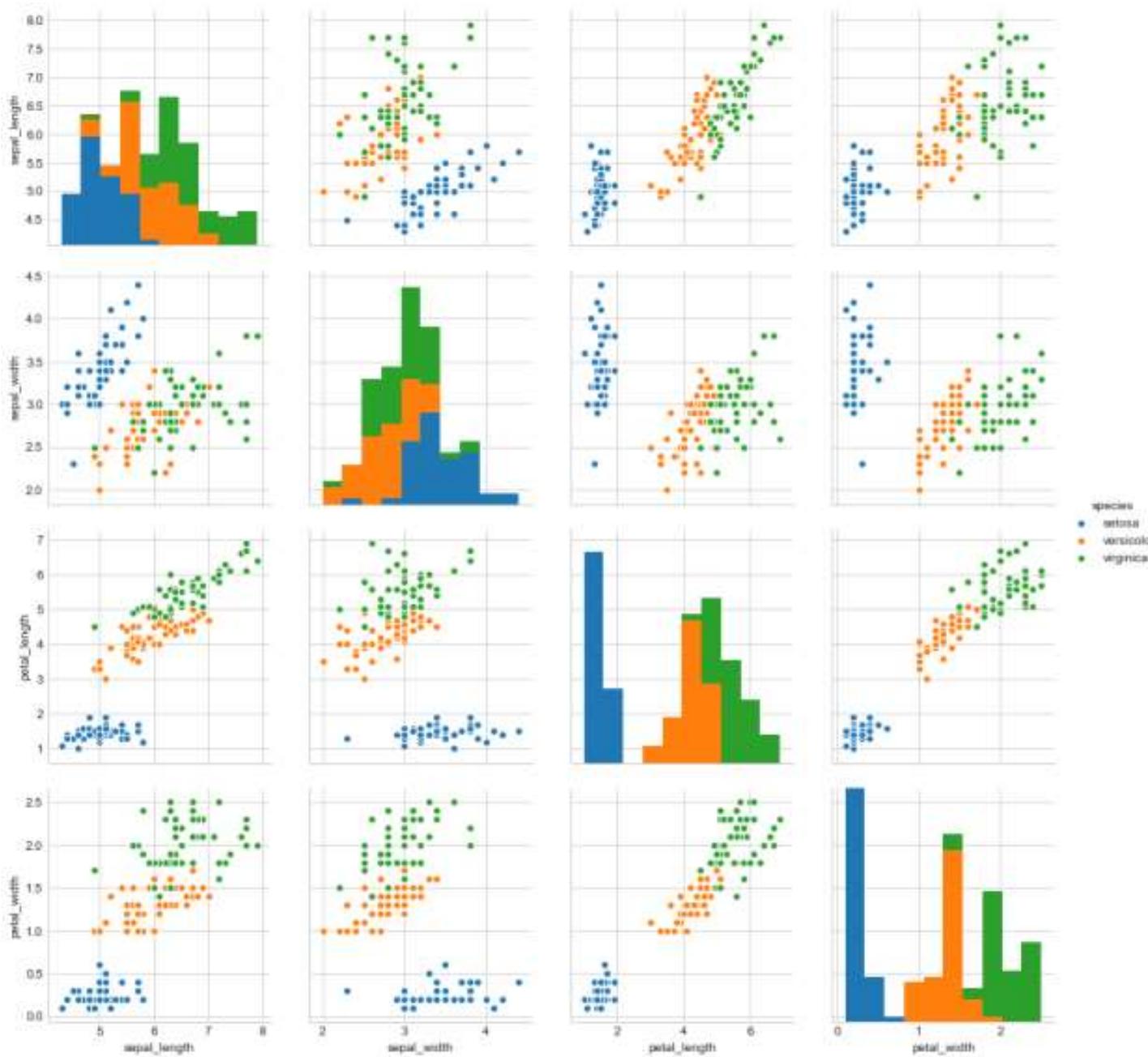


Scatter plot o gráfico de dispersión

Line plot o gráfico de líneas



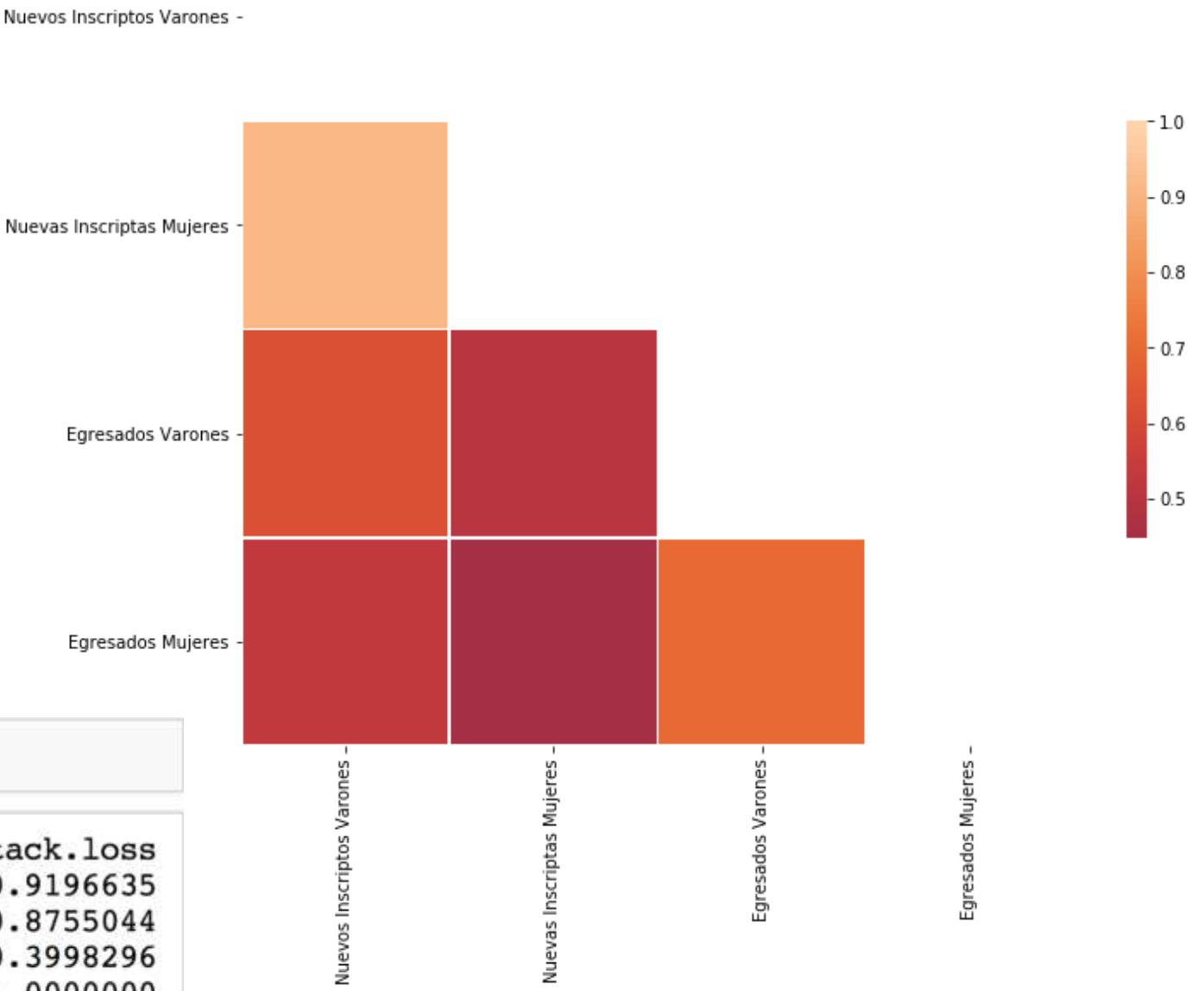
Faceting: pair plot



Correlation matrix

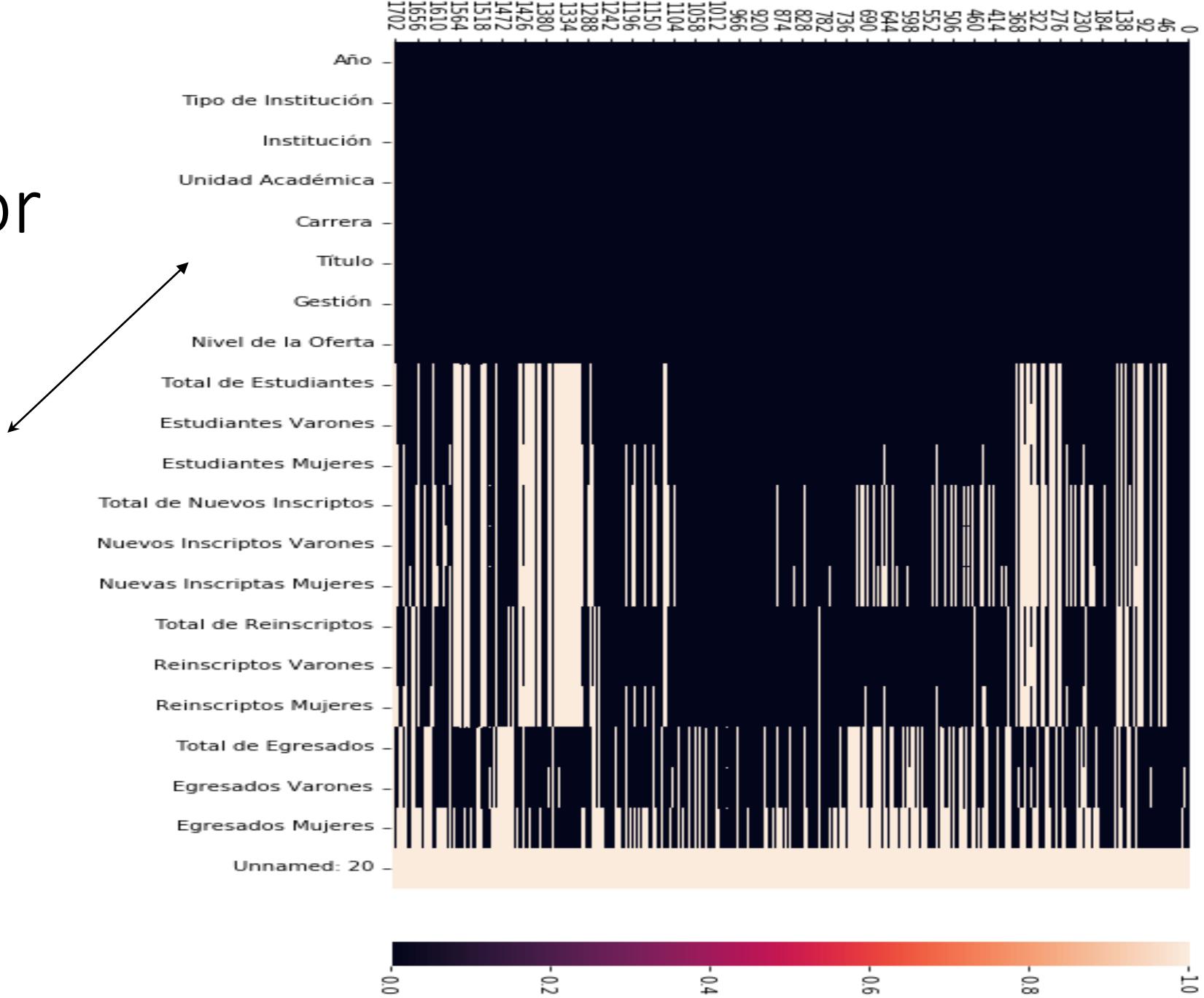
```
cor(stackloss)
```

	Air.Flow	Water.Temp	Acid.Conc.	stack.loss
Air.Flow	1.0000000	0.7818523	0.5001429	0.9196635
Water.Temp	0.7818523	1.0000000	0.3909395	0.8755044
Acid.Conc.	0.5001429	0.3909395	1.0000000	0.3998296
stack.loss	0.9196635	0.8755044	0.3998296	1.0000000



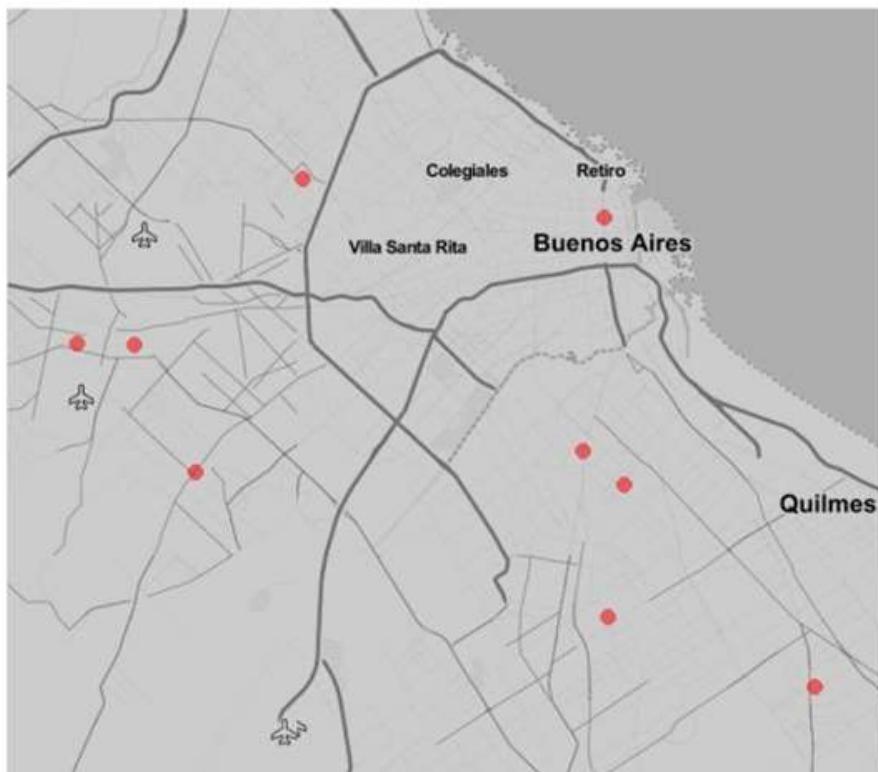
Heat maps o mapas de calor

```
sexo_id  
edad  
maximo_grado_academico_id  
disciplina_maximo_grado_academico_id  
disciplina_titulo_grado_id  
disciplina_experticia_id  
tipo_personal_id  
producciones_ult_anio  
producciones_ult_2_anios  
producciones_ult_3_anios  
producciones_ult_4_anios  
categoria_conicet_id  
max_dedicacion_horaria_docente_id  
produccion_cantidad_articulos_total  
produccion_cantidad_articulos_SJR_Q1  
produccion_cantidad_articulos_SJR_Q2  
produccion_cantidad_articulos_SJR_Q3  
produccion_cantidad_articulos_SJR_Q4  
produccion_patentes_solicitadas  
proyectos_financiamiento_externo  
score_evaluaciones  
score_estancias_exterior  
score_formacion  
score_otras_producciones  
dtype: int64
```

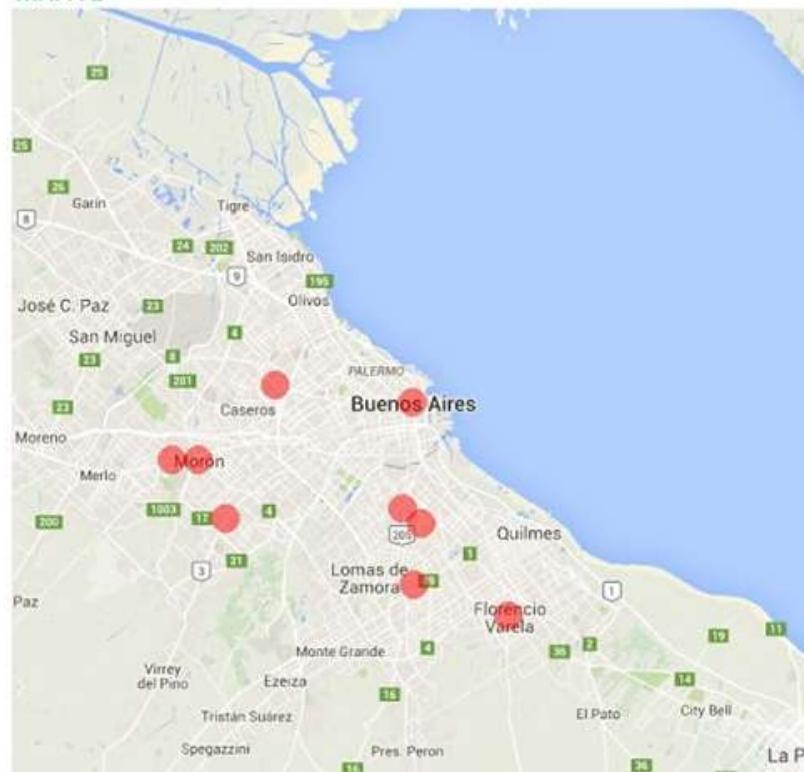


Map Plots

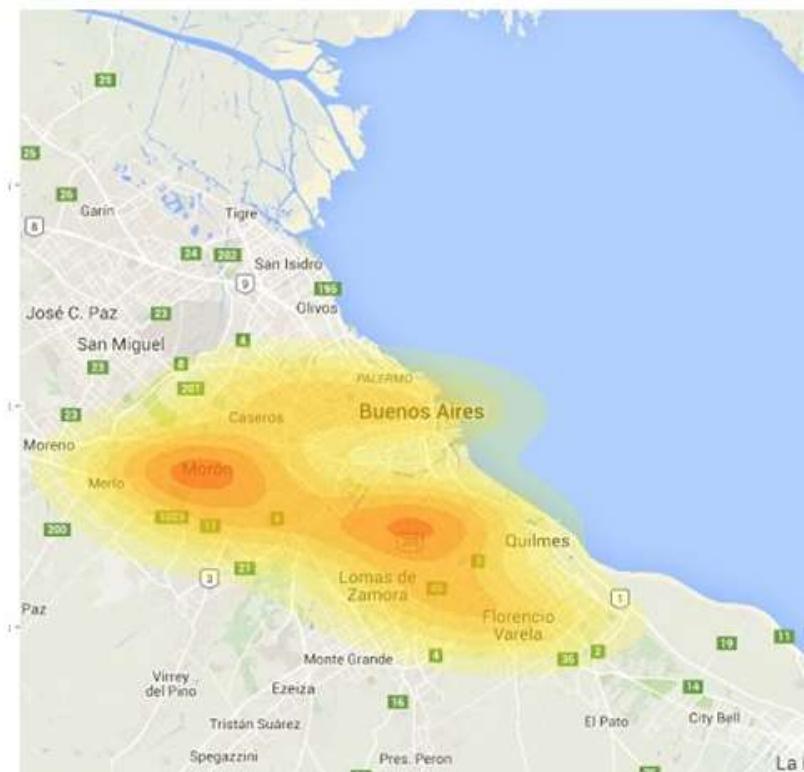
MAPA 1



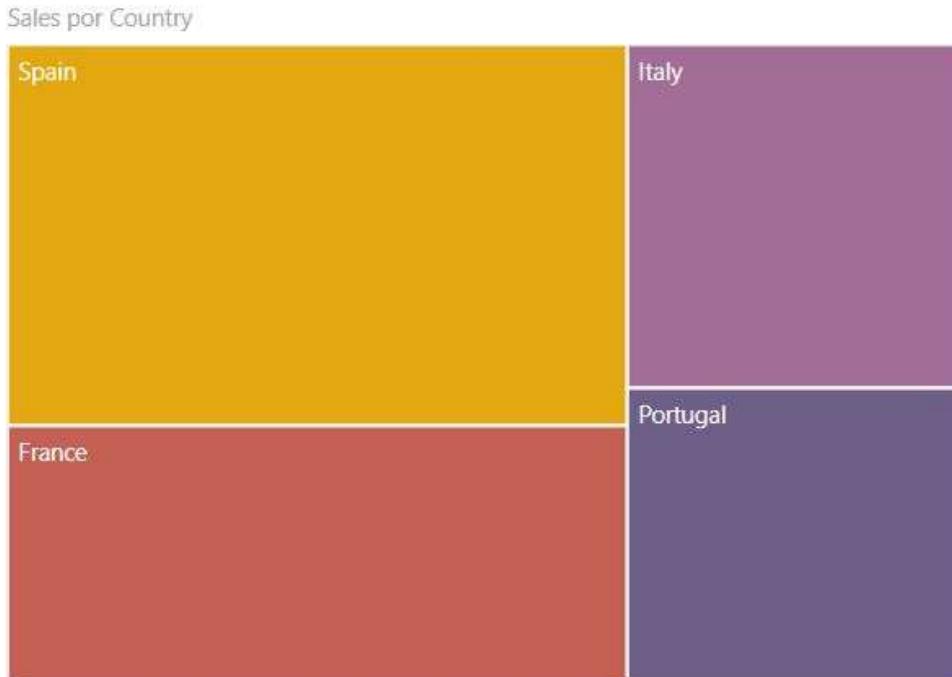
MAPA 2



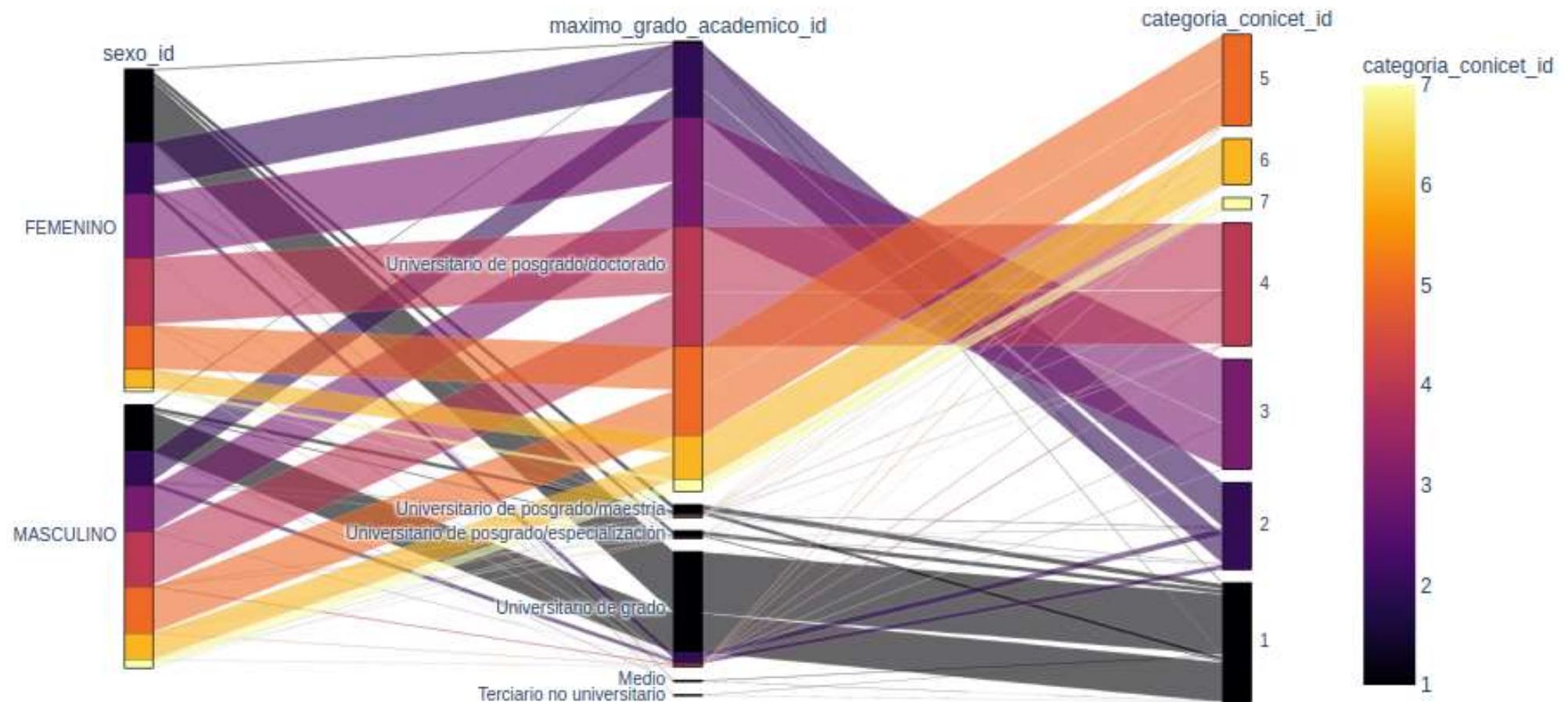
MAPA 3

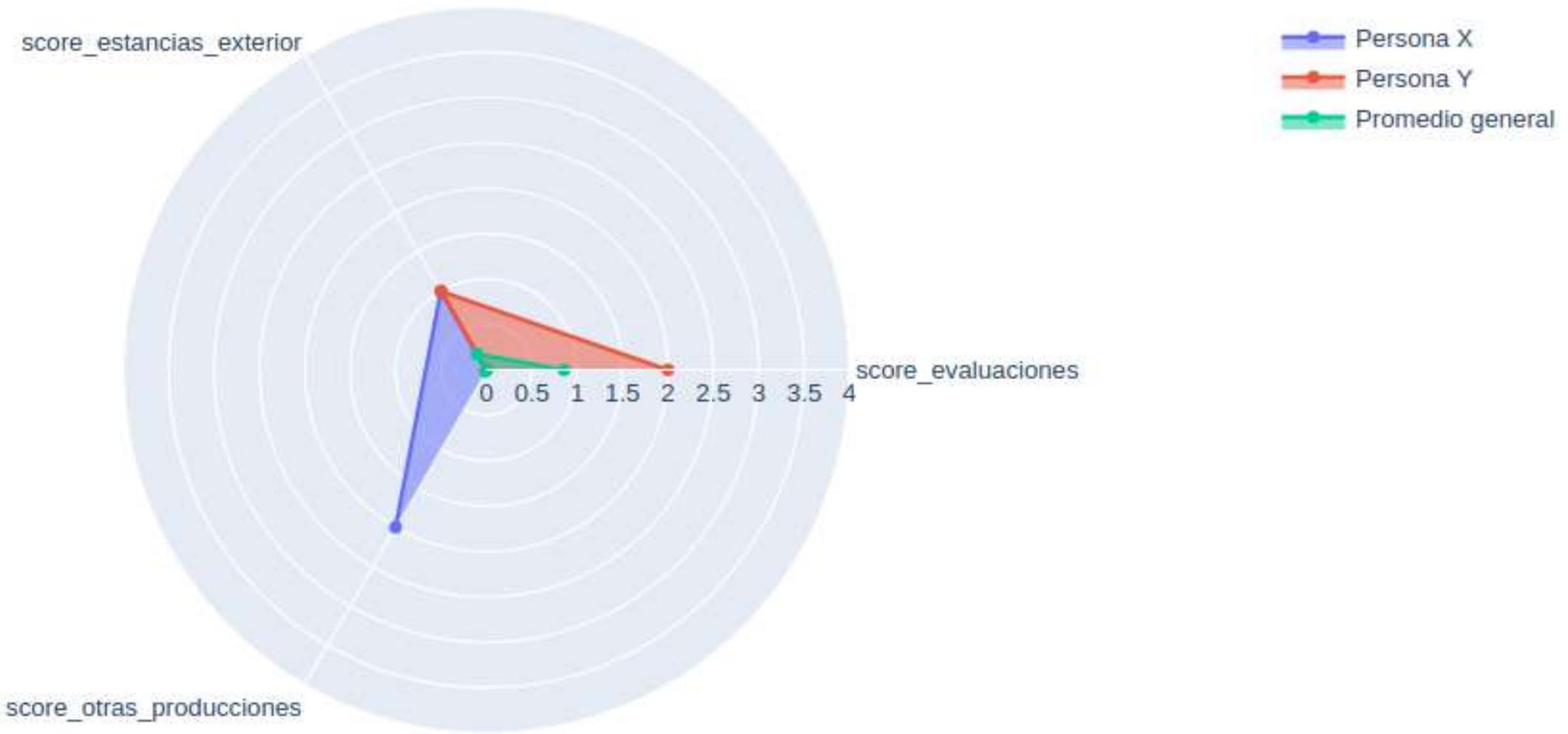


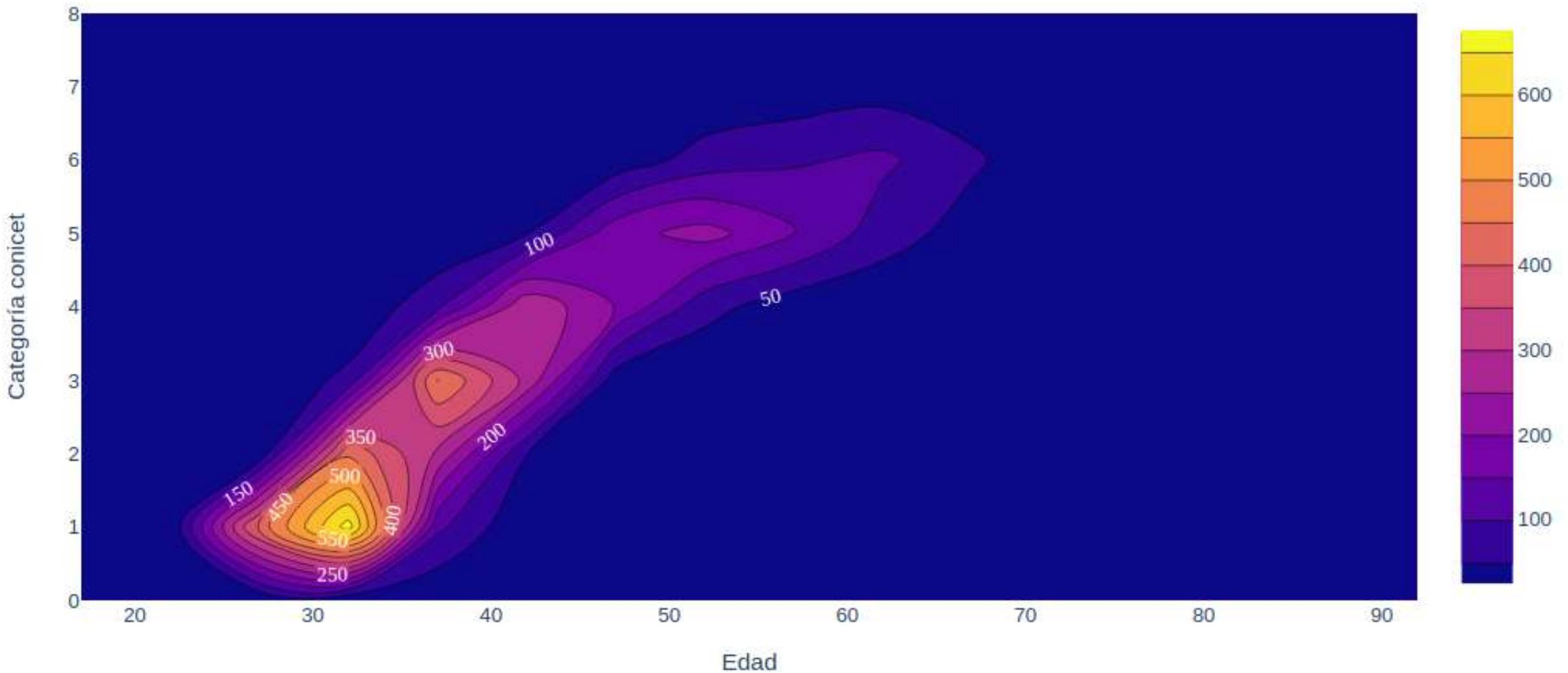
Tree Map

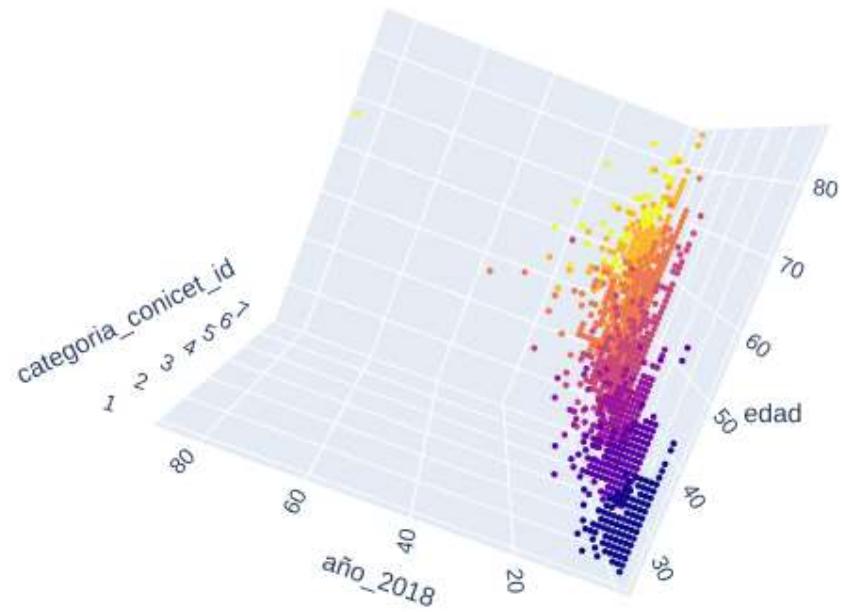


Y el grado de complejidad lo decide uno







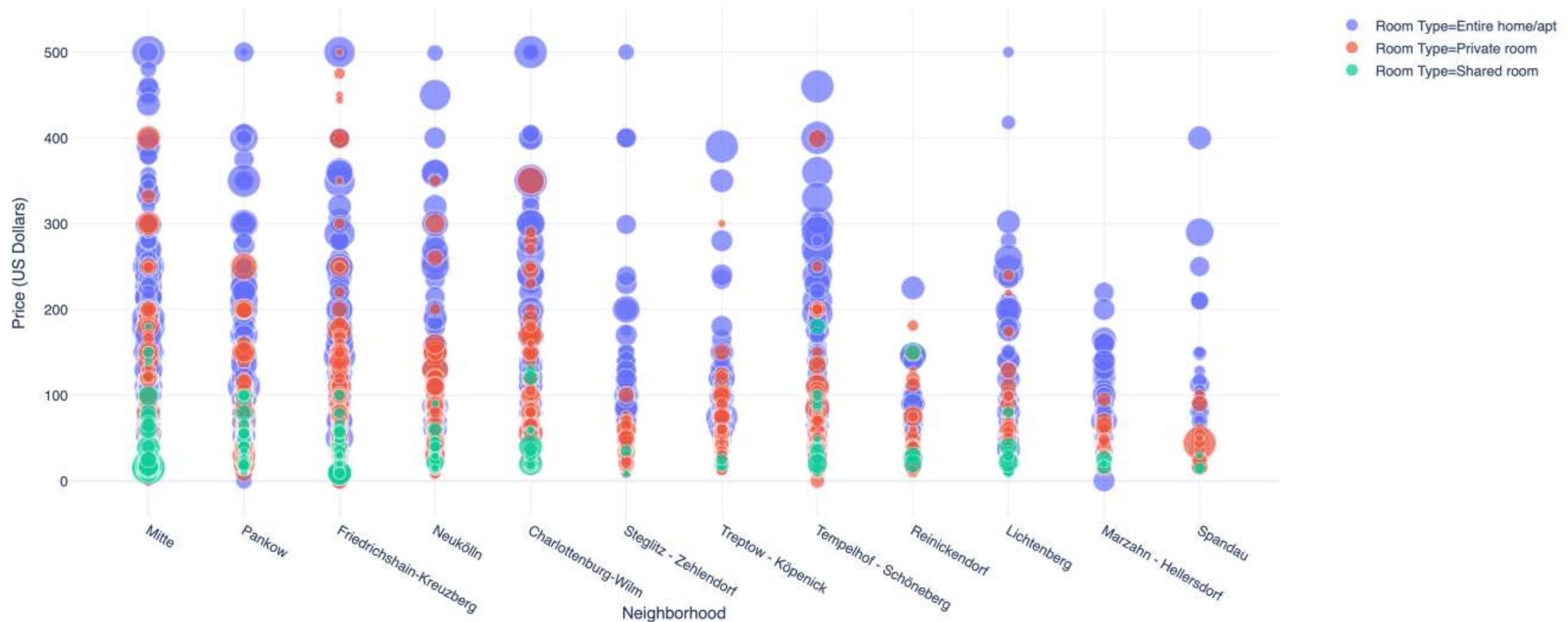


	año_2015	año_2016	año_2017	año_2018
Agricultura, Silvicultura y Pesca	2.3	2.1	1.8	1.1
Arte	0.75	0.54	0.42	0.073
Biotecnología Agropecuaria	2.8	2.7	2.5	1.8
Biotecnología Industrial	3.6	3.1	2.7	2.4
Biotecnología de la Salud	3.1	3.2	2	1.3
Biotecnología del Medio Ambiente	3	2.6	2.8	2
Ciencia Política	1.4	1.3	1.1	0.75
Ciencias Biológicas	2.8	2.3	2	1.3
Ciencias Físicas	2.3	1.8	1.5	0.75
Ciencias Químicas	2.8	2	2.1	1.1
Ciencias Veterinarias	2.5	2.2	1.7	1
Ciencias de la Computación e Información	1.5	1.4	1.2	0.51
Ciencias de la Educación	1.3	1	0.86	0.53
Ciencias de la Salud	1.5	1.3	1	0.53
Ciencias de la Tierra y relacionadas con el Medio Ambiente	2.4	2.3	1.9	1.1
Comunicación y Medios	1.1	0.81	0.68	0.4
Derecho	0.9	0.81	0.74	0.43
Economía y Negocios	0.79	0.73	0.71	0.35
Filosofía, Ética y Religión	1.9	1.4	1.2	0.68
Geografía Económica y Social	1.4	1.3	1.2	0.67
Historia y Arqueología	2.2	1.8	1.4	0.9
Ingeniería Civil	1.4	0.99	0.71	0.38
Ingeniería Eléctrica, Ingeniería Electrónica e Ingeniería de la Información	1.3	1.3	0.91	0.48
Ingeniería Mecánica	1	1.4	1.2	0.55
Ingeniería Médica	1.3	0.8	1.1	0.31
Ingeniería Química	2.7	2	2	0.98
Ingeniería de los Materiales	3.8	3.1	2.3	1.4
Ingeniería del Medio Ambiente	2.1	1.7	1.4	0.88
Lengua y Literatura	1.5	1.1	0.82	0.56
Matemáticas	1.1	0.92	0.82	0.54
Medicina Básica	3.4	3	2.4	1.4
Medicina Clínica	1	0.96	0.62	0.28
Nanotecnología	3.1	3.6	2.6	2.5
Otras Ciencias Agrícolas	1.8	1.7	1.4	0.74
Otras Ciencias Médicas	0.71	0.62	0.48	0.16
Otras Ciencias Naturales y Exactas	2.5	1.9	1.4	0.87
Otras Ciencias Sociales	1.3	0.95	0.82	0.56
Otras Humanidades	1.2	1	0.78	0.37
Otras Ingenierías y Tecnologías	2.1	1.9	1.5	0.89
Producción Animal y Lechería	2	1.5	1.2	0.76
Psicología	1.6	1.1	0.73	0.32
Sociología	1.9	1.5	1.4	0.93



Gráficos Interactivos

How much should you expect to pay or charge in a Berlin neighborhood?



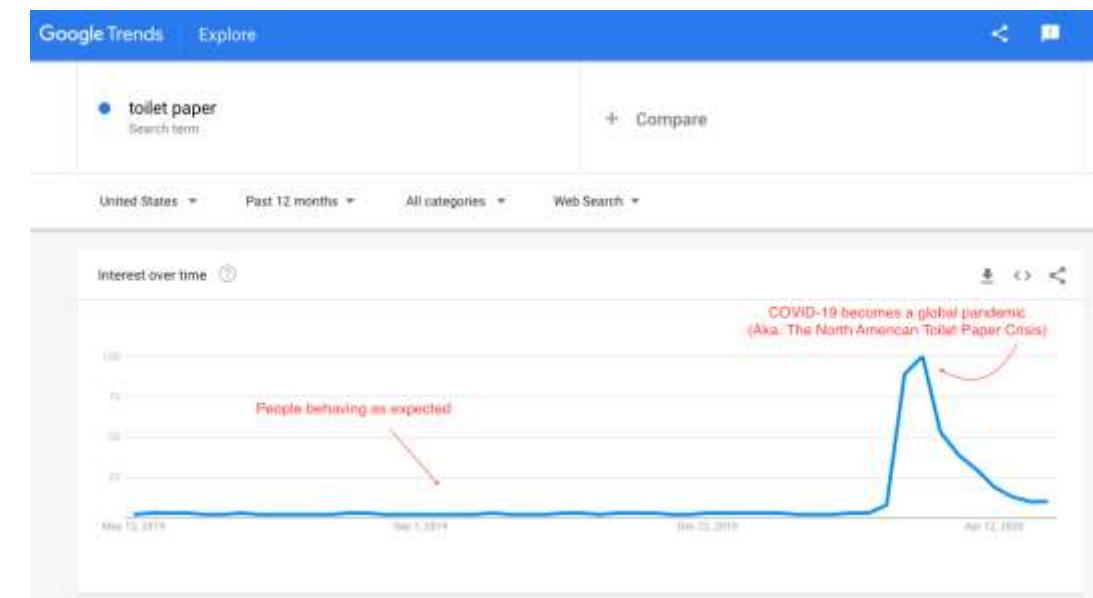
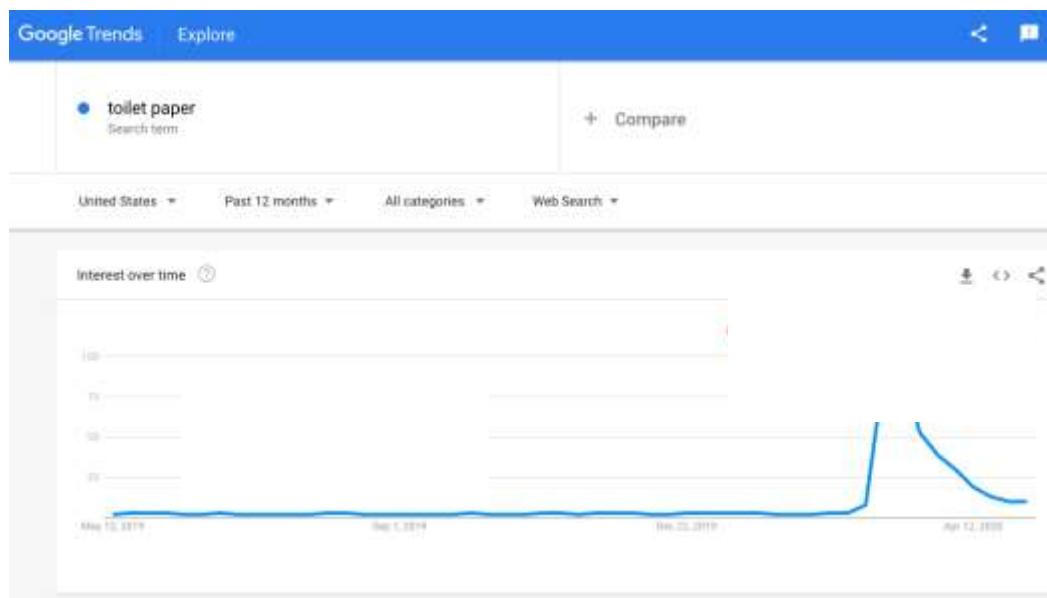
Tips para Visualizaciones

1. Hacer que las visualizaciones sean fácilmente legibles.



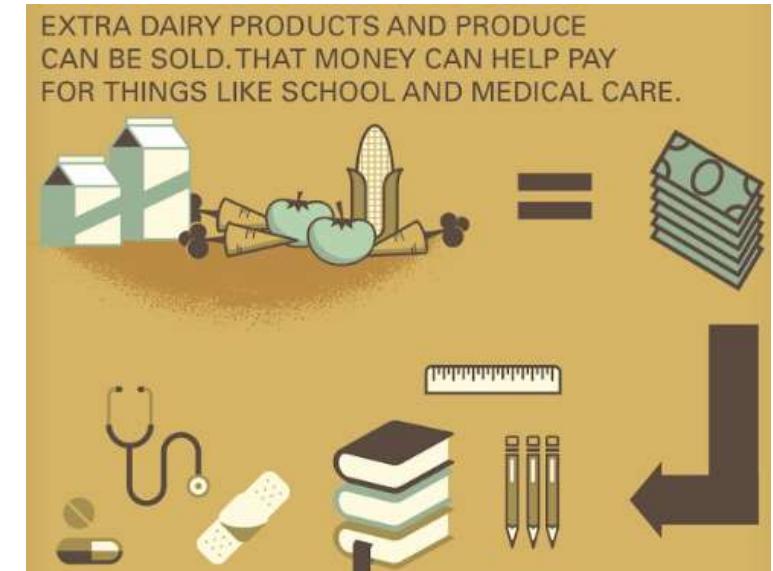
Tips para Visualizaciones

2. Identificar datos atípicos, en caso que sean relevantes.



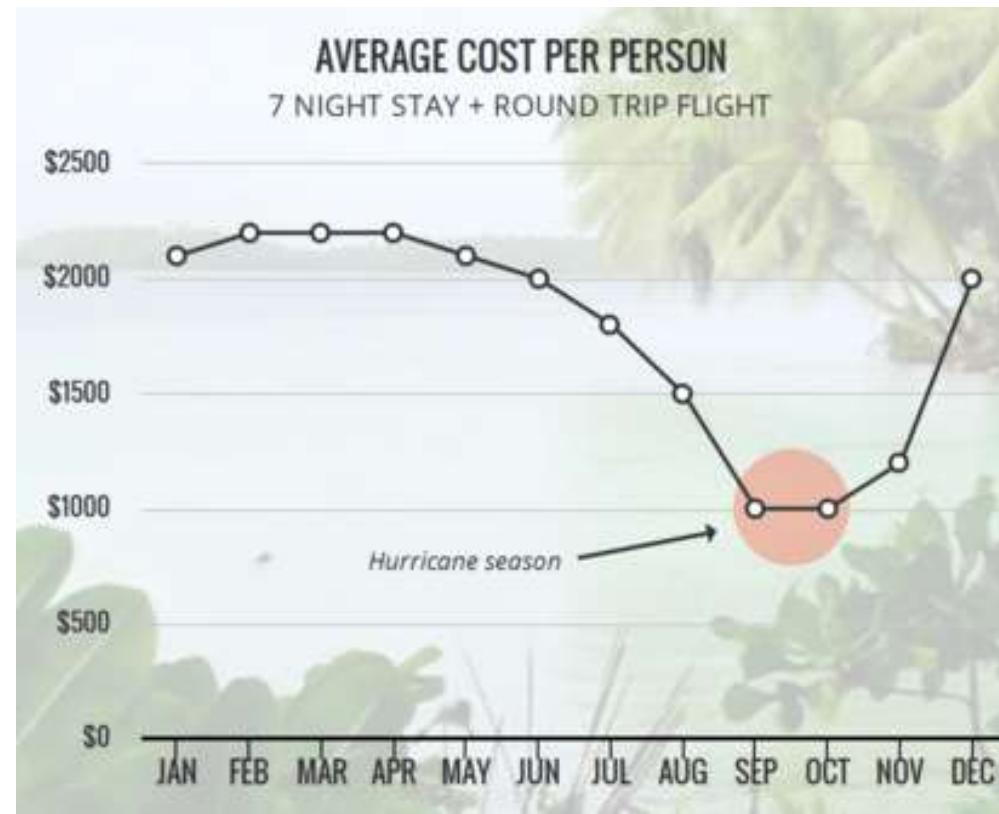
Tips para Visualizaciones

3. Contar una historia con los datos



Tips para Visualizaciones

4. Reforza la opinión o argumento que queres expresar



Tips para Visualizaciones

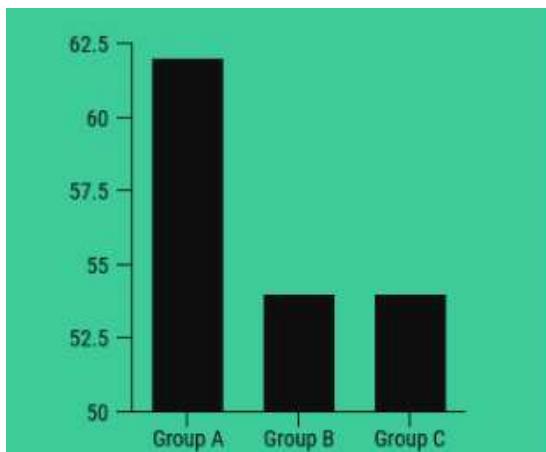
5. Usar negrita y colores para resaltar lo que queremos mostrar.

6. Centrarse en estadísticas clave



¿Qué no hacer?

1. Sesgar los datos a través del eje.



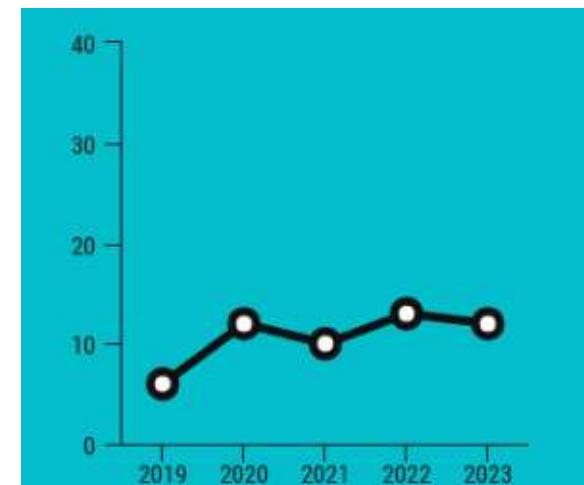
MISLEADING

- Starting the vertical axis at 50 makes a small difference between groups seem massive
- Group A looks much larger than Groups B and C

VS

ACCURATE

- Starting the vertical axis at 0 offers a more accurate depiction of the data
- The difference between the groups does not seem as dramatic



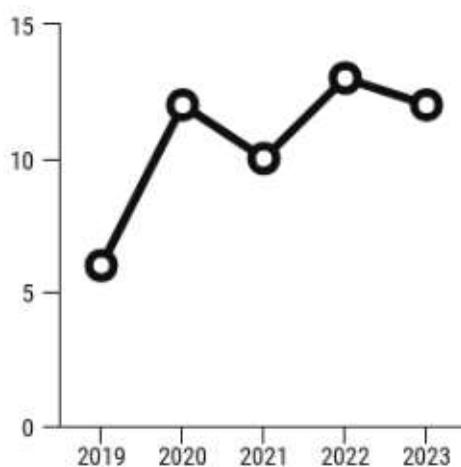
MISLEADING

- The scale is disproportionate to the data, making the change over time seem small

VS

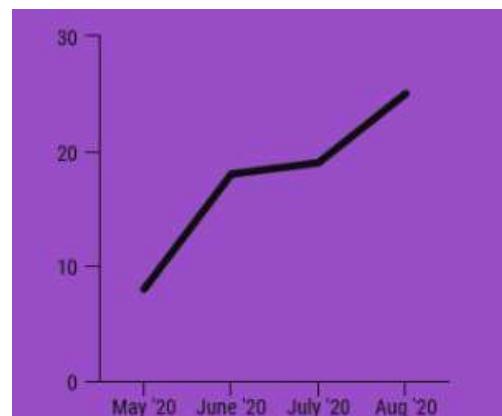
ACCURATE

- The scale is proportionate to the data, showing a greater change over time



¿Qué no hacer?

2. Elegir mal los datos, los gráficos y los colores



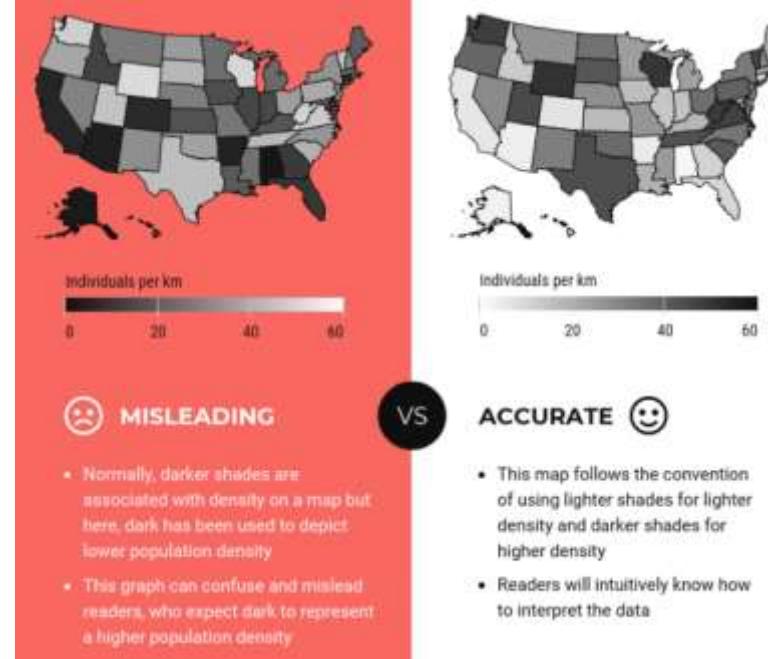
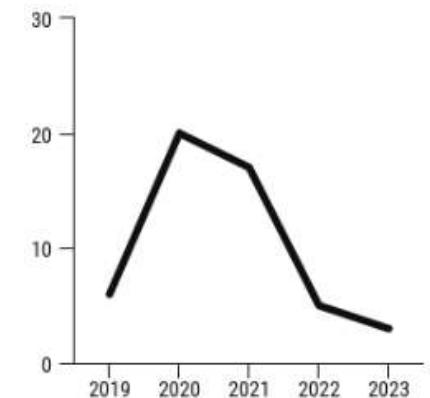
MISLEADING

- Only a few months out of the year are graphed, depicting an upward trend

VS

ACCURATE

- A much wider date range is graphed, revealing an overall downward trend
- This graph shows the bigger picture



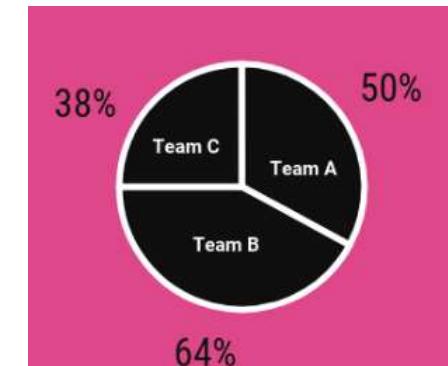
MISLEADING

- Normally, darker shades are associated with density on a map but here, dark has been used to depict lower population density
- This graph can confuse and mislead readers, who expect dark to represent a higher population density

VS

ACCURATE

- This map follows the convention of using lighter shades for lighter density and darker shades for higher density
- Readers will intuitively know how to interpret the data



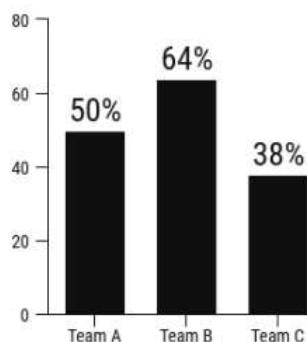
MISLEADING

- Pie charts are used to compare parts of a whole, not the difference between groups
- A different type of graph should be used to compare the three teams

VS

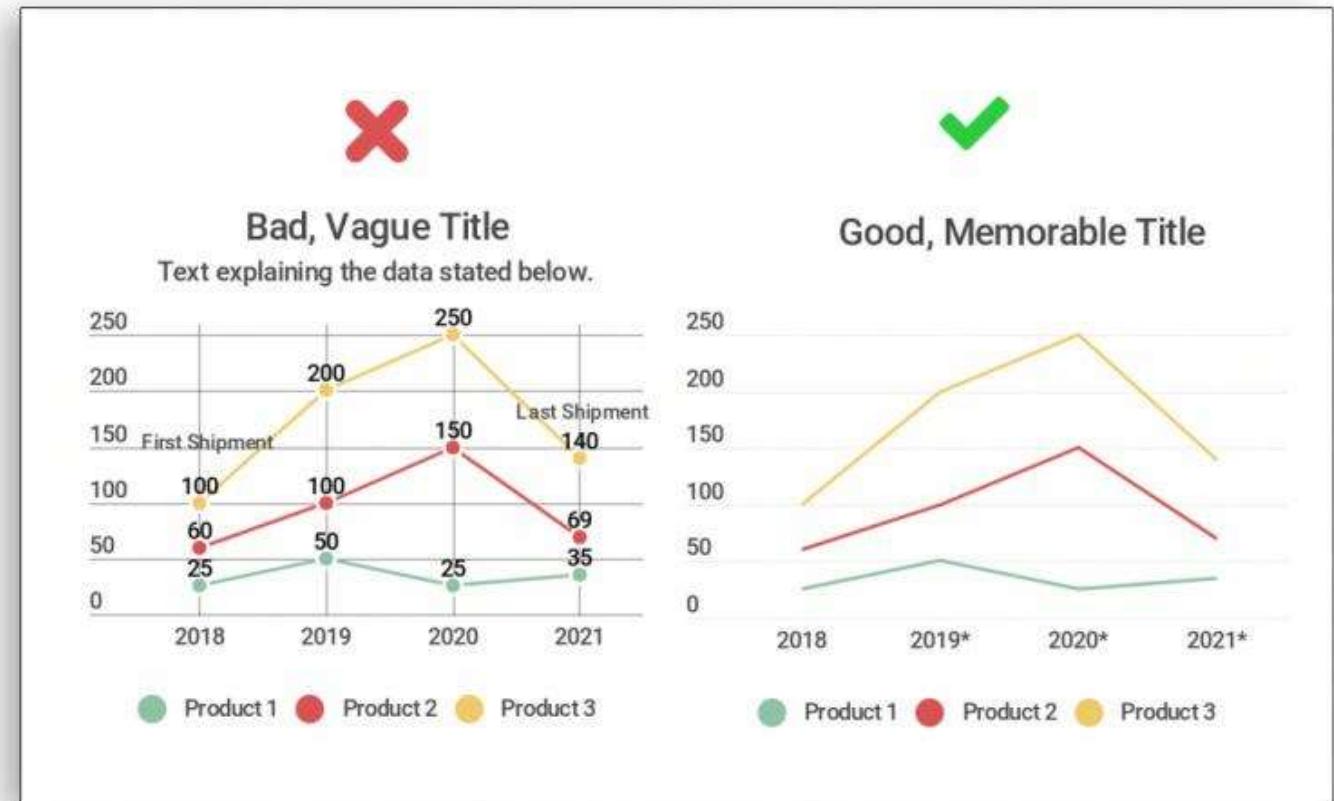
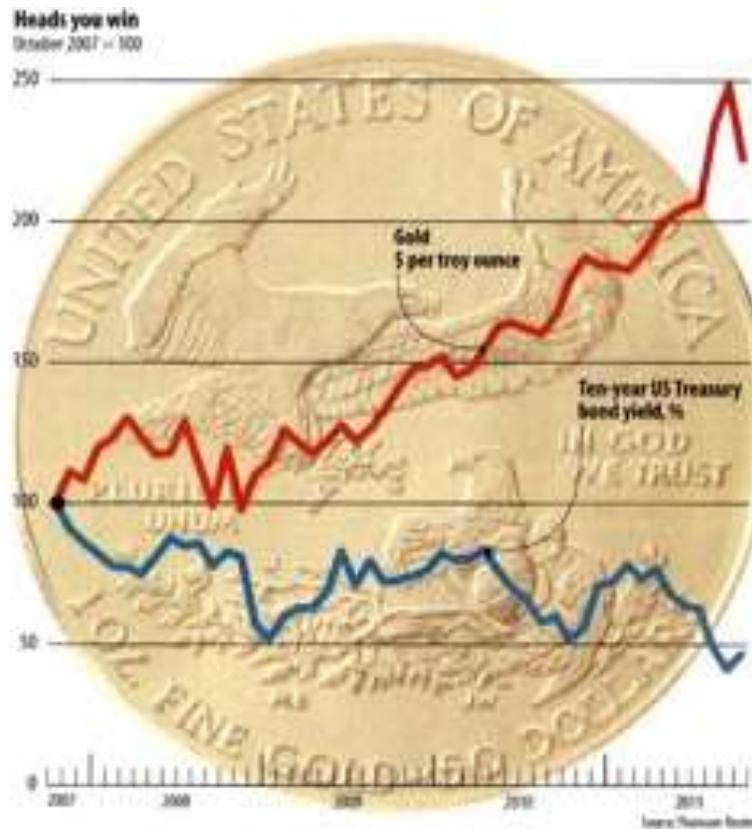
ACCURATE

- Bar graphs are better for showing the differences between groups
- This chart is a better visualization of the data



¿Qué no hacer?

3. Complicar los gráficos con datos innecesarias.



Material interesante

- <https://www.data-to-viz.com/>
- <https://python-graph-gallery.com/>
- <https://www.youtube.com/watch?v=KvZ2KSxIWBY> (Stephen Elston - Data Visualization and Exploration with Python)
- <https://link.springer.com/book/10.1007/978-3-319-64410-3> (Probability and Statistics for Computer Science)
- <https://library.educause.edu/-/media/files/library/2007/10/eli7030-pdf.pdf>

MAPA DE CONCEPTOS CLASE 7

¡Para recordar!



CRONOGRAMA DEL CURSO

Clase 6



Introducción a la
programación con
Python (Parte II)



¡VAMOS AL CÓDIGO



PRÁCTICA INTEGRADORA:
PANDAS Y SERIES DE
TIEMPO

Clase 7



Visualizaciones en
Python (Parte I)



¡VAMOS AL CÓDIGO

Clase 8



Visualizaciones en
Python (Parte II)



PRÁCTICA INTEGRADORA:
VISUALIZACIÓN EN PYTHON

*INTRODUCCIÓN A LA
VISUALIZACIÓN EN
PYTHON*

La librería Matplotlib



¿Por qué Matplotlib?

- Es la librería de visualización **más utilizada** en el entorno de Python.
- Es **sencilla** y fácil de usar.
- Permite un alto nivel de **personalización** de los gráficos.
- Es open source.
- Es la base sobre la que se construyen otras librerías como Seaborn.

matplotlib

CODER HOUSE



Interfaces de Matplotlib

- Definen la **forma en la que interactuamos** con el gráfico.
- Proveen compatibilidad con el lenguaje que inspiró la librería: MATLAB
- Existen dos interfaces disponibles:
 - **Interfaz orientada a estados**, orientada a usuarios de MATLAB para mantener compatibilidad.
 - **Interfaz orientada a objetos**: Permite **mayor grado de control** sobre los gráficos porque los tratamos como **objetos**.
Más Pythonista para nuestro gusto (y la más utilizada)

Del lado Python de la vida! 🐍



Setup para la clase

Antes que nada...

- Importemos las librerías que usaremos en la clase.

```
import matplotlib as mpl  
import matplotlib.pyplot as plt  
import seaborn as sns  
import pandas as pd
```

- Los estilos por defecto de matplotlib no son muy estéticos.
Podemos cambiar el estilo fácilmente.

```
mpl.style.use('bmh')
```



Primera toma de contacto

- Grafiquemos una línea que une los puntos con coordenadas $(x,y) = (1, 2)$ y $(x,y) = (3, 4)$. Necesitamos:
 - Una arreglo con las dos coordenadas del eje x [1, 3]
 - Una arreglo con las dos coordenadas del eje y [2, 4]

Interfaz orientada a objetos

```
fig, ax = plt.subplots()  
ax.plot([1, 3], [2, 4])
```

Interfaz orientada a estados:

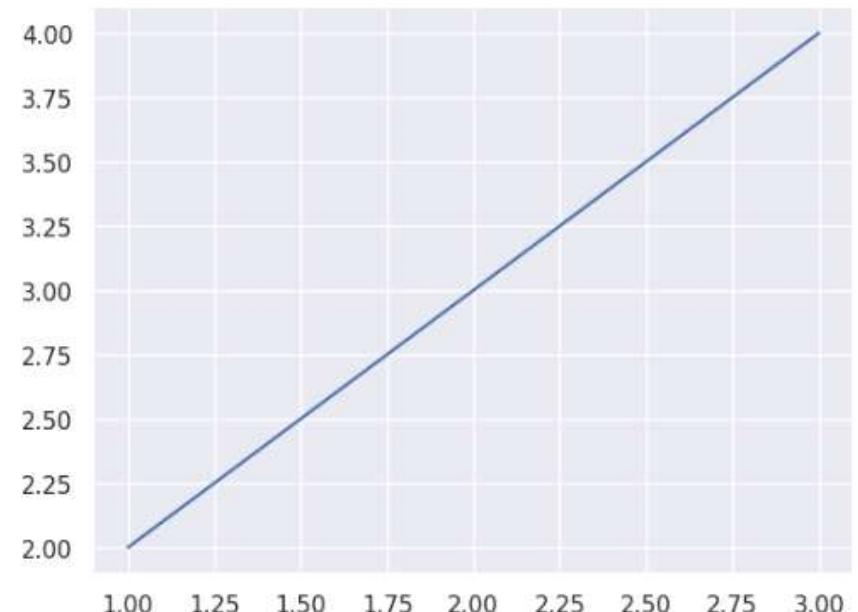
```
plt.plot([1, 3], [2, 4])
```



Primera toma de

- Ambas formas retornan el mismo resultado.
- La interfaz orientada a estados parece más simple, pero al hacer gráficos más complejos y profesionales es **más difícil** de implementar.
- Para evitar confusión, usaremos y recomendaremos la interfaz **orientada a objetos**.

contacto

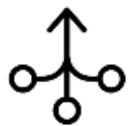




Comandos básicos

... para contextualizar los gráficos

- Cuando generemos visualizaciones, una buena práctica es **procurar siempre incluir información** acerca de lo que se muestra.
 - **Etiquetar los ejes** `ax.set_xlabel` y `ax.set_ylabel`
 - **Añadir un título** `ax.set_title`
 - **Añadir una leyenda** `ax.legend`



Ejemplo de uso

- Importemos el Data Frame de precipitaciones de la clase pasada. Los datos están disponibles en [este enlace](#).

```
df_lluvias = pd.read_csv('<ruta>/pune_1965_to_2002.csv')
df_lluvias.head()
```

	Year	Jan	Feb	Mar	Apr	May
0	1965	0.029	0.069	0.000	21.667	17.859
1	1966	0.905	0.000	0.000	2.981	63.008
2	1967	0.248	3.390	1.320	13.482	11.116
3	1968	0.318	3.035	1.704	23.307	7.441
4	1969	0.248	2.524	0.334	4.569	6.213

- Por conveniencia, ponemos a Year como índice del Data Frame y lo eliminamos de las columnas

```
df_lluvias.index = df_lluvias['Year']
df_lluvias = df_lluvias.drop('Year', axis='columns')
```



Ejemplo de uso

- Tras el cambio de índice, los datos quedan de la siguiente manera:

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
1965	0.029	0.069	0.000	21.667	17.859	102.111	606.071	402.521	69.511	5.249	16.232	22.075
1966	0.905	0.000	0.000	2.981	63.008	94.088	481.942	59.386	150.624	1.308	41.214	4.132
1967	0.248	3.390	1.320	13.482	11.116	251.314	780.006	181.069	183.757	50.404	8.393	37.685
1968	0.318	3.035	1.704	23.307	7.441	179.872	379.354	171.979	219.884	73.997	23.326	2.020
1969	0.248	2.524	0.334	4.569	6.213	393.682	678.354	397.335	205.413	24.014	24.385	1.951



Ejemplo de uso

- Grafiquemos las precipitaciones acumuladas para los distintos años:

- Para el eje *x*, seleccionamos los años

```
x = df_lluvias.index
```

- Para el eje *y*, acumulamos las precipitaciones por año

```
y = df_lluvias.sum(axis='columns')  
y
```



Year	
1965	1263.394
1966	899.588
1967	1522.184
1968	1086.237
1969	1739.022
1970	1273.507
1971	1176.612
1972	718.175



Ejemplo de uso

- Definimos los objetos `fig` y `ax`, los cuales contendrán la figura:

```
fig, ax = plt.subplots(figsize=(12, 4))
ax.plot(x, y, label='Precipitaciones acumuladas')
```

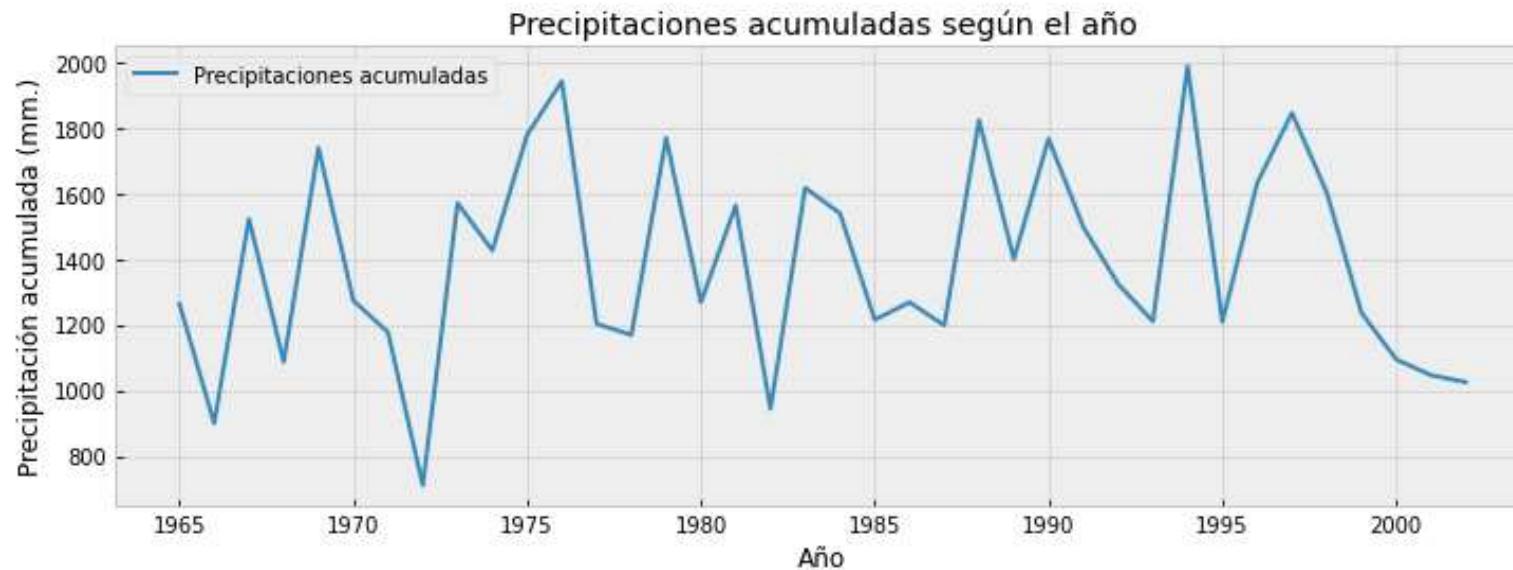
- Etiquetamos los ejes, añadimos un título e insertamos la leyenda:

```
ax.set_xlabel('Año')
ax.set_ylabel('Precipitación acumulada (mm.)')
ax.set_title('Precipitaciones acumuladas según el año')
ax.legend()
```



Ejemplo de uso

Y el resultado es...



Bonita serie de tiempo
¿verdad? 😊



Algunas observaciones

- El método `ax.plot` recibe el parámetro `figsize`, que define el tamaño del gráfico. Para una figura rectangular de 12x4

```
fig, ax = plt.subplots(figsize=(12, 4))
```

- El método `ax.legend` inserta la leyenda en alguna esquina que no tape el gráfico. Esto se puede modificar con el parámetro `loc`.

```
ax.legend(loc='upper right')
```



Forzará a que la leyenda
aparezca en la esquina
superior derecha



Pro-Tip

Si no recordamos qué parámetros acepta un método o función, podemos escribir el nombre del método seguido de un "?". Esto abre la documentación directamente en el entorno de Jupyter.

Por ejemplo, `ax.legend?`
muestra información relacionada al método `legend` 😊

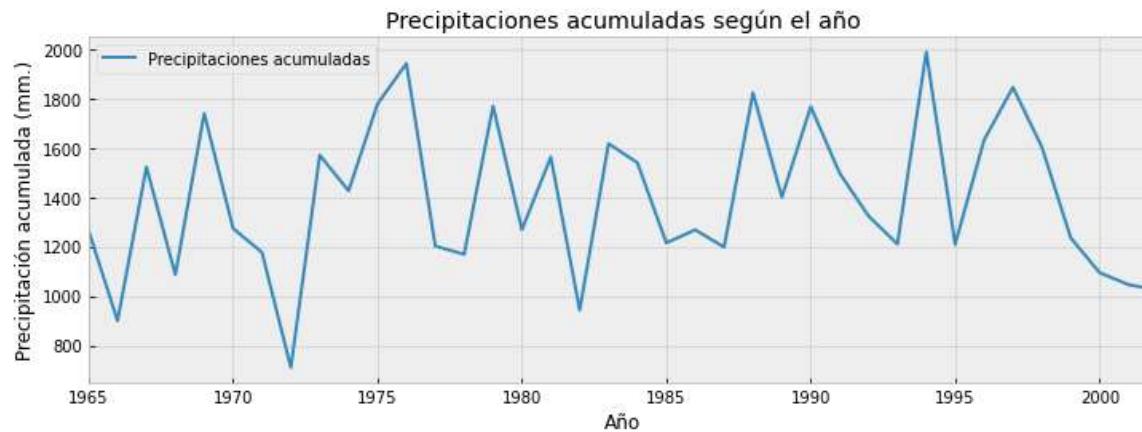


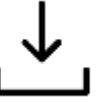
Algunas observaciones

- Si se observa la serie de tiempo anterior, pueden notarse los bordes vacíos en los laterales. Pueden recortarse con `ax.set_xlim`



```
ax.set_xlim(df_lluvias.index[0], df_lluvias.index[-1])
```





Exportando los gráficos

- Matplotlib permite guardar las visualizaciones en la computadora.
- Algunos de los **formatos soportados** son *jpeg*, *jpg*, *png*, *pdf* y *svg*
- El gráfico se guardará en la ruta actual, pero igualmente puede especificar cualquier otra ruta.



```
fig.savefig("precipitaciones_año.pdf")
```

Ejemplo
en vivo



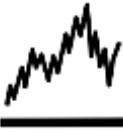
VEAMOS UN EJEMPLO EN VIVO





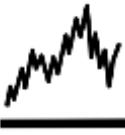
BREAK

Gráficos más comunes



Gráficos de líneas

- Son adecuados para visualizar **datos con secuencialidad temporal**, como las series de tiempo.
- Se grafican con el método `ax.plot(x, y)`
- En caso de no especificarse x, matplotlib toma como coordenadas en x al arreglo de números enteros $[0, 1, 2, \dots, n]$



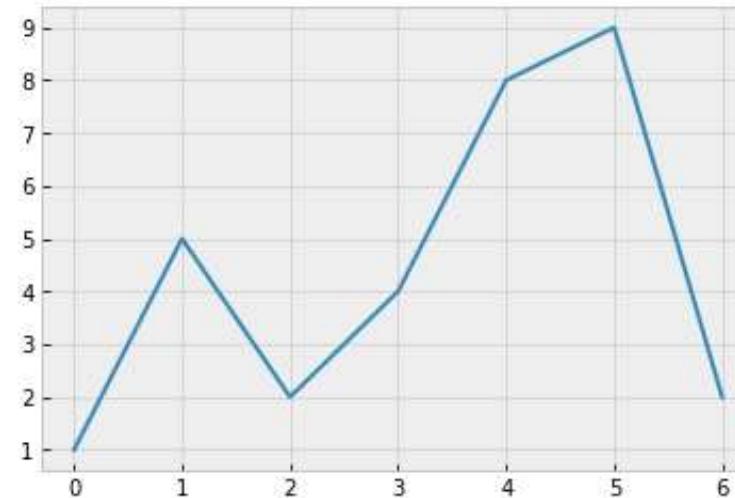
Gráficos de líneas

Veamos un ejemplo...

```
fig, ax = plt.subplots()  
ax.plot([0, 1, 2, 3, 4, 5, 6], [1, 5, 2, 4, 8, 9, 2])
```

equivalentemente....

```
fig, ax = plt.subplots()  
ax.plot([1, 5, 2, 4, 8, 9, 2])
```





Gráficos de puntos

- Útiles cuando se tienen una **gran cantidad de datos numéricos emparejados**
- Permiten visualizar la relación entre las variables por medio de la **nube de puntos**
 - Nube de puntos “alineada”  **relación fuerte**
 - Nube de puntos “dispersa”  **relación débil o nula**
- Se grafican con `ax.scatter`



Gráficos de puntos

Veamos un ejemplo...

Consideremos las mediciones del peso y altura de 50 alumnos

```
pesos = [42.8, 43.3, 42. , 44. , 44.1, 43.5, 48.1, 48.9, 47.7, 46.9, 50.4,  
52.7, 51.8, 54.5, 54.2, 56.9, 55.4, 55.5, 57.1, 58.3, 63.7, 58.8,  
64.6, 60.2, 64. , 63.8, 61.4, 66.3, 64.7, 63.9, 69.3, 67.9, 65.2,  
70.8, 70.5, 69.3, 75.3, 75.5, 78.2, 78. , 73.2, 78. , 80.1, 78.2,  
76. , 81.5, 79.4, 81.8, 81.8, 84.1]
```

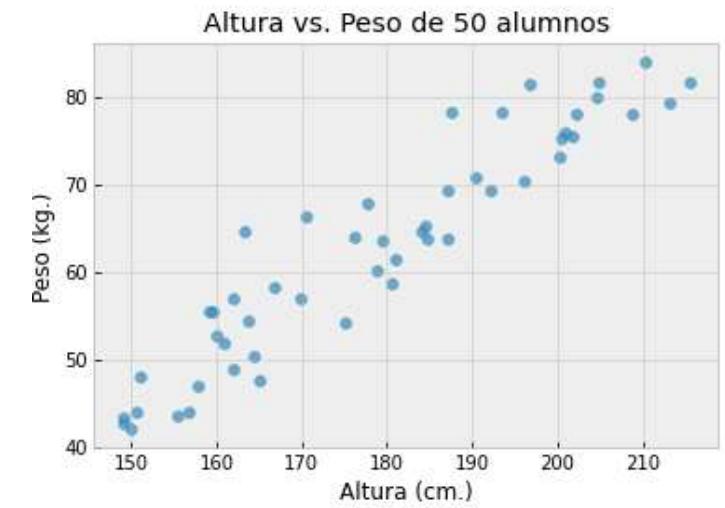
```
alturas = [149. , 149. , 149.9, 156.8, 150.6, 155.4, 151. , 162. , 165.,  
157.8, 164.4, 160.1, 160.8, 163.8, 175.2, 162. , 159.5, 159.2,  
169.8, 166.7, 179.4, 180.6, 163.3, 178.8, 176.3, 184.8, 181. ,  
170.5, 184.1, 187.1, 187.1, 177.7, 184.5, 190.3, 196. , 192.1,  
200.4, 201.8, 187.5, 202.1, 200.3, 208.8, 204.6, 193.5, 200.9,  
196.8, 213.1, 204.8, 215.5, 210.2]
```



Gráficos de puntos

Veamos un ejemplo...

```
fig, ax = plt.subplots()  
ax.scatter(alturas, pesos, alpha=0.7)  
ax.set_title('Altura vs. Peso de 50 alumnos')  
ax.set_xlabel('Altura (cm.)')  
ax.set_ylabel('Peso (kg.)')
```



Como era de esperar, se observa una fuerte relación positiva entre el peso y la altura ?



Gráficos de puntos

Algunas observaciones...

- Para poder visualizar la relación, los valores de los arreglos que se emparejan **deben guardar correspondencia entre sí**. El peso de la primera persona debe estar junto con la altura de la misma persona.
- El parámetro alpha permite cambiar la transparencia de los puntos. Muy útil cuando graficamos muchos puntos.
 - $\alpha = 1$ ➡ puntos sólidos
 - $\alpha = 0.01$ ➡ puntos casi transparentes



Gráficos de puntos

Probemos el Data Frame de precipitaciones

¿Guardarán algún tipo de relación las precipitaciones de Agosto respecto de las de septiembre?

```
fig, ax = plt.subplots()
mapeo_colores = ax.scatter(df_lluvias['Aug'], df_lluvias['Sep'], c=df_lluvias.index)
fig.colorbar(mapeo_colores)
ax.set_title('Precipitaciones Agosto-Septiembre')
ax.set_xlabel('Precipitaciones en Agosto (mm.)')
ax.set_ylabel('Precipitaciones en Septiembre (mm.)')
```



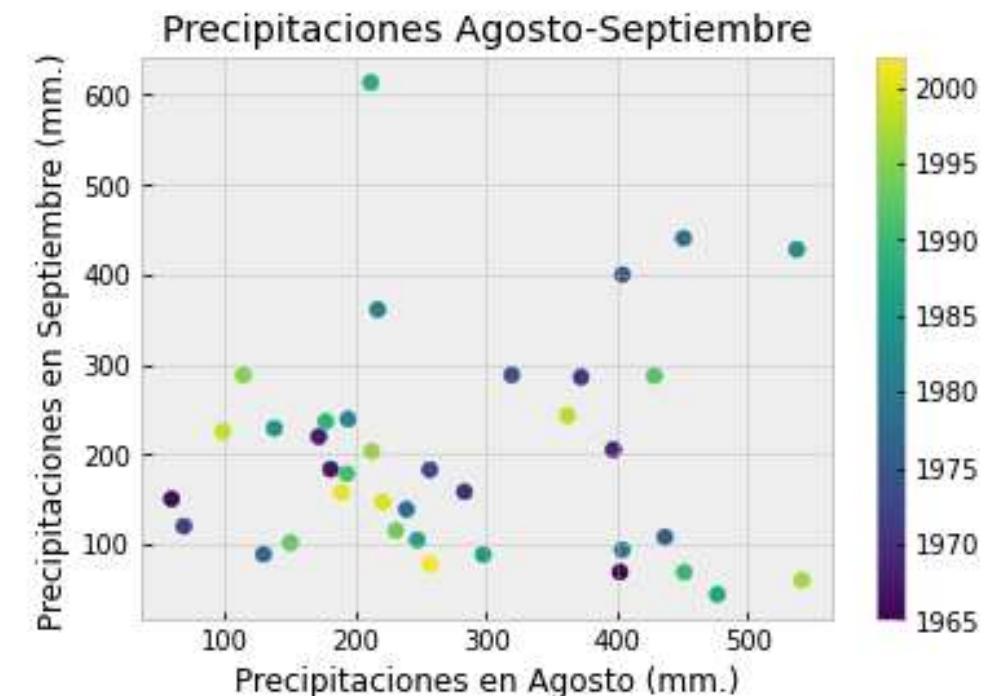
Gráficos de puntos

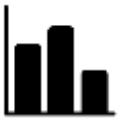
Probemos el Data Frame de precipitaciones

La relación en este caso es **débil**



- Es posible asignar un *rango de colores* a los puntos con el parámetro c y fig.colorbar
- También se puede asignar un *rango de tamaño* con el parámetro s

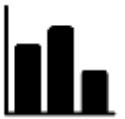




Gráficos de barras

- Permiten comparar y poner en perspectiva los valores de distintas **variables categóricas**. Por ejemplo, las precipitaciones según el mes del año.
- Para el ejemplo, acumulemos las precipitaciones para los distintos meses a lo largo de los años.

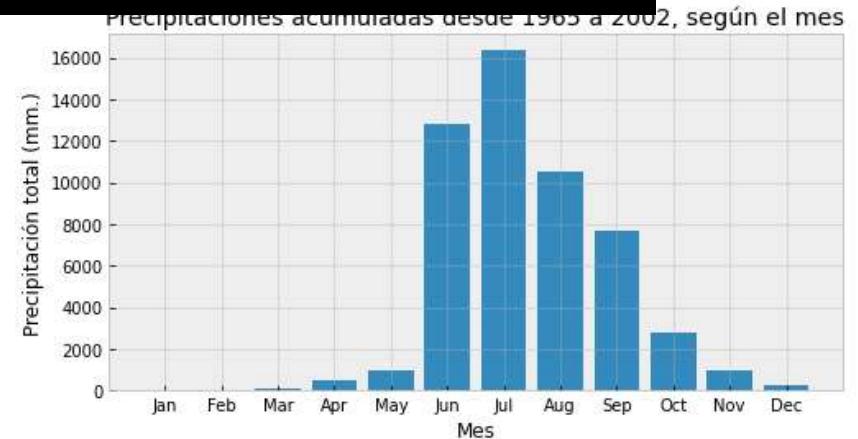
```
precipitaciones_acumuladas = df_lluvias.sum()  
precipitaciones_acumuladas
```

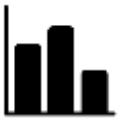


Gráficos de barras

Veamos un ejemplo

```
fig, ax = plt.subplots(figsize=(8,4))
precipitaciones_acumuladas = df_lluvias.sum()
ax.bar(df_lluvias.columns, precipitaciones_acumuladas)
ax.set_title('Precipitaciones acumuladas desde 1965 a 2002, según el mes')
ax.set_ylabel('Precipitación total (mm.)')
ax.set_xlabel('Mes')
```

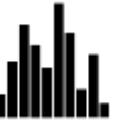




Gráficos de barras

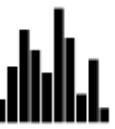
Tener en cuenta

- El eje x representa categorías. La altura de cada barra en el eje y representa la cantidad de elementos para la categoría correspondiente.
- Se grafican con `ax.bar`, que recibe como parámetros:
 - las etiquetas para el eje x
 - la altura de la barra para cada etiqueta



Histograma

- La altura de cada barra representa la **proporción o cantidad** de los distintos valores de una **variable numérica**.
- Requiere **clasificar** a los datos en **intervalos de clase**.
- Permiten comparar la **frecuencia relativa o absoluta** de cada intervalo.
- Se construyen con `ax.hist`, que recibe como parámetro:
 - **El arreglo de valores.**
 - **bins**, que representa la cantidad de intervalos a construir.



Histograma

Veamos un ejemplo...

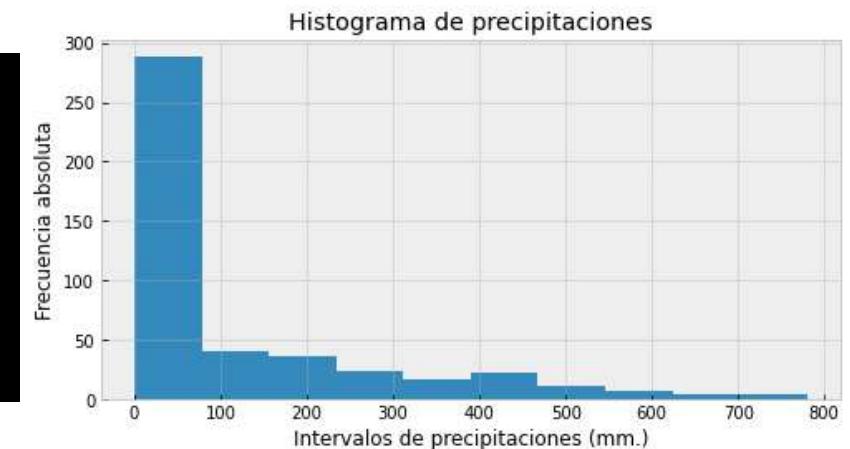
- Aplanemos los valores del Data Frame con el método flatten

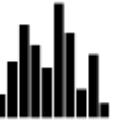
```
df_lluvias.values.flatten()
```



```
array([1.96500e+03, 2.90000e-02, ... ,0.00000e+00])
```

```
fig, ax = plt.subplots(figsize=(8, 4))
ax.hist(df_lluvias.values.flatten(), bins=10)
ax.set_title('Histograma de precipitaciones')
ax.set_xlabel('Intervalos de precipitaciones (mm.)')
ax.set_ylabel('Frecuencia absoluta')
```





Histograma

Algunas observaciones...

- La forma del histograma depende del número de intervalos de clase que pasemos al parámetro *bins*.
- En el ejemplo se representó la **frecuencia absoluta** de los intervalos.
- También se puede representar la **frecuencia relativa de cada intervalo** o el **porcentaje respecto del total**.

Enriqueciendo las visualizaciones



Múltiples elementos

- En ocasiones necesitamos **resaltar** ciertas características de los datos.
- Por ejemplo, ¿Qué pasa si quisiéramos resaltar el punto **máximo** en una serie de tiempo?
- Podemos cargar al *objeto ax* con **múltiples elementos** para que los muestre todos juntos?



Múltiples elementos

Veamos un ejemplo...

1. Comparemos las precipitaciones de Enero y Febrero *en el mismo objeto ax*

```
fig, ax = plt.subplots(figsize=(12, 3))
ax.plot(df_lluvias.index, df['Jan'], label='Precipitaciones de enero')
ax.plot(df_lluvias.index, df['Feb'], label='Precipitaciones de febrero', color='C1')
```

1. Con una agregación, calculamos el máximo de cada uno

```
maximo_enero = df_lluvias['Jan'].max()
maximo_febrero = df_lluvias['Feb'].max()
```



Múltiples elementos

Veamos un ejemplo...

3. El método `axhline` permite graficar *líneas horizontales*. Usemos esto para resaltar los máximos de cada serie de tiempo

```
ax.axhline(maximo_enero, color='red', linestyle='--', alpha=0.5, linewidth=3, label='Máxima de enero')
ax.axhline(maximo_febrero, color='red', linestyle=':', alpha=0.5, linewidth=3, label='Máxima de febrero')
```



También se puede graficar líneas verticales con el método `axvline`

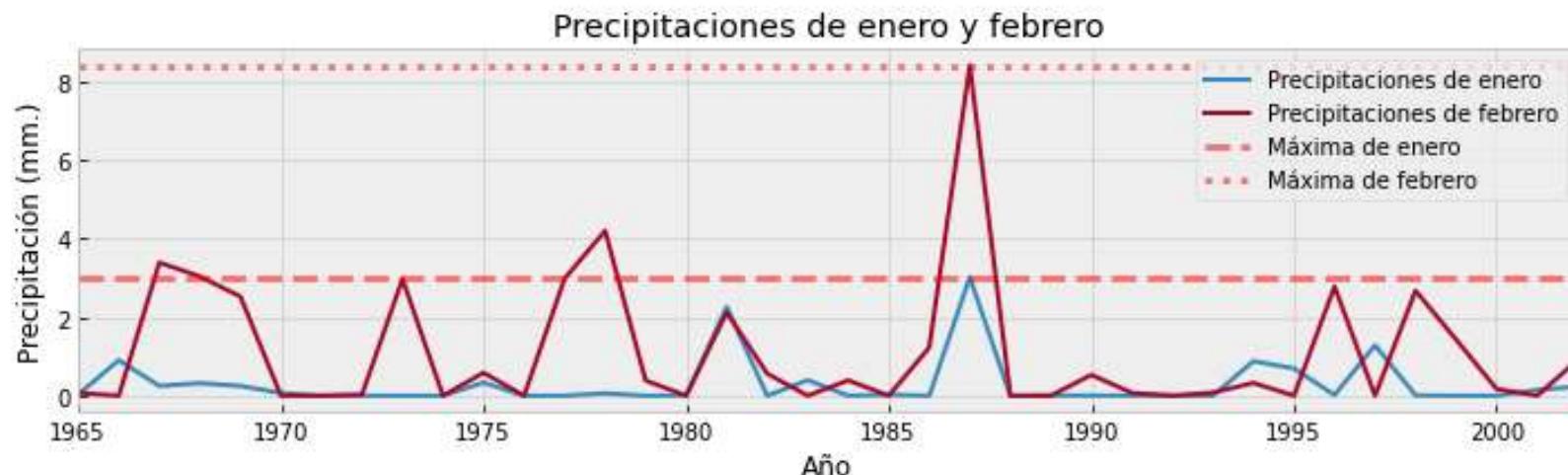


Múltiples elementos

Veamos un ejemplo...

4. Las etiquetas👉

```
ax.set_xlabel('Año')
ax.set_ylabel('Precipitación (mm.)')
ax.set_title('Precipitaciones de enero y febrero')
ax.set_xlim(df_lluvias.index[0], df_lluvias.index[-1])
ax.legend()
```



PRÁCTICA CON ARRAYS EN NUMPY

Formato: Entregar un archivo con formato .ipynb. Debe tener el nombre “Arrays+Apellido”.

Sugerencia: Preparar el código y probar los resultados con distintas entradas.

Desafío
entregable



>> Consigna:

1. Generar un array aleatorio de 100 elementos. Calcular la mediana correspondiente.
2. Recordar los ejercicios con funciones para cálculo de factorial y suma de serie de la Clase 02. Repetir ambos ejercicios, pero ahora utilizar las nuevas operaciones aprendidas con los ndarrays.
3. En [este link](#) se provee un archivo con los resultados de la Encuesta de Sueldos de Openqube de Febrero 2020 (<https://sueldos.openqube.io/encuesta-sueldos-2020.02/>), Calcular y comparar media y mediana de los sueldos netos.

>>Aspectos a incluir en el entregable:

El código debe estar hecho en un notebook y debe estar probado

MAPA DE CONCEPTOS CLASE 5

¡Para recordar!



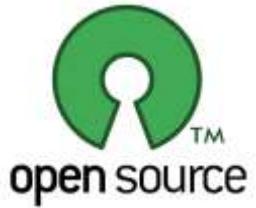
Nociones básicas de Pandas

Data Wrangling y Pandas

"Garbage in, garbage

- A menudo nos encontraremos con **datos caóticos**.
- Al introducirlos en nuestros modelos y algoritmos, si no los preparamos adecuadamente obtendremos **todo tipo de errores y resultados inválidos**.
- Por lo tanto, la **limpieza de los datos** o **Data Wrangling / Data Munging** es un aspecto fundamental de todo proyecto de Data Science.





Pandas al rescate

- Pandas facilita la manipulación de grandes volúmenes de datos a través de un conjunto de métodos y estructuras diseñadas para tal fin.
- Extiende las funcionalidades de Numpy, por lo que sus estructuras de datos son totalmente compatibles.
- Es de uso libre.



Las estructuras de datos en Pandas



Pandas Series

... como los Numpy arrays, pero con índices

- Se construyen a partir de otros objetos particulares, como las listas o los Numpy arrays
- Tienen índice propio.

```
Numeros = range(50, 70, 2)
Numeros_serie = pd.Series(Numeros)
print(Numeros_serie)
```

```
print(Numeros_serie[2])
```

0 50
1 52
2 54
3 56
4 58
5 60
6 62
7 64
8 66
9 68
dtype: int64

54



Pandas Series

- Están formados por dos objetos vinculados: el **arreglo de índices** y el **arreglo de valores**

```
print(numero_serie.index)
```



RangeIndex(start=0, stop=10, step=1)

```
print(numero_serie.values)
```



[50 52 54 56 58 60 62 64 66 68]



Pandas Series

- Tanto el arreglo de índices como el de valores pueden modificarse

```
Numeros_en_texto =  
['primero','segundo','tercero','cuarto','quinto','sexta','séptimo','octavo','noveno','dé  
cimo']  
Numeros_serie_2 = pd.Series(Numeros,index=Numeros_en_texto)  
Numeros_serie_2
```

primero 50
segundo 52
tercero 54
cuarto 56
quinto 58
sexta 60
séptimo 62
octavo 64
noveno 66
décimo 68
dtype: int64



Pandas Data Frames

- Son una **extensión** de los objetos **Series**.
- Pueden pensarse como una **tabla**, formada por varias Series de igual longitud.
- Como toda tabla, consta de filas y columnas.
- Cada fila tiene un elemento índice asociado.

	columna 0	columna 1
0	elemento 0, 0	elemento 0, 1
1	elemento 1, 0	elemento 1, 1
2	elemento 2, 0	elemento 2, 1

Ejemplo ↗



Pandas Data Frames

Construyendo un Data Frame manualmente ▶ □

1. Definir las listas que contienen la información
2. Construir las Series

```
modelos = ['A4 3.0 Quattro 4dr manual',  
          'A4 3.0 Quattro 4dr auto',  
          'A6 3.0 4dr',  
          'A6 3.0 Quattro 4dr',  
          'A4 3.0 convertible 2dr']  
peso = [3583, 3627, 3561, 3880, 3814]  
precios = ['$33,430', '$34,480', '$36,640', '$39,640',  
          '$42,490']
```

```
Autos_peso = pd.Series(peso, index=modelos)  
Autos_precio =  
pd.Series(precios, index=modelos)
```



Pandas Data Frames

Construyendo un Data Frame manualmente ▶ □

Hasta ahora...

```
print(Autos_precio)
```

```
A4 3.0 Quattro 4dr manual      $33,430
A4 3.0 Quattro 4dr auto        $34,480
A6 3.0 4dr                      $36,640
A6 3.0 Quattro 4dr              $39,640
A4 3.0 convertible 2dr          $42,490
dtype: object
```

```
print(Autos_peso)
```

```
A4 3.0 Quattro 4dr manual      3583
A4 3.0 Quattro 4dr auto        3627
A6 3.0 4dr                      3561
A6 3.0 Quattro 4dr              3880
A4 3.0 convertible 2dr          3814
dtype: int64
```



Pandas Data Frames

Construyendo un Data Frame manualmente ▶ □

3. Construir el Data Frame a partir de las Series:

```
Autos =  
pd.DataFrame({'Peso':Autos_peso,'Precio':Autos_precio})  
  
Autos
```



... voilà! 🎉

	Peso	Precio
A4 3.0 Quattro 4dr manual	3583	\$33,430
A4 3.0 Quattro 4dr auto	3627	\$34,480
A6 3.0 4dr	3561	\$36,640
A6 3.0 Quattro 4dr	3880	\$39,640
A4 3.0 convertible 2dr	3814	\$42,490



Pandas Data Frames

Construyamos ahora un tablero de ajedrez

```
Ajedrez_64 = np.arange(1,65).reshape(8,8)
Ajedrez_df = pd.DataFrame(
    Ajedrez_64,
    columns=range(1,9),
    index=['A','B','C','D','E','F','G','H']
)
Ajedrez_df
```

	1	2	3	4	5	6	7	8
A	1	2	3	4	5	6	7	8
B	9	10	11	12	13	14	15	16
C	17	18	19	20	21	22	23	24
D	25	26	27	28	29	30	31	32
E	33	34	35	36	37	38	39	40
F	41	42	43	44	45	46	47	48
G	49	50	51	52	53	54	55	56
H	57	58	59	60	61	62	63	64



Selección de elementos



Seleccionando una Serie

Con Pandas, existen tres formas de seleccionar elementos:

- Mediante el índice



```
Numeros_serie_2['quinto']
```

- Mediante el método *loc()*



```
Numeros_serie_2.loc['quinto']
```

- Mediante el método *iloc()*



```
Numeros_serie_2.iloc[5]
```

que utiliza únicamente índices numéricos

primer	50
segundo	52
tercero	54
cuarto	56
quinto	58
sexto	60
séptimo	62
octavo	64
noveno	66
décimo	68
	dtype: int64



Seleccionando un Data Frame

También podemos seleccionar partes específicas del Data Frame, como índices, columnas y valores

Autos.index

```
Index(['A4 3.0 Quattro 4dr manual', 'A4 3.0 Quattro 4dr auto', 'A6 3.0 4dr',
       'A6 3.0 Quattro 4dr', 'A4 3.0 convertible 2dr'],
      dtype='object')
```

Autos.columns

```
Index(['Peso', 'Precio'], dtype='object')
```

Autos.values

```
array([[3583, '$33,430'],
       [3627, '$34,480'],
       [3561, '$36,640'],
       [3880, '$39,640'],
       [3814, '$42,490]], dtype=object)
```

	Peso	Precio
A4 3.0 Quattro 4dr manual	3583	\$33,430
A4 3.0 Quattro 4dr auto	3627	\$34,480
A6 3.0 4dr	3561	\$36,640
A6 3.0 Quattro 4dr	3880	\$39,640
A4 3.0 convertible 2dr	3814	\$42,490



Seleccionando un Data Frame

- Selección de columna

```
Autos['Peso']
```



```
A4 3.0 Quattro 4dr manual      3583
A4 3.0 Quattro 4dr auto        3627
A6 3.0 4dr                      3561
A6 3.0 Quattro 4dr              3880
A4 3.0 convertible 2dr          3814
Name: Peso, dtype: int64
```

- Selección de fila

```
Autos.values[1]
```



```
array([3561, '$36,640'], dtype=object)
```

```
Autos.loc['A4 3.0 Quattro 4dr auto', ]
```



```
Peso           3627
Precio        $34,480
Name: A4 3.0 Quattro 4dr auto, dtype: object
```



Seleccionando un Data Frame

- Selección con condición

Supongamos que necesitamos un listado de precios de aquellos autos con peso mayor a 3600 ↗

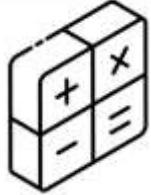
```
Autos.loc[Autos.Peso >= 3600,'Precio']
```



```
A4 3.0 Quattro 4dr auto      $34,480
A6 3.0 Quattro 4dr          $39,640
A4 3.0 convertible 2dr      $42,490
Name: Precio, dtype: object
```

Operaciones en Pandas

Operaciones básicas con datos en Pandas



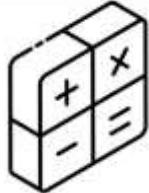
Transposición

- Consiste en **intercambiar** filas de un Data Frame por sus columnas.
- Puede resultar más cómodo trabajar con el Data Frame transpuesto que con el original.

Autos.T

	A4 3.0 Quattro 4dr manual	A4 3.0 Quattro 4dr auto	A6 3.0 4dr	A6 3.0 Quattro 4dr	A4 3.0 convertible 2dr
Peso	3583	3627	3561	3880	3814
Precio	\$33,430	\$34,480	\$36,640	\$39,640	\$42,490

En ocasiones necesitamos ver las cosas desde otra perspectiva... 😊



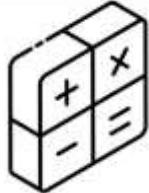
Funciones Vectorizadas

Desde Numpy, en Pandas

- Las *funciones vectorizadas* o *ufuncs* de Numpy pueden realizarse también sobre Data Frames y Series.
- Tras ejecutar la operación **se conservan los índices**.

```
Numeros_3 = range(51,70,2)
Numeros_serie_3 = pd.Series(Numeros_3,index=Numeros_en_texto)
Numeros_serie_3
```

primero	51
segundo	53
tercero	55
cuarto	57
quinto	59
sexto	61
séptimo	63
octavo	65
noveno	67
décimo	69
dtype:	int64



Ufuncs sobre Data Frames

Los Data Frames también admiten operaciones vectorizadas

Calculemos el porcentaje de un Data Frame con respecto a los valores de su primera fila

```
largo = [179, 179, 192, 192, 180]
```

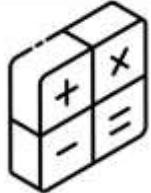
```
Autos_2 = pd.DataFrame({'Peso':peso,'Largo':largo},index=modelos)
```

```
Autos_2
```

```
Autos_2 / Autos_2.iloc[0] * 100
```



	Peso	Largo	Peso	Largo
A4 3.0 Quattro 4dr manual	100.000000	100.000000	3583	179
A4 3.0 Quattro 4dr auto	101.228021	100.000000	3627	179
A6 3.0 4dr	99.385989	107.262570	3561	192
A6 3.0 Quattro 4dr	108.289143	107.262570	3880	192
A4 3.0 convertible 2dr	106.447111	100.558659	3814	180



Conservación de índices

Veamos qué sucede con los índices al sumar series

Numeros_serie_2

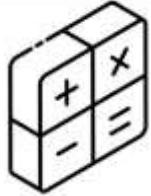


Numeros_serie_2 + Numeros_serie_3

```
primer 51
segundo 53
tercero 55
cuarto 57
quinto 59
sexto 61
séptimo 63
octavo 65
noveno 67
décimo 69
dtype: int64
```

primer	101
segundo	105
tercero	109
cuarto	113
quinto	117
sexto	121
séptimo	125
octavo	129
noveno	133
décimo	137

dtype: int64



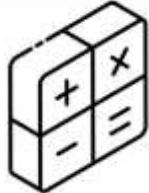
Conservación de Índices

- Otra forma de sumar Series o DataFrames es con el método `.add()`

```
Numeros_serie_2.add(Numeros_serie_3)
```



primer	101
segundo	105
tercero	109
cuarto	113
quinto	117
sexto	121
séptimo	125
octavo	129
noveno	133
décimo	137
	dtype: int64



Conservación de índices

¿Y si sumamos series con índices incompatibles?



```
Numeros_serie_2_porcion = Numeros_serie_2[4:7]  
Numeros_serie_3_porcion = Numeros_serie_3[5:8]  
print(Numeros_serie_3_porcion, Nnumeros_serie_2_porcion)
```

```
sexto      61  quinto      58  
séptimo    63  sexto       60  
octavo     65  séptimo    62  
dtype: int64  dtype: int64
```

```
print(Numeros_serie_2_porcion + Numeros_serie_3_porcion)
```



```
octavo      NaN  
quinto      NaN  
sexta      121.0  
séptimo    125.0  
dtype: float64
```

¡Los índices que no coinciden se rellenan con **NaN**!



Houston... Tenemos un problema

¿Qué hacemos con los valores faltantes?





Lidiando con valores faltantes

- La mayoría de las operaciones de Pandas admiten un parámetro `fill_value`, que indica el valor a insertar en caso de resultar un `NaN`.

```
Numeros_serie_2_porcion.add(Numeros_serie_3_porcion, fill_value=0)
```

*En este caso, especificamos
que caso de encontrar un valor
faltante lo reemplace por 0*

octavo 65.0 ↗
quinto 58.0
sexto 121.0
séptimo 125.0
dtype: float64



¿*Datos ausentes, por qué?* 😕

- Como futuros Data Scientists, comúnmente nos toparemos con valores faltantes o **ausentes** que podrían provenir de las siguientes situaciones:
 - Fallas** en algún paso de la carga de datos.
 - Omisión** directa de la carga de datos.
 - Reticencia** de parte de un encuestado a dar una respuesta determinada.

¡Los valores faltantes son más comunes de lo que piensa!



Not a Number

El representante del valor faltante

- NaN significa **Not a Number** y es el **valor faltante por defecto**.
- Es un tipo de dato especial de punto flotante.
- Tiene **propiedades especiales**: *cualquier operación que involucre NaN da como resultado NaN.*



Propagación de valores faltantes

- Probemos esta propiedad utilizando el objeto NaN de Numpy:
Veamos qué sucede al operar con NaN

```
valor_nan = np.nan  
type(valor_nan)
```

☞ float

```
2 * valor_nan
```

☞ nan

- Algunas funciones están preparadas para trabajar con NaN:

```
np.nanprod([2,valor_nan])
```

☞ 2.0

En este caso, Numpy le asignó un valor de 1 y realizó la multiplicación normalmente



Trabajando con datos

ausentes

- Estos valores **podrían no ser adecuados** para algunos algoritmos de Data Science. Por ello, deben ser manejados correctamente.
- Pandas nos provee de herramientas para trabajar con ellos.

Primero que nada, definamos nuestro conjunto de prueba:

```
Numeros_nan = Numeros_serie_2_porcion + Numeros_serie_3_porcion  
Numeros_nan
```

octavo NaN
quinto NaN
sexto 121.0
séptimo 125.0
dtype: float64



Trabajando con datos

ausentes

- Podemos marcarlos

Veamos algunos ejemplos.

```
Numeros_nan.isnull()
```



octavo	True
quinto	True
sexto	False
séptimo	False
dtype: bool	

- Podemos reemplazarlos

```
Numeros_nan.fillna(0)
```



octavo	0.0
quinto	0.0
sexto	121.0
séptimo	125.0
dtype: float64	

- Podemos eliminarlos

```
Numeros_nan.dropna()
```



sexto	121.0
séptimo	125.0
dtype: float64	

MAPA DE CONCEPTOS CLASE 6

¡Para recordar!



CRONOGRAMA DEL CURSO

Clase 5



Introducción a la manipulación de datos con Pandas (Parte I)



NOCIONES BÁSICAS DE PANDAS



OPERACIONES EN PANDAS

Clase 6



Introducción a la manipulación de datos con Pandas (Parte II)



¡VAMOS AL CÓDIGO



PRÁCTICA INTEGRADORA:
PANDAS Y SERIES DE TIEMPO

Clase 7



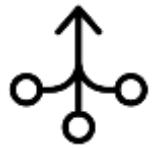
Visualizaciones en Python (Parte I)



¡VAMOS AL CÓDIGO

Operaciones en Pandas

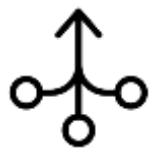
Agregaciones con Pandas



Recall: Agregaciones

De la clase anterior recordemos que ...

- Las agregaciones son un **tipo de operación**.
- Se realizan **sobre un conjunto** de datos.
- Retornan un resultado que es una **medida resumen** del conjunto de datos
- Las principales agregaciones de Numpy son:
`np.sum, np.mean, np.max, np.std, np.var`



Agregaciones en Pandas

- Pandas permite realizar agregaciones sobre Data Frames enteros o porciones del mismo.

En primer lugar, importemos nuestro *dataset de prueba*:

1. Descargue el archivo con formato .csv desde [este enlace](#).
2. Copie la ruta del archivo y pásela al método *read_csv* de Pandas:

```
df_lluvias_archivo = pd.read_csv('<ruta>/pune_1965_to_2002.csv')
```



Lectura de Datasets

Hasta el momento, nuestro dataset luce así 

- Se trata de **mediciones de precipitaciones** (en milímetros)
- Existe un total de trece columnas, una para el año y otras doce para cada uno de los meses
- Tiene un índice numérico

	Year	Jan	Feb	Mar	Apr	May
0	1965	0.029	0.069	0.000	21.667	17.859
1	1966	0.905	0.000	0.000	2.981	63.008
2	1967	0.248	3.390	1.320	13.482	11.116
3	1968	0.318	3.035	1.704	23.307	7.441
4	1969	0.248	2.524	0.334	4.569	6.213

Sería conveniente que el índice sea la columna Year
Construyamos un nuevo Data Frame con este índice



Lectura de Datasets

```
indice = list(df_lluvias_archivo.Year)  
indice
```

↳ [1965, 1966, 1967, ..., 2002]

```
columnas = df_lluvias_archivo.columns[1:]  
columnas
```

↳ Index(['Jan', 'Feb', ..., 'Dec'], dtype='object')

Guardamos en un arreglo
todos los valores, excepto los
de la primera columna



```
valores = df_lluvias_archivo.values[:,1:]
```



Lectura de Datasets

Ensamblamos las partes... ? ?

```
df_lluvias = pd.DataFrame(valores,index=indice,columns=columnas)  
df_lluvias
```

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
1965	0.029	0.069	0.000	21.667	17.859	102.111	606.071	402.521	69.511	5.249	16.232	22.075
1966	0.905	0.000	0.000	2.981	63.008	94.088	481.942	59.386	150.624	1.308	41.214	4.132
1967	0.248	3.390	1.320	13.482	11.116	251.314	780.006	181.069	183.757	50.404	8.393	37.685
1968	0.318	3.035	1.704	23.307	7.441	179.872	379.354	171.979	219.884	73.997	23.326	2.020
1969	0.248	2.524	0.334	4.569	6.213	393.682	678.354	397.335	205.413	24.014	24.385	1.951

¡Y listo!

Ya tenemos
preparado
nuestro Data
Frame



Agregaciones en Pandas

- Suma de las precipitaciones para cada mes.
- Promedio de precipitaciones de cada año.

```
df_lluvias.mean(axis='columns')
```

			Jan	11.186
			Feb	41.843
			Mar	63.733
			Apr	470.487
			May	952.272
			Jun	12809.663
			Jul	16340.395
			Aug	10529.357
			Sep	7642.245
			Oct	2783.320
			Nov	958.492
			Dec	230.646
				dtype: float64
	1965	105.282833		
	1966	74.965667		
↳	1967	126.848667		
	1968	90.519750		
	...			
	2002	85.406750		
				dtype: float64



El método describe

- Podemos obtener un breve resumen del Data Frame con `describe()`
- Podemos redondear los valores de un Data Frame con el método `round()`



```
df_lluvias.describe().round(1)
```

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
count	38.0	38.0	38.0	38.0	38.0	38.0	38.0	38.0	38.0	38.0	38.0	38.0
mean	0.3	1.1	1.7	12.4	25.1	337.1	430.0	277.1	201.1	73.2	25.2	6.1
std	0.6	1.7	2.5	13.7	22.5	171.7	178.0	132.2	123.7	62.9	31.8	11.7
min	0.0	0.0	0.0	0.1	0.5	94.1	84.9	59.4	44.6	1.1	0.3	0.0
25%	0.0	0.0	0.0	2.3	7.0	226.2	322.5	183.2	105.9	22.0	3.7	0.0
50%	0.0	0.2	0.6	5.5	18.1	312.1	415.1	243.2	181.0	49.8	14.7	0.5
75%	0.2	1.9	2.1	19.8	33.1	412.6	555.3	401.2	242.4	115.7	37.0	4.2
max	3.0	8.4	9.6	53.3	80.5	773.7	780.0	541.6	613.5	225.9	122.8	37.7



El método describe

- Si **transponemos** el Data Frame antes de aplicar `describe`, obtenemos el resumen según el año



```
df_lluvias.T.describe().round(1)
```

	1965	1966	1967	1968	1969	1970	1971	1972	1973	1974	...
count	12.0	12.0	12.0	12.0	12.0	12.0	12.0	12.0	12.0	12.0	...
mean	105.3	75.0	126.8	90.5	144.9	106.1	98.1	59.2	130.8	118.8	...
std	193.8	136.7	223.5	121.5	226.5	130.8	139.3	100.1	209.3	148.0	...
min	0.0	0.0	0.2	0.3	0.2	0.0	0.0	0.0	0.0	0.0	...
25%	4.0	1.2	7.1	2.8	2.4	0.1	0.0	0.1	2.3	0.2	...
50%	19.8	22.7	25.6	23.3	15.1	42.1	20.1	12.5	10.2	34.0	...
75%	77.7	70.8	181.7	174.0	252.5	189.8	163.2	81.7	201.6	213.8	...
max	606.1	481.9	780.0	379.4	678.4	330.5	372.7	338.5	696.0	405.4	...

Operaciones sobre Strings



Operaciones sobre Strings

- A menudo, tendremos que trabajar con **datos en forma de Strings**, es decir cadenas de caracteres o texto.
- Es muy probable que no tengan el **formato** requerido
- Pandas provee métodos para **manipular Strings masivamente** 😊



Operaciones sobre Strings

Para estos ejemplos, usaremos el *dataset de presidentes de EEUU*:

1) Descargue el archivo .csv en [este enlace](#).

```
Presidentes_archivo = pd.read_csv('<ruta>/us_presidents.csv')
```

2) Seleccione la columna *president*

```
Presidentes_nombres =
pd.Series(Presidentes_archivo['president'])
Presidentes_nombres
```

George Washington
John Adams
Thomas Jefferson
...
Donald Trump
Name: president, dtype: object





Operaciones sobre Strings

Veamos algunos ejemplos...

- Convertir a mayúsculas

```
Presidentes_nombres.str.upper()  
)
```



```
0      GEORGE WASHINGTON  
1          JOHN ADAMS  
...  
44          DONALD TRUMP  
Name: president, dtype: object
```

- Longitud total, incluyendo espacios y otros caracteres que puedan aparecer

```
Presidentes_nombres.str.len()
```



```
0      17  
1      10  
...  
44     12  
Name: president, dtype: int64
```

Operaciones sobre Strings

Veamos algunos ejemplos...

- Evaluar si comienzan con una determinada letra

```
Presidentes_nombres.str.startswith(['J'])
```

```
0      False
1      True
...
44     False
Name: president, dtype: bool
```

- Separar en una lista
usando el espacio como separador

```
Presidentes_nombres.str.split()
```

```
0      [George, Washington]
1      [John, Adams]
...
44     [Donald, Trump]
Name: president, dtype: object
```

Introducción a Series de Tiempo



Series de Tiempo

... datos, ligados al tiempo ⏲

- Son **tipos de datos especiales** donde el tiempo toma un rol fundamental.
- Observamos **cambios en los valores** de la variable a lo largo del tiempo.
- Si ignoramos esa **dimensión temporal**, los valores *pierden contexto*.



Series de Tiempo

... datos, ligados al tiempo ⏲

- Python provee tres tipos de datos relacionados al tiempo:
 - **Time stamp o marca de tiempo:** representan un punto en el tiempo. Por ejemplo, fecha y hora.
 - **Período:** representan un intervalo de tiempo. Por ejemplo, los minutos transcurridos desde que comenzó la clase hasta ahora.
 - **Duración:** representa una duración medida en tiempo, pero independientemente del momento en que sucede. Por ejemplo, 15 minutos.



Series de Tiempo

... datos, ligados al tiempo ⏲

- Por su parte, Pandas provee un **objeto índice** para cada uno de esos objetos temporales:

Tipo de dato	Objeto en Python	Índice en Pandas
Time stamp	Timestamp	DatetimeIndex
Período	Period	PeriodIndex
Duración	Timedelta	TimedeltaIndex



Operando objetos de tiempo

Veamos algunos ejemplos...

- Convertir String a Timestamp:

```
fecha = pd.to_datetime('03/01/2020',dayfirst=True)  
fecha
```

Timestamp('2020-01-03 00:00:00')

- Días desde el 3 de enero al del 2020 al 10 de enero del 2020:

```
fin = pd.to_datetime('10/01/2020',dayfirst=True)  
fechas_1 = pd.date_range(start=fecha, end=fin)
```

```
DatetimeIndex(['2020-01-03', '2020-01-04', '2020-01-05', '2020-01-06',  
                '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10'],  
               dtype='datetime64[ns]', freq='D')
```



Operando objetos de tiempo

Veamos algunos ejemplos...

- Ocho fechas desde el 3 de enero de 2020, con períodos:

```
fechas_2 = pd.date_range(start=fecha, periods=8)  
fechas_2
```

☞ DatetimeIndex(['2020-01-03', '2020-01-04', '2020-01-05', '2020-01-06',
'2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10'],
dtype='datetime64[ns]', freq='D')

- La **frecuencia** por defecto es de un día. Por lo tanto, ocho períodos representan ocho días.



Operando objetos de tiempo

Veamos algunos ejemplos...

- Cambiando la frecuencia a meses en lugar de días:

```
fechas_3 = pd.date_range(start= fecha, periods= 8, freq='M')  
fechas_3
```



```
DatetimeIndex(['2020-01-31', '2020-02-29', '2020-03-31', '2020-04-30',  
                '2020-05-31', '2020-06-30', '2020-07-31', '2020-08-31'],  
               dtype='datetime64[ns]', freq='M')
```

- Notar que como día se toma el último de cada período



Operando objetos de tiempo

Veamos algunos ejemplos...

- Ocho meses consecutivos, a partir del mes de inicio:

```
mes_inicio = fecha.strftime('%Y-%m')
mes_inicio
```

2020-01

```
fechas_4 = pd.period_range(start=mes_inicio, periods=8, freq='M')
fechas_4
```

PeriodIndex(['2020-01', '2020-02', '2020-03', '2020-04', '2020-05', '2020-06',
 '2020-07', '2020-08'],
 dtype='period[M]', freq='M')



Operando objetos de tiempo

Veamos algunos ejemplos...

- ¿Cuánto tiempo pasó desde el primer periodo al último?

```
cuanto_tiempo = fechas_3[7] - fechas_3[0]  
cuanto_tiempo
```

↳ `Timedelta('213 days 00:00:00')`

¡Al utilizar operadores normales sobre objetos de tiempo, obtenemos como resultado objetos de tiempo! 😊



Operando objetos de tiempo

Veamos algunos ejemplos...

- ¿Cuántos meses pasaron desde el primer periodo al último?

```
cuanto_tiempo_meses = fechas_3[7].to_period('M') - fechas_3[0].to_period('M')
cuanto_tiempo_meses
```



<7 * MonthEnds>



Conversión a *DatetimeIndex*

Veamos algunos ejemplos...

Ahora que sabemos manipular objetos de tiempo, retomemos el Data Frame de presidentes. Seleccionamos las fechas de asunción

```
fechas_presidentes_orig = Presidentes_archivo['start']
fechas_presidentes_orig
```

0 April 30, 1789
1 March 4, 1797
2 March 4, 1801
...
44 January 20, 2017
Name: start, dtype: object



```
type(fechas_presidentes_orig)
```

↳ pandas.core.series.Series X



Conversión a *DatetimeIndex*

Veamos algunos ejemplos...

Transformemos las fechas en formato string a **índices de tiempo**

```
fechas_presidentes = pd.DatetimeIndex(fechas_presidentes_orig)  
fechas_presidentes
```

⌚ DatetimeIndex(['1789-04-30', '1797-03-04', ..., '2017-01-20'],
dtype='datetime64[ns]', name='start', freq=None) ✓ □



Conversión a *DatetimeIndex*

Veamos algunos ejemplos...

Ahora que tenemos las fechas en el tipo de dato correcto,
construyamos la Serie

```
Serie_presidentes = pd.Series(Presidentes_nombres.values, index=fechas_presidentes)  
Serie_presidentes
```

```
start  
1789-04-30      George Washington  
1797-03-04      John Adams  
1801-03-04      Thomas Jefferson  
...  
2017-01-20      Donald Trump  
dtype: object
```

¡Listo! 🎉

*Ya podemos ejecutar
operaciones con
objetos de tiempo* 😊



PRÁCTICA INTEGRADORA: PANDAS Y SERIES DE TIEMPO

Trabajaremos sobre un conjunto de datos aplicando los conceptos aprendidos

PRÁCTICA INTEGRADORA: PANDAS Y SERIES DE TIEMPO

Formato: Entregar un archivo con formato .ipynb. Debe tener el nombre “Pandas+Apellido”.

Sugerencia: Preparar el código y probar los resultados con subconjuntos del conjunto original.

Desafío
entregable



>> Consigna:

1. Con [este archivo](#), construir un objeto de series de tiempo con el índice igual al año de los juegos olímpicos
2. Separar nombres de apellidos en dos columnas distintas usando la coma como separador
3. Obtener medidas resumen del conjunto de datos. Cuál es el país que ganó más medallas?
4. Construir una tabla que muestre cuántas medallas obtuvieron los hombres en total en cada año que se realizó el evento.

>>Aspectos a incluir en el entregable:

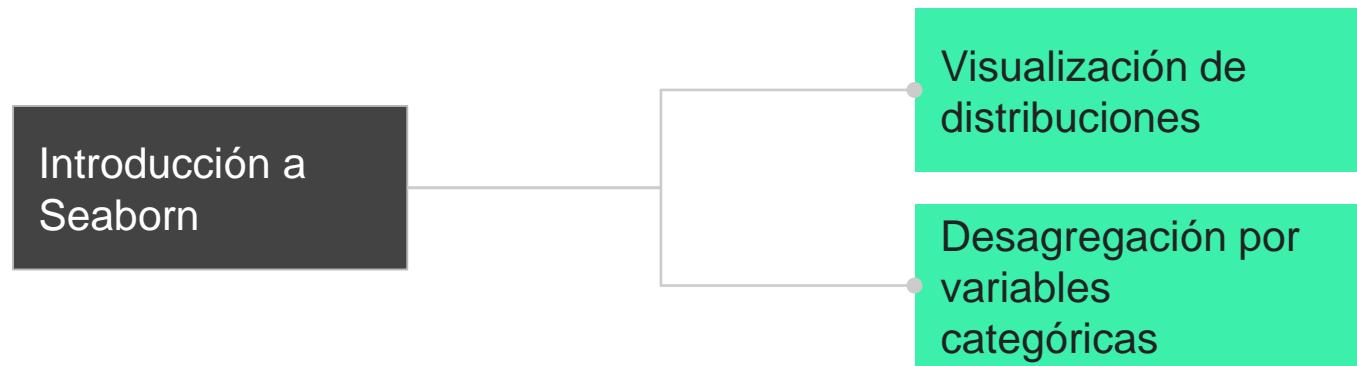
El código debe estar hecho en un notebook y debe estar probado.



OPINA Y VALORA
ESTA CLASE

MAPA DE CONCEPTOS CLASE 8

¡Para recordar!



CRONOGRAMA DEL CURSO

Clase 7



Visualizaciones en
Python (Parte I)



¡VAMOS AL CÓDIGO

Clase 8



Visualizaciones en
Python (Parte II)



PRÁCTICA INTEGRADORA:
VISUALIZACIÓN EN PYTHON

Clase 8



Estadística descriptiva

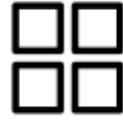
¿Repasamos?



En la primera parte de Visualización en Python vimos los siguientes temas:

- La librería Matplotlib 
- Gráficos más comunes 
 - Gráficos de línea.
 - Gráficos de puntos.
 - Gráficos de barra.
 - Histograma.

Subgráficos



Subgráficos

- Podemos definir una **grilla de gráficos** dentro de una misma figura.
- En plt.subplots, especificamos:
 - El *número de filas* de la grilla → nrows
 - El *número de columnas* de la grilla → ncols
- El *objeto ax se convierte en un array*. Por lo tanto, **debemos usar corchetes** ⚠



Subgráficos

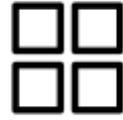
Veamos un ejemplo...

Comparemos las precipitaciones de Enero, Febrero y Marzo a lo largo de los años. ¿Cuál será el más seco? □

1. Definimos un *objeto ax* con tres filas y una sola columna
 - En el eje x los años
 - En el eje y las precipitaciones

```
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(12, 5), sharex=True, sharey=True)
```

Como los años son los mismos para todos los gráficos, ponemos el parámetro sharex en True



Subgráficos

Veamos un ejemplo...

2. A cada fila, le asignamos las precipitaciones de un mes

```
ax[0].plot(df_lluvias.index, df_lluvias['Jan'], label='Precipitaciones de enero')
ax[1].plot(df_lluvias.index, df_lluvias['Feb'], label='Precipitaciones de febrero', color='C1')
ax[2].plot(df_lluvias.index, df_lluvias['Mar'], label='Precipitaciones de marzo', color='C2')
```



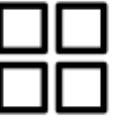
Subgráficos

Veamos un ejemplo...

3. Por último, añadimos texto y leyendas

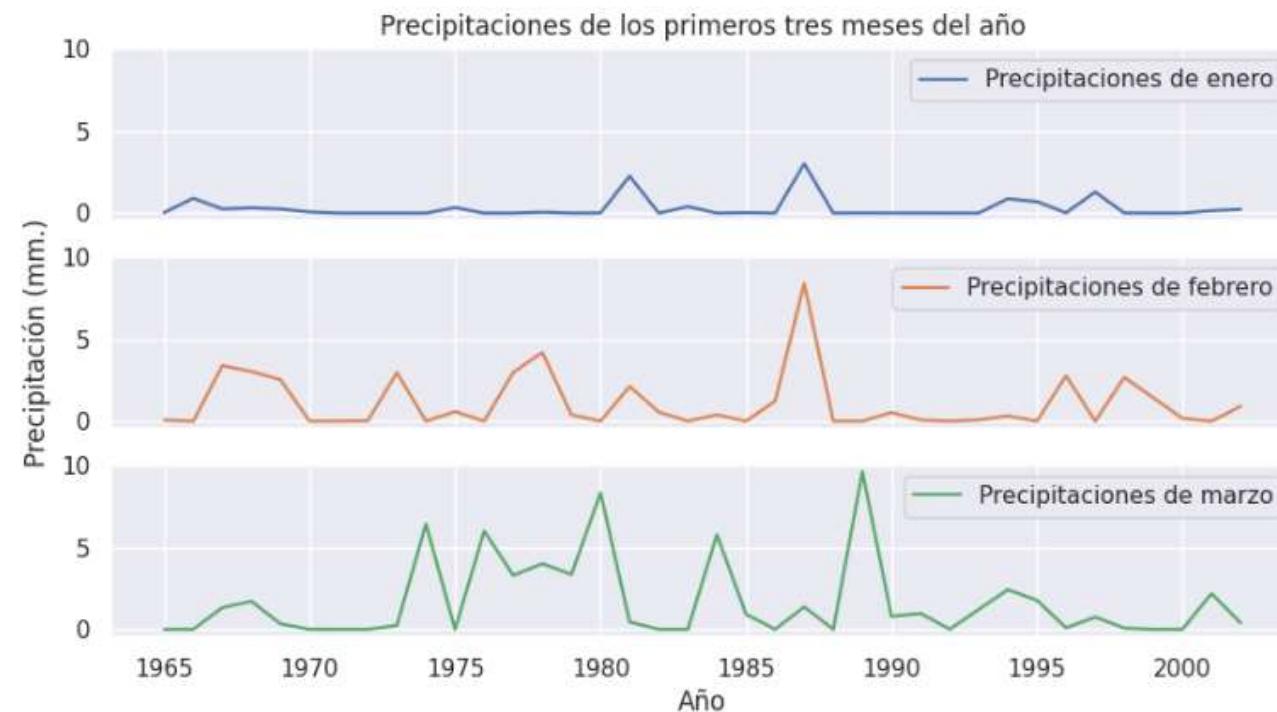
```
ax[0].set_title('Precipitaciones de los primeros tres meses del año')
ax[2].set_xlabel('Año')
ax[1].set_ylabel('Precipitación (mm.)')

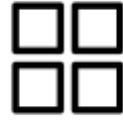
ax[0].legend()
ax[1].legend()
ax[2].legend()
```



Subgráficos

El resultado 😊





Subgráficos

Algunas observaciones...

- Al pasar `sharey=True`, los subgráficos comparten la escala en el eje y. Esto permitió comparar a simple vista el volumen de precipitaciones.
- Cada subgráfico puede tener su propio título y etiquetas.
- En caso de tener dos filas y dos columnas, `ax` se torna *bidimensional*:
 - El subgráfico superior izquierdo se referencia con `ax[0,0]`
 - El subgráfico superior derecho se referencia con `ax[0,1]`

Personalizando Matplotlib



Personalizando Matplotlib

- Matplotlib permite modificar cada aspecto de sus gráficos por medio de **parámetros**.
- Al importar la librería, Matplotlib establece sus parámetros **por defecto**.
- Los parámetros se guardan en una estructura de datos de **tipo dict**. Se puede obtener una lista de los parámetros consultando sus claves.

```
mpl.rcParams.keys()
```



Personalizando Matplotlib

Algunos parámetros comunes... □

Parámetro	Descripción	Valor por defecto
axes.grid	Mostrar grilla	True
axes.titleweight	Grosor tipografía título	"normal"
axes.titlelocation	Posición del título	"center"



Personalizando Matplotlib

Algunos parámetros comunes... □

Parámetro	Descripción	Valor por defecto
axes.grid.axis	Ejes de la grilla	"both"
axes.labelcolor	Color de etiquetas	"black"
axes.labelsize	Tamaño de fuente de las etiquetas	"large"



Personalizando Matplotlib

Algunos parámetros comunes... □

Parámetro	Descripción	Valor por defecto
axes.labelweight	Grosor de fuente de las etiquetas	"normal"
grid.alpha	Transparencia de la grilla	1.0
grid.color	Color de la grilla	"#b2b2b2"



Personalizando Matplotlib

Algunos parámetros comunes... □

Parámetro	Descripción	Valor por defecto
grid.linestyle	Estilo de grilla	"--"
grid.linewidth	Grosor de la grilla	0.5
legend.fontsize	Tamaño de fuente de la leyenda	"medium"



Personalizando Matplotlib

Veamos un ejemplo... □

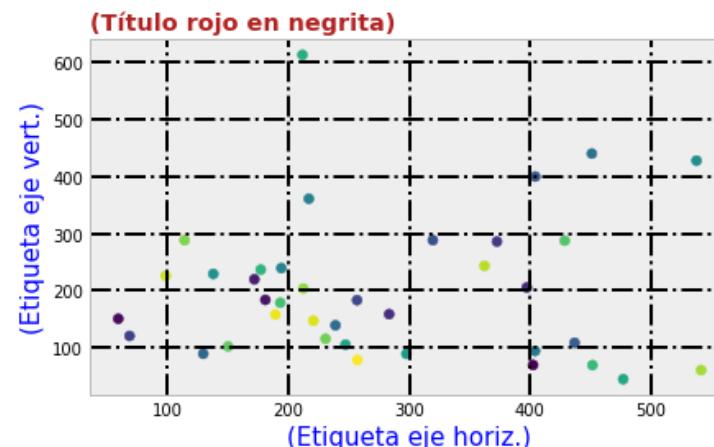
```
mpl.rcParams['axes.titleweight'] = 'bold'  
mpl.rcParams['axes.titlelocation'] = 'left'  
mpl.rcParams['axes.titlecolor'] = 'firebrick'  
mpl.rcParams['axes.labelcolor'] = 'blue'  
mpl.rcParams['axes.labelsize'] = '10'  
mpl.rcParams['axes.labelweight'] = 'light'  
mpl.rcParams['axes.linewidth'] = '1'  
mpl.rcParams['grid.color'] = 'black'  
mpl.rcParams['grid.linestyle'] = '-.'  
mpl.rcParams['grid.linewidth'] = '2'
```



Personalizando Matplotlib

Veamos un ejemplo... □

```
fig, ax = plt.subplots(figsize=(7, 4))
ax.scatter(df_lluvias['Aug'], df_lluvias['Sep'], c=df_lluvias.index)
ax.set_title('Título rojo en negrita')
ax.set_xlabel('Etiqueta eje horiz.')
ax.set_ylabel('Etiqueta eje vert.)')
```





Personalizando Matplotlib

Algunas observaciones...

- Otra forma de modificar los parámetros es mediante plt.rc

```
plt.rc('axes', titlelocation='left', titlecolor='firebrick', ...)  
plt.rc('grid', color='black', linestyle='-.', ...)
```

- Es posible restablecer los parámetros por defecto



```
mpl.rcParams.update(mpl.rcParamsDefault)
```

La librería Seaborn

Seaborn

Extendiendo las capacidades de matplotlib 🚀

- Funciona por encima de matplotlib.
- Se integra muy bien con las estructuras de datos de Pandas.
- Provee métodos que facilitan la generación de gráficos para la comparación de variables categóricas.
- Provee sus propios estilos y colores (*muy estéticos, por cierto 😊*).





Seaborn

Cuestiones a considerar

- Seaborn tiene una sintaxis **diferente a matplotlib**, por lo que sólo lo aprovecharemos por su punto fuerte: las visualizaciones de variables categóricas.
- Como Seaborn se construye sobre matplotlib, puede actualizar los parámetros de matplotlib con los estilos de Seaborn y seguir graficando normalmente.

```
sns.set()
```



SURFEANDO EN SEABORN



BREAK



Seaborn

Manos a la obra □

Seaborn trae algunos datasets de prueba, exploremos uno:

```
df_ejercicio = sns.load_dataset('exercise')
df_ejercicio = df_ejercicio.drop('Unnamed: 0', axis='columns')
df_ejercicio.head()
```



	id	diet	pulse	time	kind
0	1	low fat	85	1 min	rest
1	1	low fat	85	15 min	rest
2	1	low fat	88	30 min	rest
3	2	low fat	90	1 min	rest
4	2	low fat	92	15 min	rest



Seaborn

Manos a la obra □

Los valores son *mediciones del pulso* de 30 personas tras realizar algún *tipo de actividad* por un *determinado tiempo*. Las columnas son:

- **Id:** número identificador de la persona
- **Diet:** dieta de la persona 👉 baja en grasas o sin grasas
- **Time:** duración del ejercicio 👉 1 min, 15 min o 30 min
- **Kind:** tipo de ejercicio 👉 reposo, caminar o correr



Seaborn

Manos a la obra □

Observemos la distribución de las mediciones luego de 30 minutos de realizar el ejercicio.

1. Extraigamos sólo aquellas observaciones que se corresponden con un ejercicio de 30 minutos

```
df_30_min = df_ejercicio[df['time'] == '30 min']
df_30_min.head()
```



	id	diet	pulse	time	kind
2	1	low fat	88	30 min	rest
5	2	low fat	93	30 min	rest
8	3	low fat	94	30 min	rest
11	4	low fat	83	30 min	rest
14	5	low fat	91	30 min	rest



Seaborn

Manos a la obra □

2. Graficamos las distribuciones con sns.displot

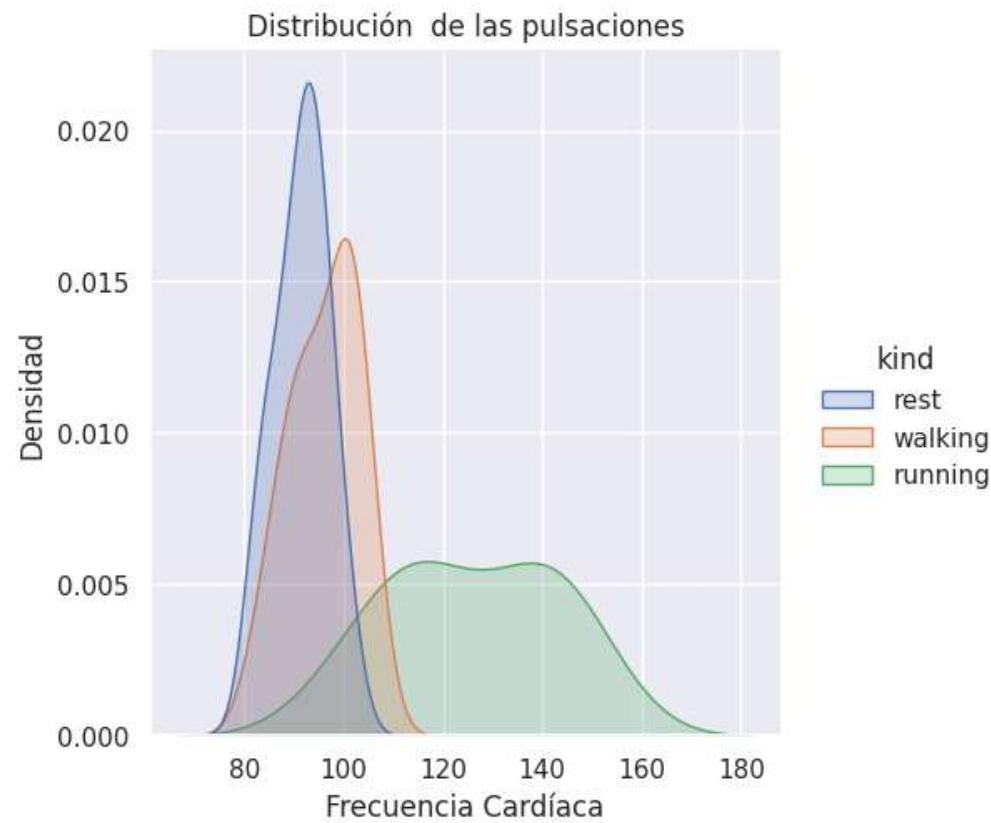
```
plt.figure()
ax = sns.displot(data=df_30_min, x='pulse', kind='kde', hue='kind', fill=True)
ax.set(xlabel='Frecuencia Cardíaca', ylabel='Densidad', title='Distribución de las pulsaciones')
```

- Debemos especificar la columna de valores en el parámetro x, en este caso nos interesa la columna *pulse*
- Como queremos separar las distribuciones según el *tipo de actividad*, pasamos el parámetro hue='kind'



Seaborn

El resultado...



Como era de esperar, aquellas personas que corrieron terminaron (*en promedio*) con una mayor frecuencia cardíaca respecto de las que no lo hicieron 😊



Seaborn

Algunas observaciones...

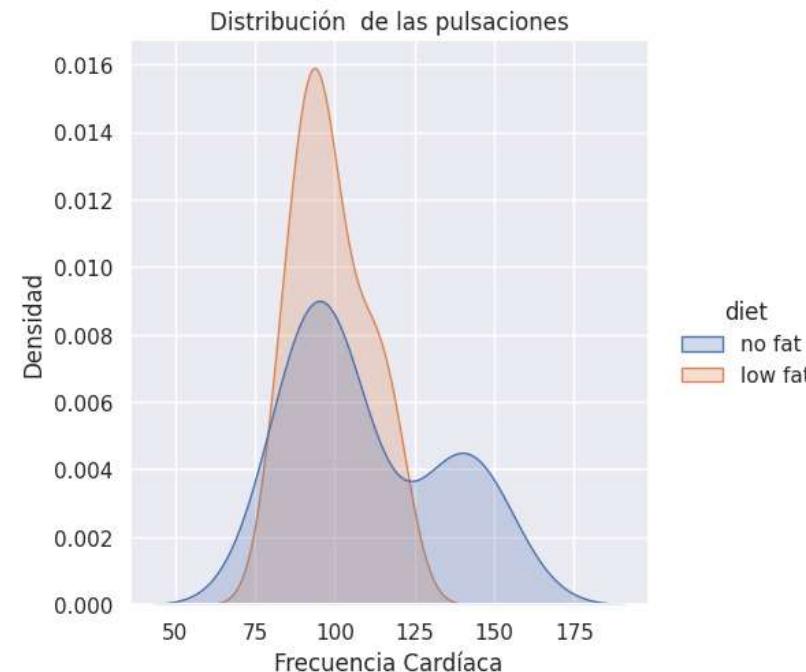
- Con el parámetro kind='kde', Seaborn realiza una estimación de la distribución a partir de los datos del Data Frame. A grandes rasgos, se puede pensar a este tipo de gráficos como una *versión suavizada del histograma de frecuencias relativas*.
- Con el parámetro kind='hist', Seaborn graficará un histograma.
- El parámetro fill añade el sombreado debajo de la distribución



Seaborn

Algunas observaciones...

- Para comparar las distribuciones en base a las dietas, sólo hay que cambiar el parámetro hue





Seaborn: Categorical Plots

Comparación desagregando por dos categorías

- Si bien los ejemplos anteriores nos permitían visualizar las distribuciones de acuerdo a una categoría, podríamos querer visualizar los datos en base a **dos** variables distintas □
- sns.catplot con el parámetro kind='violin' permite comparar distribuciones separando los datos en base a **dos categorías simultáneamente**.



Seaborn: Categorical Plots

Desagregación en base a duración y dieta

- Visualicemos las pulsaciones en base a la duración de la actividad y al tipo de dieta:

```
ax = sns.catplot(data=df_ejercicio, kind='violin', x='time', y='pulse', hue='diet', split=True)
ax.set(xlabel='Duración de ejercicio', ylabel='Frecuencia cardíaca', title='Categorización de la distribución de pulsaciones')
```

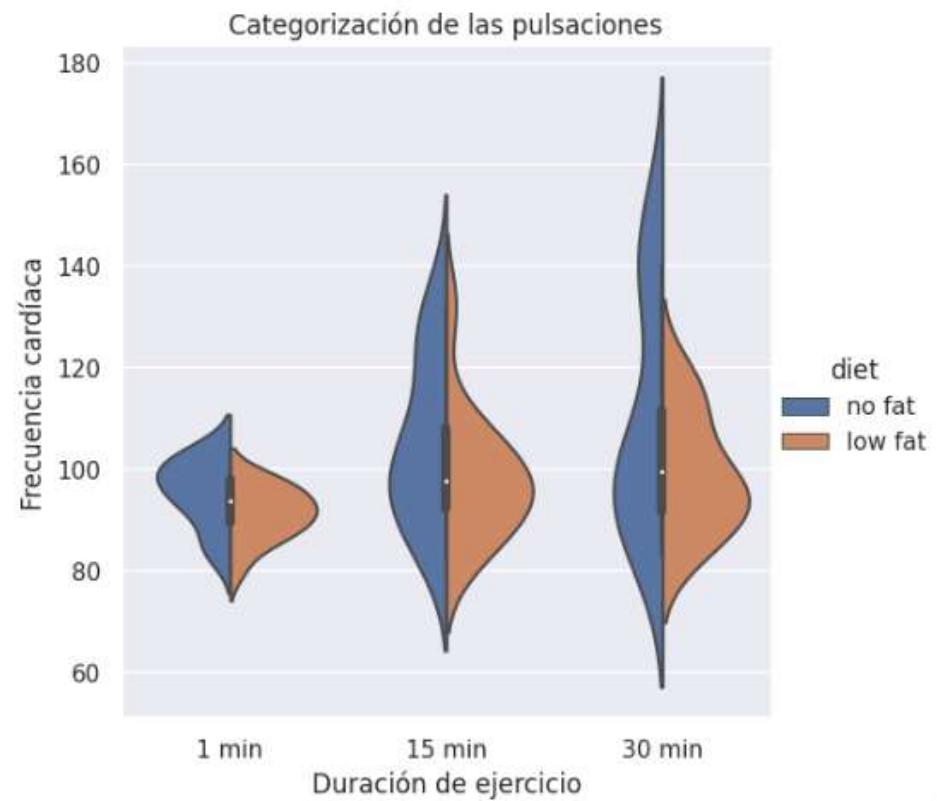


Ejemplo
en vivo



Seaborn: Categorical Plots

El resultado...





Seaborn: Categorical Plots

Desagregación en base a actividad y dieta

- Para visualizar en base a la actividad, únicamente cambiamos el valor del parámetro x

```
ax = sns.catplot(data=df_ejercicio, kind='violin', x='kind', y='pulse', hue='diet', split=True)
ax.set(xlabel='Duración de ejercicio', ylabel='Frecuencia cardíaca', title='Categorización de la distribución de pulsaciones')
```

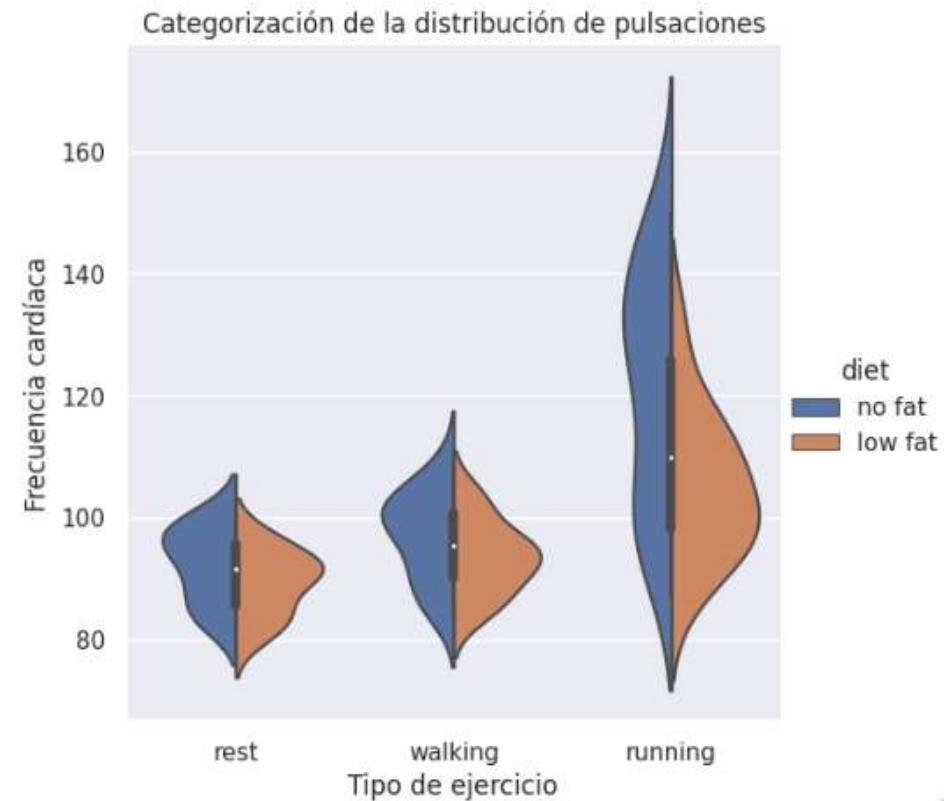


Ejemplo
en vivo



Seaborn: Categorical Plots

El resultado...





PRÁCTICA INTEGRADORA: VISUALIZACIÓN EN PYTHON

Trabajaremos sobre un conjunto de datos aplicando los conceptos aprendidos

PRÁCTICA INTEGRADORA: VISUALIZACIÓN EN PYTHON

Formato: Entregar un archivo con formato .ipynb. Debe tener el nombre “Visualizacion+Apellido”.

Sugerencia: Preparar el código y probar los resultados con subconjuntos del conjunto original.

Desafío
entregable



>> Consigna:

1. Cargar [este archivo](#) en Python. Realizar estadísticas descriptivas básicas
2. Realizar un histograma con los salarios. Qué rango(s) de salarios son los más populares?
3. Realizar un gráfico de violín con los salarios, discriminados por género.
4. Graficar la serie de tiempo correspondiente a la fecha de contratación (DateofHire)

>>Aspectos a incluir en el entregable:

El código debe estar hecho en un notebook y debe estar probado.