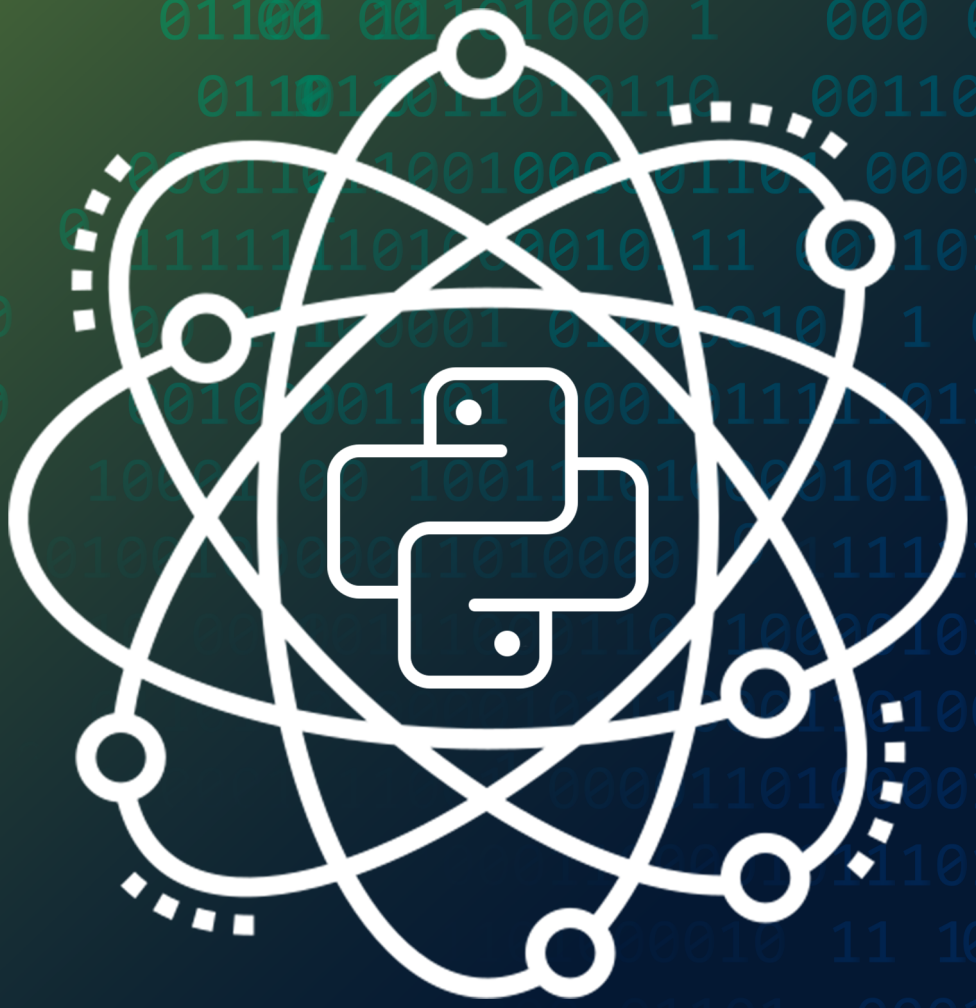


# PYTHON PARA DATA SCIENCE

LATAM

# Sumário

Capítulo 1

# INTRODUCCIÓN A PYTHON

Este curso tiene como foco principal la preparación inicial de científicos de datos para el mercado de trabajo. Se trata de un curso introductorio donde hablaremos un poco sobre manipulación, tratamiento y limpieza de conjunto de datos.

Para esta tarea iremos a tratar lo básico del lenguaje Python para que puedas dominar los conceptos principales y algunas bibliotecas de su ecosistema que son bastante utilizadas en análisis de datos.

## 1.1 UN POCO SOBRE EL LENGUAJE PYTHON

Python es un lenguaje de programación interpretado, orientado a objetos y de alto nivel. Fue creado por Guido van Rossum en 1991, Python fue idealizado para ser un lenguaje fácil, intuitivo, de código abierto y adecuado para tareas del día a día. Su simplicidad ayuda en la reducción del mantenimiento del código, su soporte a módulos y paquetes fomentan la programación modularizada y reutilización de código.

Python también posee un amplio ecosistema de bibliotecas que ofrecen conjuntos de herramientas especializadas en diversas áreas, incluyendo *Data Science*.

Es uno de los lenguajes que más ha crecido, debido a su compatibilidad (funciona en la mayoría de los sistemas operativos) y capacidad de ayudar otros lenguajes. Aplicaciones como *Dropbox*, *Reddit* e *Instagram* son escritas en Python. Python también es el lenguaje más popular para análisis de datos y se convirtió como referencia dentro de la comunidad científica.

## 1.2 ¿POR QUÉ USAR PYTHON PARA DATA SCIENCE?

El lenguaje Python no es la única herramienta disponible para hacer ciencia de datos. Existen otros, como el lenguaje R, SAS, Matlab, etc. que son bastante útiles en este sentido, pero podemos mencionar un conjunto de características y motivos que pueden haber ayudado en la popularidad del lenguaje Python en el contexto de ciencia de datos.

- **Comunidad** - En los últimos años Python desarrolló una comunidad activa y grande en el campo de análisis de datos y procesamiento científico. Este es un punto extremadamente importante para fortalecer el uso del lenguaje. Cuanto mayor el número de especialistas utilizando una herramienta mayor será el número de proyectos desarrollados, más paquetes serán implementados y mejorados

y más fácilmente encontraremos ayuda para el desarrollo de nuestros propios proyectos.

- **Fácil Aprendizaje** - Como mencionado en la sección anterior, Python fue proyectado para ser un lenguaje fácil, intuitivo y adecuado para tareas del día a día.
- **Ecosistema de bibliotecas** - Por haber alcanzado una comunidad grande y creciente en el área de ciencia de datos, el ecosistema de bibliotecas dirigidas a *Data Science* y *Machine Learning* sufrió beneficios tanto en la creación de nuevas herramientas como en la mejoría significativa en el soporte de las ya existentes.

## 1.3 PRINCIPALES PAQUETES PARA DATA SCIENCE

Existen diversos paquetes Python disponibles para hacer *download* en Internet. Cada paquete tiene como objetivo la solución de determinado tipo de problema y para eso son desarrollados nuevos tipos, funciones y métodos. Algunos paquetes son bastante utilizados en un contexto de ciencia de datos como por ejemplo:

- NumPy
- Pandas
- Scikit-learn
- Matplotlib

Algunos paquetes no son distribuidos con la instalación patrón de Python. En este caso debemos instalar los paquetes que necesitamos en nuestro sistema para poder utilizar sus funcionalidades.

En este curso haremos una introducción a los paquetes *Numpy* y *Pandas*.

## 1.4 NUESTRO AMBIENTE DE DESARROLLO

### Utilizando *Google Colab*

**Visión general de los recursos básicos (ingles)**:

https://colab.research.google.com/notebooks/basic_features_overview.ipynb

*Google Colab* puede ser entendido como un servicio de nube gratuito basado en el proyecto **Jupyter Notebook** (https://jupyter.org/). **Jupyter Notebook** es una aplicación web abierta que permite crear y compartir documentos que contienen código activo, ecuaciones, visualización y textos explicativos. Los usos incluyen: limpieza y transformación de datos, simulación numérica, modelado estadístico, visualización de datos, aprendizaje de máquina, etc.

Para utilizar ***Colab*** necesitamos apenas tener una cuenta de Google. Caso todavía no tengas una cuenta, basta acceder al siguiente link para registrarte: https://myaccount.google.com/

Entrando al link https://colab.research.google.com/notebooks/welcome.ipynb, accedemos a la ventana inicial del Google Colab.

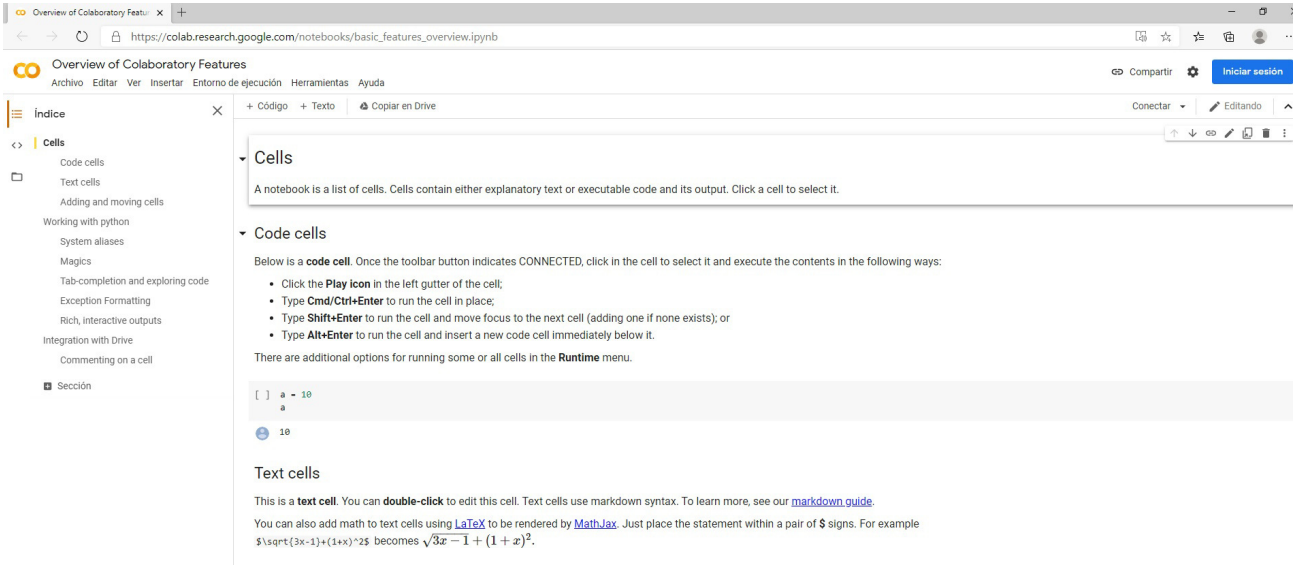



Figura 1.1: Colab Tela Inicial

## *Notebook*

Un *notebook* es un documento compuesto por **celdas** que pueden contener código ejecutable (foco de nuestro curso) y otros elementos (textos, ecuaciones, imágenes, links, etc.).

Son muy útiles en la fase inicial de desarrollo de proyectos en Data Science_ y en tareas de aprendizaje, pues permiten unir notas explicativas elaboradas, código ejecutable y los respectivos resultados.

Para crear un nuevo *notebook* en el Google Colab, luego después de hacer *login*, basta acceder al menú `Archivos > Nuevo notebook` en la esquina superior izquierda de la página. La siguiente página será exhibida en tu navegador con el nuevo *notebook* listo para editar:

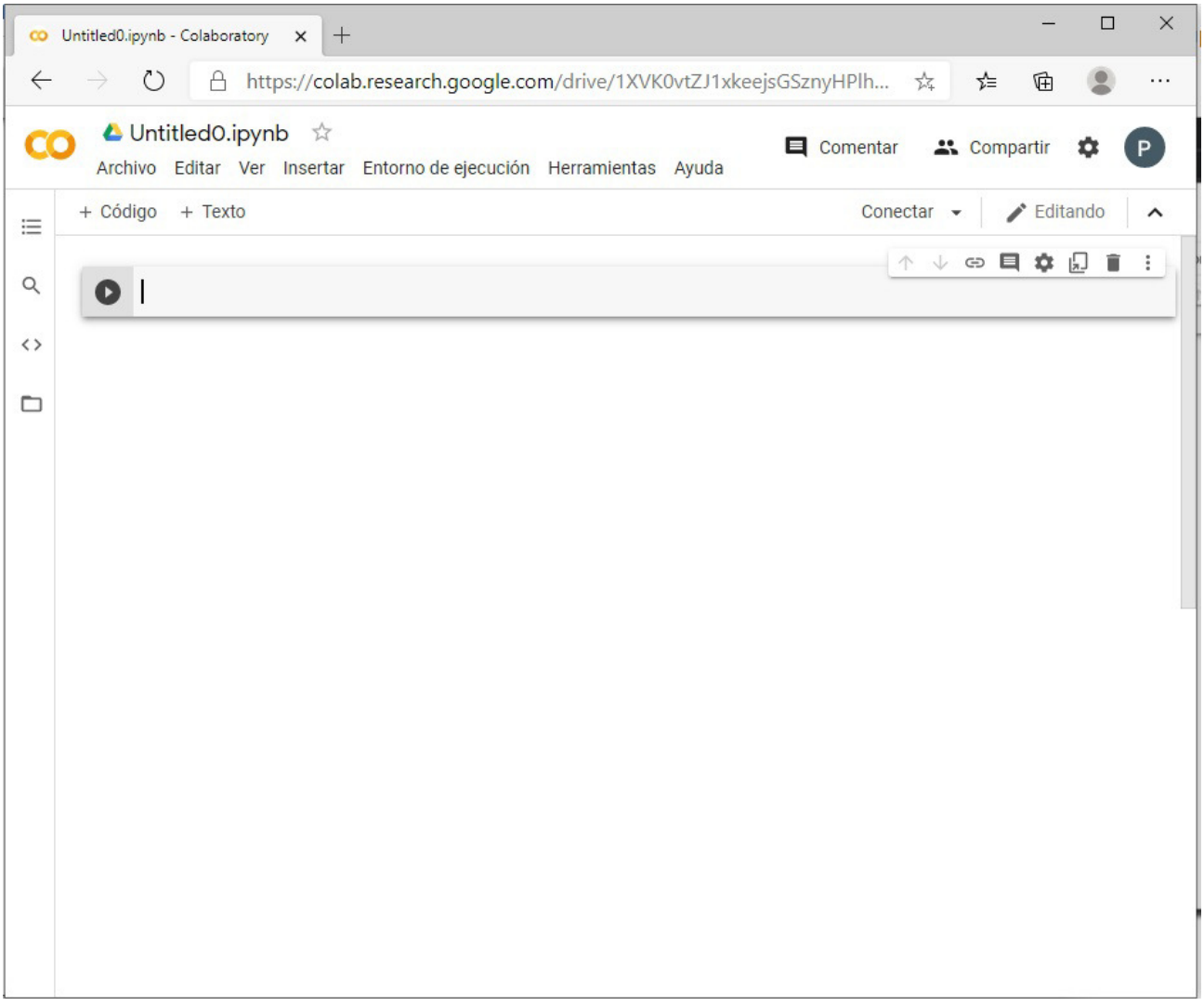



Figura 1.2: Colab Notebook

## Celdas de código

En la figura de abajo tenemos el ejemplo de una celda de código en un *notebook* de Google Colab con sus recursos destacados por letras

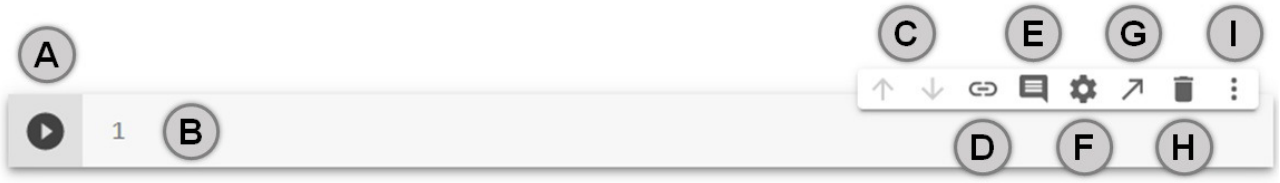



Figura 1.3: Colab Celdas

| Marca | Descripción |
|---|---|
| **A** | Ejecutar celda. |
| **B** | Área de digitación del código Python o texto. |

| | |
|---|---|
| **C** | Flechas de navegación. Posibilitan la modificación de la posición de la celda. |
| **D** | Crea un link para la celda. Permite compartir el código con otros usuarios. |
| **E** | Agrega un comentario para la celda. |
| **F** | Abre las configuraciones del editor. |
| **G** | Duplica la celda en una pestaña. |
| **H** | Borra la celda. |
| **I** | Abre un menú extra con opciones para copiar y recortar la celda y también una opción para crear formularios. |

Después de establecer la conexión (Connect) en la barra de herramientas de Colab, haz clic en la celda para seleccionarla, escribe el código Python y ejecuta el código de una de las siguientes formas:

- Haz clic en el icono **Run Cell** (ítem A de la figura de arriba) en la esquina izquierda de la celda, o
- Pulsa `Ctrl + Enter` para ejecutar la celda y mantenerla activa, o
- Pulsa `Shift + Enter` para ejecutar la celda y mover el foco para la próxima celda (adicionando una caso no exista); o
- Pulsa `Alt + Enter` para ejecutar la celda e insertar una nueva celda de código inmediatamente abajo.

Existen opciones adicionales para ejecutar algunas o todas las celdas de nuestro *notebook*. Estos comandos se encuentran en el menú `Runtime`.

## 1.5 PRIMER PROGRAMA

Como se trata de un curso de introducción sobre Python vamos a hacer un primer programa utilizando las celdas de un *notebook* del Google Colab. En todos los cursos sobre lenguajes de programación tenemos el famoso "Hello, world!", que es un programa extremadamente simple que solamente muestra las palabras "Hello, world!" en la pantalla. En Python podemos crear este nuevo programa de la siguiente forma:

```
print("Hello, World!")
```

**Salida:**

```
Hello, World!
```

En el código de arriba, utilizamos una instrucción de impresión con el uso de la función `print()`. Esta es una de las funciones *built-in* (integradas) de Python utilizada para imprimir los argumentos pasados (en nuestro ejemplo "Hello, World") en la pantalla. Noten que las comillas dobles no fueron impresas, eso porque ellas funcionan apenas como delimitadores del texto que queremos imprimir.

## 1.6 CARACTERÍSTICAS BÁSICAS DEL LENGUAJE

En esta sección hablaremos de una serie de características básicas del lenguaje Python que necesitan ser conocidas por todo programador.

## Operaciones matemáticas

Como todo lenguaje de programación , Python también posee un conjunto de operadores aritméticos para realizar operaciones matemáticas y otras operaciones que discutiremos en las próximas secciones.

**Suma (+)**

```
2 + 2
```

**Salida:**

```
4
```

**Resta (-)**

```
2 - 2
```

**Salida:**

```
0
```

**Multiplicación (*)**

```
2 * 3
```

**Salida:**

```
6
```

**División (/) y (//)**

La operación división (/) siempre retorna un número de punto flotante, hablaremos sobre los tipos de datos más adelante en este capítulo.

```
10 / 3
```

**Salida:**

```
3.3333333333333335
```

El operador // retorna apenas la parte entera del resultado de la división entre dos números. Observe el ejemplo abajo:

```
10 // 3
```

**Salida:**

```
3
```

**Potencia (**)**

Cuando necesitamos elevar un número a una determinada potencia usamos el operador **. En el código de abajo el número dos está elevado al cubo, o sea, 2 x 2 x 2 = 8.

```
2 ** 3
```

**Salida:**

```
8
```

**Resto de la división (%)**

El operador % retorna el resto de la división entre dos números.

```
10 % 3
```

**Salida:**

```
1
```

Este tipo de operación es bastante utilizada cuando queremos verificar si un determinado número es par o impar. Eso porque todo número par cuando está dividido por 2 tiene como resto 0.

```
10 % 2
```

**Salida:**

```
0
```

**Expresiones matemáticas**

Expresiones matemáticas son operaciones que contiene más de un operador, el orden de evaluación de las operaciones va a depender de un conjunto de reglas:

- Operaciones de multiplicación y división serán evaluadas primero antes que operaciones de suma y resta.

```
5 * 2 + 3 * 2
```

**Salida:**

```
16
```

- Los paréntesis pueden ser utilizados para forzar la evaluación de determinada operación en el orden que deseamos. En el ejemplo de abajo, la operación dentro de los paréntesis es realizada primero y después las multiplicaciones

```
5 * (2 + 3) * 2
```

**Salida:**

```
50
```

- Después de los paréntesis la operación de potencia tiene el orden de prioridad más elevado. El

resultado del código abajo es 40 y no 1000

```
5 * 2 ** 3
```

**Salida:**

```
40
```

- Caso los operadores tengan las misma orden de prioridad (multiplicación y división o suma y resta) la evaluación será realizada de izquierda a derecha.

**La variable guion bajo '_'**

En el modo interactivo, Python almacena automáticamente el valor de la última expresión en la variable "_". Observe el resultado de la expresión abajo.

```
5 * 2
```

**Salida:**

```
10
```

En la próxima celda podemos usar "_", que en este caso está el valor 10, como parte de una expresión matemática. Veamos el próximo código.

```
_ + 3 * 2
```

**Salida:**

```
16
```

Existen otras utilizaciones para la variable "_", que veremos a lo largo de este curso.

## Variables

Variables pueden ser entendidas como espacios en la memoria del computador a los cuales se le asigna un nombre y donde serán almacenados valores que pueden ser alterados durante la ejecución del programa.

En algunos lenguajes de programación, las variables son consideradas como *containers* donde almacenamos datos. En Python, por otro lado, las variables pueden ser mejor definidas como punteros y no como *containers*. En resumen, cuando hacemos una asignación a una variable estamos creando una referencia a un objeto.

**Nombre de variables**

Python tiene algunas reglas para escoger el nombre para las variables.

- Nombre de variables pueden comenzar con letras (a - z, A - Z) o con el carácter guion bajo (_)

```
Altura
```

```
_peso
```

-El resto del nombre puede contener letras. números y el carácter "_":

```
nombre_de_variable
_valor
dia_28_11_
```

- Los nombre son ***case sensitive***, o sea, existe diferencia entre letra mayúscula y minúscula: **Nombre_De_Variable ≠ nombre_de_variable ≠ NOMBRE_DE_VARIABLE**
- A partir del Python 3 ya es posible utilizar las tildes "´", la "ñ" , y la "ü" en el nombre de las variables. Sin embargo, recomendamos evitar en lo posible para simplificar los nombres de las variables.
- Existen algunas palabras reservadas del lenguaje que no pueden ser utilizadas como nombres de variable:

| | **Lista de palabras reservadas en Python** | |
|---|---|---|
| and | as | not |
| assert | finally | or |
| break | for | pass |
| class | from | nonlocal |
| continue | global | raise |
| def | if | return |
| del | import | try |
| elif | in | while |
| else | is | with |
| except | lambda | yield |
| False | True | None |

**Declaración de variables**

Para crear una variable en Python basta escoger el nombre, utilizar el operador de asignación (=) y pasar el valor que será referenciado.

Abajo creamos 3 variables para uno de los proyectos que vamos desarrollar a lo largo de este curso. Como estamos aprendiendo Python en un contexto de *Data Science*, necesitamos trabajar con datos, y durante las clases utilizaremos informaciones de dos *datasets* ficticios, uno con datos sobre un conjunto de carros disponibles para la venta y el otro con datos sobre el conjunto de inmuebles disponibles para locación.

```
anho_actual = 2020
```

```
anho_fabricacion = 2003
km_total = 44410.0
```

Como dijimos anteriormente, evitamos usar caracteres especiales como es el caso de la letra "ñ", por ello que la palabra año, la nombramos "anho".

Para consultar el valor de una variable en un *notebook* basta digitar el nombre de la variable en la celda y ejecutar el código.

```
anho_actual
```

**Salida:**

```
2020
```

Podemos también utilizar la función `print()` para visualizar el valor de más de una variable a partir de una única celda.

```
print(anho_fabricacion)
print(km_total)
```

**Salida:**

```
2003
44410.0
```

**Operaciones con variables**

En proyectos de *Data Science* una de las tareas que necesita ser realizada por el científico de datos es la creación de nuevas variables a partir de las ya existentes. Este tipo de combinación puede ayudar al entendimiento de algunos fenómenos que están siendo investigados en el estudio.

Con el uso de variables, este tipo de procedimiento queda bastante facilitado y mejor documentado.

En nuestro proyecto vamos a necesitar calcular el kilometraje medio de los vehículos de nuestro *dataset*. Esta variable puede ser de gran valor caso necesitemos generar un modelos de *Machine Learning* para cálculo de precios de vehículos usados.

La fórmula representada abajo muestra el cálculo que necesitamos hacer para crear la variable **km_media**.

$$km_{medio} = \frac{km_{total}}{(año_{actual} - año_{fabricacion})}$$

Figura 1.4: Fórmula

Recordando las reglas de precedencia utilizando variables creadas anteriormente, tenemos el siguiente código:

```
km_media = km_total / (anho_actual - anho_fabricacion)
km_media
```

**Salida:**

```
2775.625
```

## Tipos de dados

Los tipos de datos especifican como números y caracteres serán almacenados y manipulados dentro de un programa. Los tipos de datos básicos en Python son:

1. **Numéricos**
   - ***int*** - Enteros
   - ***float*** - Punto flotante
2. **Booleanos** - Asume los valores True o False. Esencial cuando comenzamos a trabajar con declaraciones condicionales
3. **Cadena de caracteres (Strings)** - Secuencia de uno o más caracteres que pueden incluir letras, números u otros tipos de caracteres. Representa un texto.
4. **None** - Representa la ausencia de valor

En el lenguaje Python tenemos un modelo de objetos muy bien estructurado. En Python todo es un objeto. Clases, módulos, funciones, strings, números, etc. son referenciados como objetos Python por el intérprete y poseen su propio tipo asociado y una serie de otras informaciones.

Por este motivo las referencias a objetos (nombre de variables, funciones, etc.) no poseen ningún tipo asociado a ellas. En Python las informaciones sobre el tipo de objeto quedan almacenadas en el propio objeto.

Como todo objeto tienen un tipo asociado, Python puede ser considerado un lenguaje fuertemente tipado.

#### Números

En Python tenemos dos objetos para representar los números, el `int` y el `float` .

```
anho_actual = 2020
```

Para verificar el tipo del objeto que una variable referencia basta utilizar la función *built-in* `type()` y pasar como argumento el nombre de una variable. Hablaremos más sobre las funciones *built-in* (integradas) en el capítulo de funciones.

```
type(anho_actual)
```

**Salida:**

```
int
```

Para tener un tipo `float`, que es utilizado para representar número con partes fraccionarias, debemos atribuir por lo menos una casa decimal. Observe el ejemplo abajo:

```
km_total = 44410.0
```

Utilizando nuevamente la función `type()` vemos que se trata de un tipo `float`.

```
type(km_total)
```

**Salida:**

```
float
```

**Booleanos**

Tipo utilizado para representar valores lógicos que pueden asumir apenas dos valores: *True* (verdadero) o *false* (falso).

```
cero_km = True
```

En los próximos capítulos trabajaremos también con expresiones booleanas

```
type(cero_km)
```

**Salida:**

```
bool
```

***Strings***

*Strings* son secuencias inmutables de caracteres Unicode que son utilizadas para representar texto. Podemos escribir *strings* utilizando como delimitadores las comillas simples (' ').

```
nombre = 'Jetta Variant'
nombre
```

**Salida:**

```
'Jetta Variant'
```

Y también las comillas dobles (" "):

```
nombre = "Jetta Variant"
nombre
```

**Salida:**

```
'Jetta Variant'
```

Para crear *strings* con múltiples líneas podemos utilizar comillas triplas, que pueden ser ''' ou """.

```
carro = '''
```

```
Nombre
edad
Nota
'''
```

Verificando el tipo de dato de una *string* con la función `type()` tenemos como respuesta `str` , que es el objeto utilizado para representar datos textuales en Python.

```
type(carro)
```

**Salida:**

```
str
```

**None**

*None* es la forma como el tipo nulo es representado en el lenguaje Python.

```
kilometraje = None
kilometraje
```

verificando el tipo de una variable *None* tenemos como respuesta `NoneType` .

```
type(kilometraje)
```

**Salida:**

```
NoneType
```

## Conversión de tipos

Considere las cuatro variables abajo que **a** y **b** son números enteros y **c** y **d** son *strings*.

```
a = 10
b = 20
c = 'Python es '
d = 'chévere'
```

Como vimos anteriormente, podemos realizar operaciones entre **a** y **b** sin ningún problema.

```
a + b
```

**Salida:**

```
30
```

Ahora ¿que pasaría si utilizamos el operador suma con las variables **c** y **d**?

```
c + d
```

**Salida:**

```
'Python es chévere'
```

El operador suma gana una nueva funcionalidad cuando utilizamos objetos de tipo *string*. En estos casos el operador “+” funciona como un concatenador, o sea junta las *strings*. Observe el resultado de

arriba. Pero y qué pasa si utilizamos este mismo operador con tipos diferentes:

```
c + a
```

**Salida:**

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-87-72f9c903e6a0> in <module>
----> 1 c + a

TypeError: can only concatenate str (not "int") to str
```

Un `TypeError` será generado informando que solo es posible la concatenación entre *strings*.

Como Python es fuertemente tipado, las conversiones de tipo sólo ocurren en algunas situaciones específicas, en los demás casos necesitamos realizar la conversión de forma explícita.

## Funciones `int()`, `float()`, `str()` y `bool()`

Los tipos `int`, `float`, `str` y `bool` son también funciones integradas de Python y pueden ser utilizadas para realizar la conversión directa del tipo.

Usando la función `str()` y pasando como argumento la variable **a**, que es de tipo `int`, tenemos el siguiente resultado:

```
str(a)
```

**Salida:**

```
'10'
```

Note que ahora el número 10 está representado en el *output* de la celda con comillas simples ('10') indicando que se trata de un *string* y no más de tipo numérico.

```
type(str(a))
```

**Salida:**

```
str
```

Con este recurso, ahora podemos realizar la concatenación de variables **c** y **a** de la siguiente forma:

```
c + str(a)
```

**Salida:**

```
'Python es 10'
```

Observe que si utilizamos la función `int()` en un número que tenga casas decimales ( `float` ), apenas la parte entera del número será considerada.

```
var = 3.141592
int(var)
```

**Salida:**

```
3
```

## Indentación, comentarios y formato de *strings*

### Indentación

En el lenguaje Python los programas son estructurados por medio de indentación. En cualquier lenguaje de programación la práctica de indentación es bastante útil, facilitando la lectura y también el mantenimiento del código. En Python la indentación no es solamente una cuestión de organización y estilo, más si un requisito del lenguaje.

Observe el siguiente código:

```
anho_actual = 2020
anho_fabricacion = 2020

if (anho_actual == anho_fabricacion):
    print('Verdadero')
else:
    print('Falso')
```

**Salida:**

```
Verdadero
```

Hablaremos sobre cláusulas `if-else` en los próximos capítulos, en el momento quería apenas llamar la atención para dos detalles. El primero es para el carácter `:` que indica el inicio de un bloque de código, y el segundo es el espacio (pueden ser tabulaciones o espacios) en la línea luego abajo de la instrucción que termina con `:`.

Eso indica para el intérprete que el comando `print('verdadero')` hace parte del bloque del código de instrucción `if`, o sea, si **anho_actual** es igual al **anho_fabricacion** será impreso en la consola la palabra 'Verdadero".

Luego vemos que la estructura se repite para el bloque del código de instrucción `else`.

### Comentarios

Los comentarios son extremadamente importantes en un programa. Consiste en un texto que describe lo que el programa o una parte específica del programa está haciendo. Los comentarios son ignorados por el intérprete de Python.

Podemos tener comentarios de una única línea o de múltiples líneas. Vea abajo los ejemplos.

```
# Este es un comentario
anho_actual = 2020

'''Este es un
comentario'''
```

```
anho_actual = 2020
```

```
# Definiendo variables
anho_actual = 2020
anho_fabricacion = 2020

'''
Estructura condicional que vamos a
aprender en las próximas clases
'''
if (anho_actual == anho_fabricacion):   # Probando si  la condición es verdadera
    print('Verdadero')
else:                                   # Probando si la condición es falsa
    print('Falso')
```

**Salida:**

```
Verdadeiro
```

### Formato de *strings*

Para finalizar este primer capítulo del curso vamos a aprender dos formas de mejorar las salidas (*outputs*) de nuestros códigos.

#### `str.format()`

**Documentación**: https://docs.python.org/3.6/library/stdtypes.html#str.format El método `format()` del objeto `str` ejecuta una operación de formato en un *string*. El *string* en el cual este método es llamado puede contener texto literal o campos de sustitución delimitados por llaves {}.

```
print('Hola, {}!'.format('Rodrigo'))
```

**Salida:**

```
Hola, Rodrigo!
```

Cada campo de sustitución {} contiene un índice numérico correspondiente a los argumentos pasados o nombre de un argumento.

```
print('Hola, {}. Este es su acceso de número {}'.format('Rodrigo', 32))
```

**Salida:**

```
Hola, Rodrigo. Este es su acceso de número 32
```

Este método retorna una copia del *string* donde cada campo de sustitución {} es modificado por el valor del argumento correspondiente.

```
print('Hola, {nombre}. Este es su acceso de número {accesos}'.format(nombre = 'Rodrigo', accesos = 32
))
```

**Salida:**

```
Hola, Rodrigo. Este es su acceso de número 32
```

#### *f-strings*

**Documentación**: https://docs.python.org/3.6/library/stdtypes.html#str.format

Un *f-string* es un *string* literal que tiene como prefijo la letra `f` o `F`. Estes *string* también pueden contener campos de sustitución {}. La diferencia del método anterior es que estos campos pueden ser llenados por cualquier variable del programa.

```
nombre = 'Rodrigo'
accesos = 32

print(f'Hola, {nombre}. Este es su acceso de número {accesos}')
```

**Salida:**

```
Hola, Rodrigo! Este é seu acesso de número 32
```

## 1.7 PARA SABER MÁS ...

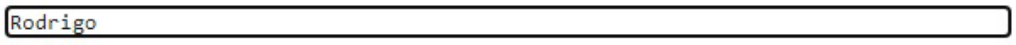
### `input()`

**Documentación**: https://docs.python.org/3/library/functions.html#input

La función `input()` lee la línea de entrada, convierte a un *string* y retorna ese valor. Para leer esa línea de entrada, una caja de texto es mostrada para que el usuario rellene con el contenido solicitado.

```
input()
```

**Salida:**

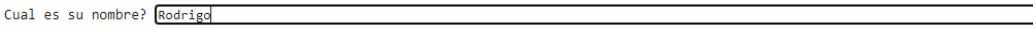



```
'Rodrigo'
```

Es posible agregar un texto explicativo para ayudar al usuario con el llenado del valor.

```
input('¿Cuál es su nombre? ')
```

**Salida:**



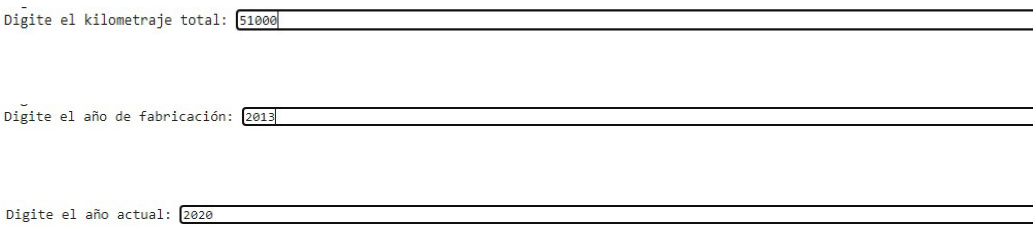

```
¿Cuál es su nombre? Rodrigo
```

En el siguiente código creamos un programa para pedir algunas informaciones sobre el vehículo del usuario y retornar el kilometraje medio de este vehículo.

```
anho_actual = input('Digite el año actual: ')
anho_fabricacion = input('Digite el año de fabricación: ')
km_total = input('Digite el kilometraje total: ')

km_media = int(km_total) / (int(anho_actual) - int(anho_fabricacion))
km_media
```

**Salida:**




```
7285.714285714285
```

Note que tuvimos que utilizar la función `int()` antes de realizar el cálculo de la media. Eso fue necesario porque la función `input()` convierte el valor digitado por el usuario en un *string*.

## 1.8 EJERCICIOS

El objetivo de este primer capítulo fue conocer los aspectos básicos del lenguaje Python. Ahora vamos a practicar un poco de este conocimiento que adquirimos.

1. Utilizando apenas las variables declaradas en la celda de abajo, imprima la frase: **"Sumando 10 con 20 tenemos 30"**.

```
var_1 = 10
var_2 = 20
var_3 = 'Sumando '
var_4 = ' con '
var_5 = ' tenemos '
```

**Respuesta:**

```
var_3 + str(var_1) + var_4 + str(var_2) + var_5 + str(var_1 + var_2)
```

**Salida:**

```
'Sumando 10 con 20 tenemos 30'
```

2. Escriba un programa que reciba el año del nacimiento de un usuario y retorna su edad en años en el año 2020.

Sugerencia: utilice los conocimientos adquiridos en la sección "para saber más..." en la construcción de este programa

**Respuesta:**

```
anho = input("Digite el año de su nacimiento:")

edad = 2020 - int(anho)
print('En el 2020 tienes {edad} años.')
```

**Salida:**

```
Digite el año de su nacimiento:1976
En el 2020 tienes 44 años.
```

3. Escriba un programa que reciba las informaciones sobre el peso (kilos) y la altura (metros) de un usuario e imprima su **índice de masa corporal (IMC)**. Para obtener el IMC debemos dividir el peso del usuario por su altura elevada al cuadrado.

   Sugerencia: Utilice el conocimiento adquirido en la sección "para saber más..." en la construcción de este programa.

   **Respuesta:**

```
peso = input("Digite su peso (kilos): ")
altura = input("Digite sua altura (metros): ")

imc = float(peso) / (float(altura) ** 2)
print(f'Su IMC es {imc}.')
```

   **Salida:**

```
Digite su peso (kilos): 100
Digite su altura (metros): 1.9
Su IMC es 27.70083102493075.
```

CAPÍTULO 2

# TRABAJANDO CON SECUENCIAS

En esta sección vamos a conocer tres tipos básicos de secuencias del lenguaje Python: **listas**, **tuplas** y ***range***. Podemos clasificar estas secuencias en dos grupos: secuencias mutables y secuencias inmutables.

En el grupo de secuencias mutables tenemos las **listas**, que son generalmente utilizadas para almacenar colecciones de elementos homogéneos.

Los tipos de secuencias **tuplas** y ***range*** son clasificados como inmutables. Las **tuplas** son normalmente usadas para almacenar colecciones de datos heterogéneos, pero también son usadas en los casos en que es necesaria una secuencia inmutable de datos homogéneos.

El tipo ***range*** representa una secuencia inmutable de números y es comúnmente usado con iterador en bucles o *loops*.

## 2.1 CREANDO SECUENCIAS

### Listas

Una lista puede ser entendida como una secuencia de valores que pueden ser de cualquier tipo. Estos valores pueden ser llamados de elementos o ítems y cada uno es identificado por un índice, iniciado por 0. Las listas son delimitadas por corchetes y sus elementos son separados por coma. En seguida tenemos algunas formas de crear una lista en Python.

- Utilizando un par de corchetes: `[] , [1]`
- Utilizando un par de corchetes con ítems separados por coma: `[1, 2, 3]`
- Utilizando `list() o list(iterator)`

El código de abajo crea una lista de **accesorios** con algunos accesorios de vehículos de nuestro *dataset*. En esta lista, todos los elementos son de tipo *string*, pero no es obligatorio que todos los elementos de una lista tengan el mismo tipo, como veremos en la próxima sección.

```
accesorios = ['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero',
'Aire acondicionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia']
accesorios
```

**Salida:**

```
['Llantas de aleación',
```

```
 'Cerraduras eléctricas',
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia']
```

Verificando el tipo de datos de la variable **accesorios** vemos que se trata de tipo `list` .

```
type(accesorios)
```

**Salida:**

```
list
```

Otra forma bastante útil de crear una lista es utilizando la función de tipo `list()` . Para esta función podemos pasar como argumento los elementos de una nueva lista en formato de tupla ( veremos tuplas más adelante) o hasta mismo una lista.

```
list((1, 2, 3))
```

**Salida:**

```
[1, 2, 3]
```

También es posible pasar como argumento una *string*. En Python, las *strings* son secuencias inmutables de caracteres y la función de tipo `list()` convierte esa secuencia en una lista de caracteres, como vemos en el código abajo.

```
list('12345')
```

**Salida:**

```
['1', '2', '3', '4', '5']
```

**Listas con tipos de datos variados**

Como hablamos en la sección anterior, las listas no necesitan contener elementos del mismo tipo, En el siguiente ejemplo tenemos dos listas (**carro_1** y **carro_2**) que representan un conjunto de informaciones de dos vehículos de nuestro *dataset*.

```
carro_1 = ['Jetta Variant', 'Motor 4.0 Turbo', 2003, 44410.0, False, ['LLantas de aleación', 'Cerradu
ras  eléctricas', 'Piloto automático'], 88078.64]
carro_2 = ['Passat', 'Motor Diesel', 1991, 5712.0, False, ['Central multimedia', 'Techo panorámico',
'Frenos ABS'], 106161.94]
```

Los ejemplos de arriba muestran la posibilidad de crear listas que por su vez contienen listas como elementos. Estas listas son conocidas como anidadas y pueden ser construidas con varios niveles. En la celda de abajo creamos una lista de **carros** que tiene como elementos las listas **carro_1** y **carro_2**.

```
carros = [carro_1, carro_2]
carros
```

**Salida:**

```
[['Jetta Variant',
  'Motor 4.0 Turbo',
  2003,
  44410.0,
  False,
  ['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático'],
  88078.64],
 ['Passat',
  'Motor Diesel',
  1991,
  5712.0,
  False,
  ['Central multimedia', 'Techo panorámico', 'Frenos ABS'],
  106161.94]]
```

## Tuplas

Así como las listas, tuplas son secuencia de valores que pueden ser de cualquier tipo y son indexados por número entero iniciando con índice 0. La principal diferencia entre listas y tuplas está en la inmutabilidad. Tuplas son secuencia inmutables y puede ser construidas de varias formas.

- Utilizando un par de paréntesis: `()`
- Utilizando una coma a la derecha: `x,`
- Utilizando un par de paréntesis con ítems separados por coma: `(x, y, z)`
- Utilizando: `tuple()` o `tuple(iterador)`

Otra diferencia entre lista y tuplas está en el delimitador, que en las tuplas son los paréntesis. Abajo creamos una tupla con tres valores numéricos.

```
(1, 2, 3)
```

**Salida:**

```
(1, 2, 3)
```

Podemos también omitir los paréntesis e insertar los elementos separados por coma.

```
1, 2, 3
```

**Salida:**

```
(1, 2, 3)
```

Conforme citamos encima, podemos tener tuplas con elementos de tipos variados.

```
nombre = 'Passat'
valor = 153000
(nombre, valor)
```

**Salida:**

```
('Passat', 153000)
```

Otra forma de crear tuplas es usando la *built-in function* `tuple()` que acepta como argumento una secuencia (*string*, lista o tupla).

```
tuple([1, 2, 3])
```

**Salida:**

```
(1, 2, 3)
```

Arriba convertimos una lista en tupla y en el código de abajo estamos utilizando una *string* para crear una tupla.

```
tuple('12345')
```

**Salida:**

```
('1', '2', '3', '4', '5')
```

## *Range*

**Documentación**: https://docs.python.org/3.6/library/stdtypes.html#ranges

El tipo ***range*** representa una secuencia inmutable de números y es utilizado como iterador en procedimientos que necesitan ser repetidos un cierto número de veces (hablaremos sobre bucles o *loops* en los próximos capítulos). Puede ser creada de la siguiente forma: `range(start,stop,step)`

Los argumentos `start`, `stop` y `step` deben ser números enteros. Caso sea omitido el argumento `step`, por defecto será usado el 1. Si el argumento `start` es omitido, por defecto será el 0. Si el argumento `step` es definido como cero, un `valueError` será generado.

Normalmente declaramos `range()` en su forma más simple que solo necesita que informemos el argumento `stop`, que es el índice de parada de la secuencia.

```
range(10)
```

**Salida:**

```
range(0, 10)
```

Podemos utilizar la función de tipo `list()` como una forma de materializar un iterador y así poder verificar su contenido en el formato de una lista de Python.

```
list(range(10))
```

**Salida:**

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

Note que el argumento `stop` define el índice de parada, que no es incluido en el resultado final.

Ahora, suponga que queremos un iterador que nos genere la siguiente secuencia:

`[10,12,14,16,18,20]` . Para eso necesitamos ejecutar la siguiente línea de código:

```
list(range(10, 21, 2))
```

**Salida:**

```
[10, 12, 14, 16, 18, 20]
```

En el código anterior configuramos el argumento `start` como 10, el `stop` como 21 ( `stop=22` también generaría el mismo resultado), y el argumento `step` como 2 (para generar apenas los índices pares). Note que el argumento `step` determina cuantos pasos debemos saltar entre las iteraciones.

La ventaja de usar *range* en relación a las listas o tuplas es que un objeto `range` siempre ocupará la misma cantidad de memoria, no importa el tamaño del intervalo que él representa, eso porque él almacena apenas los valores de `start` , `stop` y `step` .

## 2.2 OPERACIONES CON SECUENCIAS

**Documentación**: https://docs.python.org/3.6/library/stdtypes.html#common-sequence-operations

Las operaciones mostradas en la tabla abajo son soportadas por la mayoría de los tipos de secuencia, mutables e inmutables. En la tabla las letras ***A*** y ***B*** representan dos secuencias , ***X*** es un objeto arbitrario que sigue las restricciones de tipo y valor impuestas por la secuencia ***A***. Los índices ***i***, ***j*** y ***k*** son números enteros.

| Operación | Resultado |
|---|---|
| `x in A` | Retorna *True* si un elemento de la secuencia *A* es igual a *x*. |
| `x not in A` | Retorna *False* si un elemento en *A* es igual a *x*. |
| `A + B` | Concatena las secuencias *A* y *B*. |
| `len( A )` | Retorna el tamaño de la secuencia *A*. |
| `A[i]` | Retorna el i-ésimo ítem de la secuencia *A*. |
| `A[i:j]` | Recorta la secuencia *A* del índice *i* hasta el *j*. En esta secuencia el elemento con índice *i* es incluido y el elemento con índice *j* no es incluido en el resultado. |
| `A[i:j:k]` | Recorta la secuencia *A* del índice *i* hasta el *j* con el paso *k*. En esta secuencia el elemento con índice *i* es incluido y el elemento con índice *j* no es incluido en el resultado. |
| `A.index(x)` | Retorna el índice de la primera ocurrencia de *x* en la secuencia *A*. |
| `A.count(x)` | Retorna el total de ocurrencias de *x* en la secuencia *A*. |

Vamos a ver una secuencia de ejemplos de la utilización de estas operaciones con algunas listas simples.

**`x in A`**

El código `x in A` retorna *True* si un elemento de la secuencia *A* es igual a *x*. Considera la lista de

abajo para nuestros ejemplos.

```
accesorios = ['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero',
'Aire  acondicionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia']
```

El próximo código verifica si el ítem "llantas de aleación" pertenece a la lista **accesorios**

```
'Llantas de aleación' in accesorios
```

**Salida:**

```
True
```

## x not in A

La prueba contraria puede ser realizada con la adición de palabras llave `not` . El código `x not in A` retorna *False* si algún elemento en *A* es igual a *x*.

```
'Llantas de aleación' not in accesorios
```

**Salida:**

```
False
```

**`A + B`**

El signo de suma (+) para secuencias funciona como un concatenador, o sea *A* y *B* dos secuencias, la expresión `A+B` junta estas secuencias (concatena).

Considere las dos tuplas de accesorios **A** y **B** siguientes:

```
A = ('LLantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero')
B = ('Aire acondicionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia')
```

Para juntar las dos tuplas de arriba basta ejecutar el siguiente código:

```
A + B
```

**Salida:**

```
('Llantas de aleación',
 'Cerraduras eléctricas',
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia')
```

**`len(A)`**

La función integrada `len()` retorna el tamaño ( número de elementos) de un objeto. El argumento puede ser una secuencia o una colección.

```
len(accesorios)
```

**Salida:**

```
8
```

**`A.index(x)`**

El método de secuencia `index()` retorna el índice de la primera ocurrencia del elemento que es pasado como argumento, o sea, `A.index()` retorna el índice de la primera ocurrencia de *x* en la secuencia *A*.

```
nombres_carros = ('Audi', 'BMW', 'Porsche', 'Ferrari', 'Volkswagen', 'Ferrari', 'BMW')
nombres_carros.index('Ferrari')
```

**Salida:**

```
3
```

Observe que tenemos más de un ítem "Ferrari" en la tupla **nombre_carros**, pero solamente el índice de la primera ocurrencia fue retornado. Recordando que la indexación de secuencia comienza con cero.

**`A.count(x)`**

Retorna el total de ocurrencias de x en la secuencia A. Utilizando la misma tupla del ejemplo anterior:

```
nombres_carros.count('Ferrari')
```

**Salida:**

```
2
```

**`A[i]`**

El uso de corchetes después del nombre de una secuencia permite acceder a uno o más ítems de esta secuencia apenas informando su posición (índice). En este contexto [] son conocidos como operador de indexación.

Para obtener como retorno el i-ésimo elemento de la secuencia *A* utilizamos `A[i]` .

```
accesorios = ['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero',
'Aire acondicionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia']
```

Como sabemos, las secuencias tienen indexación con origen en cero y, por tanto , para acceder al primer elemento de una lista podemos proceder de la siguiente forma:

```
accesorios[0]
```

**Salida:**

```
'Llantas de aleación'
```

Para acceder al segundo elemento de la lista utilizamos el índice 1 y así sucesivamente.

```
accesorios[1]
```

**Salida:**

```
'Cerraduras eléctricas'
```

Podemos utilizar índices negativos como forma alternativa para selección de ítems en secuencias en el orden inverso. Siendo que para acceder al último elemento de una lista utilizamos el índice -1, para el penúltimo utilizamos el índice -2, y así sucesivamente.

```
accesorios[-1]
```

**Salida:**

```
'Sensor de lluvia'
```

Veamos ahora cómo este tipo de selección funciona con listas anidadas. Para eso considera la lista de **carros** que creamos en una de las secciones anteriores de este capítulo.

```
carros
```

**Salida:**

```
[['Jetta Variant',
  'Motor 4.0 Turbo',
  2003,
  44410.0,
  False,
  ['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático'],
  88078.64],
 ['Passat',
  'Motor Diesel',
  1991,
  5712.0,
  False,
  ['Central multimedia', 'Techo panorámico', 'Frenos ABS'],
  106161.94]]
```

Noten que se trata de una lista anidada con tres niveles de listas. Cuando utilizamos el código de abajo seleccionamos el primer elemento de la lista **carros**. Este elemento es también una lista con sus propios ítems.

```
carros[0]
```

**Salida:**

```
['Jetta Variant',
 'Motor 4.0 Turbo',
 2003,
 44410.0,
 False,
 ['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático'],
 88078.64]
```

Para acceder a los ítems de una lista que pertenece a otra lista, podemos utilizar operadores de indexación agrupados de la siguiente forma:

```
carros[0][0]
```

**Salida:**

```
'Jetta Variant'
```

El código anterior selecciona el primer elemento de la lista **carros** y después selecciona el primer elemento de esta selección. Abajo tenemos un ejemplo similar, solo que ahora seleccionamos el primer elemento de la lista **carros** y después el penúltimo elemento (índice -2) de la primera selección.

```
carros[0][-2]
```

**Salida:**

```
['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático']
```

Observe que el ejemplo de arriba resultó en una tercera lista que puede tener sus ítems accedidos con el agrupamiento de un operador adicional de indexación.

```
carros[0][-2][1]
```

**Salida:**

```
'Cerraduras eléctricas'
```

La siguiente ilustración resume de forma simple el proceso de selección de una lista anidada.

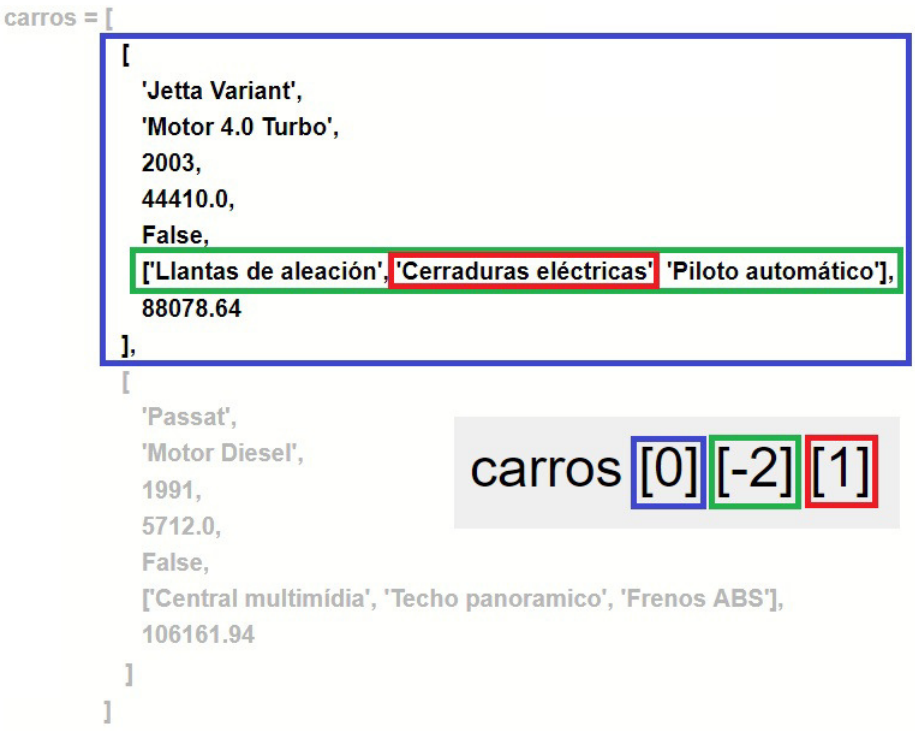



Figura 2.1: Seleção em Listas Anidadas

**A[i:j]**

*Slice* o trozos es un procedimientos que permite seleccionar una parte especifica de una secuencia. `A[i,j]` selecciona la secuencia *A* del índice i hasta el j. En este *Slice* el elemento con índice i es **incluido** y el elemento con índice j **no es incluido** en el resultado. Vamos a ver un ejemplo con la lista **accesorios**.

```
accesorios
```

**Salida:**

```
['Llantas de aleación',
 'Cerraduras eléctricas',
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia']
```

El código de abajo selecciona apenas los ítems de índices 2, 3 y 4. Observa el resultado:

```
accesorios[2:5]
```

**Salida:**

```
['Piloto automático', 'Bancos de cuero', 'Aire acondicionado']
```

Si omitimos el primer índice ( `A[:j]` ) en el operador de *slicing*, la selección comienza al inicio de la secuencia. Si omitimos el segundo índice ( `A[i:]` ), la selección va hasta el final de la secuencia. Si omitimos ambos índices ( `A[:]` ), la selección es una copia de toda la secuencia.

El *slice* de abajo tiene inicio en el tercer elemento y va hasta el final de la secuencia.

```
accesorios[2:]
```

**Salida:**

```
['Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia']
```

El siguiente código selecciona desde el inicio de la secuencia hasta el elemento del índice 4.

```
accessorios[:5]
```

**Salida:**

```
['Llantas de aleación',
 'Cerraduras eléctricas',
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado']
```

**`A[i:j:k]`**

La diferencia para el método de *slicing* presentado en la sección anterior es que `A[i:j:k]` recorta la secuencia *A* del índice i hasta el j con pasos k. En este *slice*, el elemento con índice i es **incluido** y el elemento con índice j, **no es incluido** en el resultado.

Funciona de forma bien semejante a los argumentos de `range(start,stop,step)` . Vamos a ejemplificar este tipo de *slicing* con una lista creada a partir de un ***range***.

```
numeros = list(range(10))
numeros
```

**Salida:**

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

Suponga que queremos seleccionar apenas los elementos pares de la lista **`numeros`** . En este caso podemos utilizar el operador `[i:j:k]` para ayudarnos en esta tarea

```
numeros[0:10:2]
```

**Salida:**

```
[0, 2, 4, 6, 8]
```

La selección de arriba podría haber sido hecha omitiendo los dos primeros índices.

```
numeros[::2]
```

**Salida:**

```
[0, 2, 4, 6, 8]
```

Para seleccionar apenas los elementos impares basta informar que queremos comenzar la selección a partir del segundo elemento de la lista **`numeros`** .

```
numeros[1::2]
```

**Salida:**

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

## 2.3 EJERCICIOS

En *Data Science* utilizamos informaciones provenientes de diversas fuentes, Estas informaciones pueden estar disponibles en formatos variados: listas, tuplas, texto, tablas, JSON, etc. Por este motivo, aprender a manipular en todos los formatos es indispensable para el trabajo de un científico de datos.

Vamos a entrenar un poco los conocimientos que obtuvimos sobre secuencias en Python. Para responder las preguntas a seguir, considere el siguiente conjunto de informaciones:

```
carro_1 = ['Dodge Jorney', 'Motor 3.0 32v', 2010, 99197, False, ['Vidrios eléctricos', 'Piloto automá
tico', 'Techo  panorámico', 'Aire acondicionado'], 120716.27]
carro_2 = ['Carens', 'Motor 5.0 V8 Bi-Turbo', 2011, 37978, False, ['Aire acondicionado', 'Panel digit
```

```
al', 'Central multimedia', 'Cambio automático'], 76566.49]
carro_3 = ['Ford Edge', 'Motor Diesel V6', 2002, 12859, False, ['Sensor crepuscular', 'Llantas de ale
ación', 'Techo panorámico', 'Sensor de lluvia'], 71647.59]
```

Para cada vehículo las informaciones esta disponibles en el siguiente orden.

- Nombre del vehículo
- Motor
- Año de fabricación
- Kilometraje
- ¿Es cero kilómetro? (booleano)
- Accesorios
- Valor de mercado

1. Crea una secuencia con nombre **carros** que contenga las informaciones de los 3 vehículos de arriba.

   Observación: - Mantenga el siguiente orden: carro_1, carro_2 e carro_3. - La secuencia puede ser una lista o una tupla.

   **Respuesta:**

```
carro_1 = ['Dodge Jorney', 'Motor 3.0 32v', 2010, 99197, False, ['Vidrios eléctricos', 'Pilot
o automático', 'Techo panorámico', 'Aire acondicionado'], 120716.27]
carro_2 = ['Carens', 'Motor 5.0 V8 Bi-Turbo', 2011, 37978, False, ['Aire acondicionado', 'Pan
el digital', 'Central multimedia', 'Cambio automático'], 76566.49]
carro_3 = ['Ford Edge', 'Motor Diesel V6', 2002, 12859, False, ['Sensor crepuscular', 'Llanta
s de aleación', 'Techo panorámico', 'Sensor de lluvia'], 71647.59]
```

```
carros = carro_1, carro_2, carro_3
carros
```

   **Salida:**

```
(['Dodge Jorney',
'Motor 3.0 32v',
2010,
99197,
False,
['Vidrios eléctricos',
'Piloto automático',
'Techo panorâmico',
'Aire acondicionado'],
120716.27],
['Carens',
'Motor 5.0 V8 Bi-Turbo',
2011,
37978,
False,
['Aire acondicionado',
'Panel digital',
'Central multimedia',
'Cambio automático'],
76566.49],
['Ford Edge',
'Motor Diesel V6',
2002,
```

```
12859,
False,
['Sensor crepuscular',
'Llantas de aleación',
'Techo panorámico',
'Sensor de lluvia'],
71647.59])
```

2. Utilizando una de las operaciones con secuencias que aprendimos, verifique en la secuencia **`carros`** si el vehículo **Ford Edge** tiene como accesorio **Sensor de lluvia**

**Respuesta:**

```
'Sensor de lluvia' in carros[-1][-2]
```

**Salida:**

```
True
```

3. Obtenga los códigos que tienen como resultado el siguiente conjunto de selecciones en la secuencia **`carros`**

1ª) 2002

2ª) ['Vidrios eléctricos', 'Piloto automático', 'Techo panorámico', 'Aire acondicionado']

3ª) 'Cambio automático'

4ª) ['Panel digital', 'Central multimedia']

**Respuestas:**

```
# 1ª
carros[-1][2]
```

**Salida:**

```
2002
```

```
# 2ª
carros[0][-2]
```

**Salida:**

```
['Vidrios eléctricos', 'Piloto automático', 'Techo panorámico', 'Aire acondicionado']
```

```
# 3ª
carros[1][-2][-1]
```

**Salida:**

```
'Cambio automático'
```

```
# 4ª
carros[1][-2][1:3]
```

**Salida:**

```
['Panel digital', 'Central multimedia']
```

4. obtenga la sumatoria de los valores de mercado de los tres vehículos de la secuencia **`carros`** .

**Respuesta:**

```
carros[0][-1] + carros[1][-1] + carros[2][-1]
```

**Salida:**

```
268930.35
```

## 2.4 MÉTODOS DE SECUENCIAS

**Documentación**: https://docs.python.org/3.6/library/stdtypes.html#mutable-sequence-types

Los métodos mostrados en la tabla de abajo son soportados solamente por secuencias mutables. En la tabla la letra ***A*** representa una secuencia cualquiera, ***x*** es un objeto arbitrario que sigue las restricciones de tipo y valor impuesta por la secuencia ***A*** e ***i*** es un número entero.

| Operación | Resultado |
|---|---|
| `A.sort()` | Ordena la lista *A*. |
| `A.append(x)` | Agrega el elemento *x* al final de la lista *A*. |
| `A.insert(i, x)` | Inserta *x* en la secuencia *A* en el índice informado por *i*. |
| `A.remove(x)` | Elimina la primera ocurrencia de *x* de la secuencia *A*. |
| `A.pop([i])` | Elimina y retorna el elemento de índice *i* de la secuencia *A*. |
| `A.clear()` | Elimina todos los ítems de la secuencia *A*. |
| `A.copy()` | Crea una copia de la secuencia *A*. |

Para los próximos ejemplos, considere la lista **`accesorios`** de abajo.

```
accesorios = ['LLantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero',
'Aire acondicionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia']
```

### `A.sort()`

El método `sort()` ordena la secuencia *in-place* (sin crear un nuevo objeto).

```
accesorios.sort()
accesorios
```

**Salida:**

```
['Aire acondicionado',
 'Bancos de cuero',
 'Cerraduras eléctricas',
 'Llantas de aleación',
 'Piloto automático',
 'Sensor crepuscular',
 'Sensor de estacionamiento'
```

```
 'Sensor de lluvia']
```

El método `sort()` tiene un argumento **reverse** (*default false*) que si configurado como *True* coloca la lista en orden decreciente ( `A.sort(reverse=True)` ).

**`A.append(x)`**

Agrega el elemento pasado como argumento al final de la secuencia. Observe el ejemplo a seguir.

```
accesorios.append('4 X 4')
accesorios
```

**Salida:**

```
['Aire acondicionado',
 'Bancos de cuero',
 'Cerraduras eléctricas',
 'Llantas de aleación',
 'Piloto automático',
 'Sensor crepuscular',
 'Sensor de estacionamiento'
 'Sensor de lluvia',
 '4 X 4']
```

**`A.insert(i, x)`**

Inserta *x* en la secuencia *A* en la posición (índice) informada en el primer argumento (*i*).

```
accesorios.insert(2, 'Airbag')
accesorios
```

**Salida:**

```
['Aire acondicionado',
 'Bancos de cuero',
 'Airbag',                  <- Item insertado
 'Cerraduras eléctricas',
 'Llantas de aleación',
 'Piloto automático',
 'Sensor crepuscular',
 'Sensor de estacionamiento'
 'Sensor de lluvia',
 '4 X 4']
```

El el ejemplo anterior insertamos el accesorio "Airbag" en la posición de índice 2 de la lista **`accesorios`**

**`A.remove(x)`**

Elimina la primera ocurrencia de *x* de la secuencia *A*. Para ejemplificar vamos primero a insertar un accesorio repetido en nuestra lista **accesorios**.

```
accesorios.insert(8, 'Airbag')
accesorios
```

**Salida:**

```
['Aire acondicionado',
 'Bancos de cuero',
 'Airbag',
 'Cerraduras eléctricas',
 'Llantas de aleación',
 'Piloto automático',
 'Sensor crepuscular',
 'Sensor de estacionamiento'
 'Airbag',
 'Sensor de lluvia',
 '4 X 4']
```

Tenemos ahora dos accesorios "Airbag", uno en la posición 2 y otro en la posición de índice 8. Utilizando el método `remove()` y pasando como argumento "Airbag", la primera ocurrencia encontrada de este accesorio será eliminada.

```
accesorios.remove('Airbag')
accesorios
```

**Salida:**

```
['Aire acondicionado',
 'Bancos de cuero',
 'Cerraduras eléctricas',
 'Llantas de aleación',
 'Piloto automático',
 'Sensor crepuscular',
 'Sensor de estacionamiento'
 'Airbag',
 'Sensor de lluvia',
 '4 X 4']
```

**`A.pop(i)`**

Elimina y retorna el elemento de índice *i* de la secuencia *A*. Por *default* el método `pop()` sin el argumento *i*, elimina y retorna el último elemento de la secuencia.

```
accesorios.pop()
```

**Salida:**

```
'4 X 4'
```

Accediendo a la lista **`accesorios`** tenemos:

```
accesorios
```

**Salida:**

```
['Aire acondicionado',
 'Bancos de cuero',
 'Cerraduras eléctricas',
 'Llantas de aleación',
 'Piloto automático',
 'Sensor crepuscular',
 'Sensor de estacionamiento'
 'Airbag',
 'Sensor de lluvia']
```

Para eliminar un ítem en una posición específica basta pasar el índice del elemento como argumento.

```
accesorios.pop(3)
```

**Salida:**

```
'Llantas de aleación'
```

Visualizando nuevamente la lista **accesorios** tenemos:

```
accesorios
```

**Salida:**

```
['Aire acondicionado',
 'Bancos de cuero',
 'Llantas de aleación',
 'Piloto automático',
 'Sensor crepuscular',
 'Sensor de estacionamiento'
 'Airbag',
 'Sensor de lluvia']
```

### `A.clear()`

El método `clear()` limpia la secuencia eliminando todos los elementos.

```
accesorios.clear()
accesorios
```

**Salida:**

```
[]
```

### `A.copy()`

Para que podamos entender mejor el funcionamiento del método `copy()` vamos a recrear nuestra lista **accesorios**.

```
accesorios = ['LLantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero',
'Aire acondicionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia']
accesorios
```

**Salida:**

```
['Llantas de aleación',
 'Cerraduras eléctricas',
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia']
```

Como vimos en el Capítulo 1, cuando asignamos un determinado objeto a una variable, estamos apenas generando una referencia para este objeto. Para entender mejor veamos un ejemplo.

Suponga que queremos crear una copia de la lista **`accesorios`** para modificar esa copia en algunas pruebas y dejar la lista original intacta para otras tareas. De forma intuitiva haríamos una copia apenas asignando a una nueva variable, la lista **`accesorios`** .

```
accesorios_copia = accesorios
accesorios_copia
```

**Salida:**

```
['Llantas de aleación',
 'Cerraduras eléctricas',
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia']
```

En Python esta operación apenas crea una nueva referencia para el mismo objeto, o sea, **`accesorios`** y **`accesorios_copia`** , ahora se refieren al mismo objeto. Para confirmar esto basta modificar la lista **`accesorios_copia`** y consultar nuevamente la lista **`accesorios`** .

```
accesorios_copia.append('4 X 4')
accesorios_copia
```

**Salida:**

```
['Llantas de aleación',
 'Cerraduras eléctricas',
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia',
 '4 X 4']
```

Después de la modificación de **`accesorios_copia`** examinamos la lista original y constatamos que esta sufrió la misma alteración.

```
accesorios
```

**Salida:**

```
['Llantas de aleación',
 'Cerraduras eléctricas',
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia',
 '4 X 4']
```

Para resolver esto utilizamos el método `copy()` que crea una copia de la secuencia y no una nueva referencia para una misma secuencia.

```
accesorios_copia = accesorios.copy()
accesorios_copia
```

**Salida:**

```
 ['Llantas de aleación',
 'Cerraduras eléctricas',
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia',
 '4 X 4']
```

Eliminando el accesorio "4 X 4" de la lista original.

```
accesorios.pop()
accesorios
```

**Salida:**

```
['Llantas de aleación',
 'Cerraduras eléctricas',
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia']
```

Podemos verificar ahora que la lista **`accesorios`** fue modificada y la lista **`accesorios_copia`** todavía mantiene el accesorio "4 X 4", indicando que ahora se tratan de objetos distintos.

```
accesorios_copia
```

**Salida:**

```
 ['Llantas de aleación',
 'Cerraduras eléctricas',
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia',
 '4 X 4']
```

## 2.5 EJERCICIOS

En *Data Science* es indispensable saber manipular y tratar los diversos tipos de datos en que están disponibles las informaciones. Para eso el científico de datos debe conocer y dominar el mayor número posible de métodos y operaciones disponibles para manipulación de los diversos formatos existentes.

Considere el código de celdas de abajo para resolver las cuestiones. Para cada vehículo las informaciones están puestas en el siguiente orden:

- Nombre del vehículo
- Motorización
- Año de fabricación
- Kilometraje
- ¿Es cero kilómetro? (booleano)
- Accesorios
- Valor de mercado

```
carro_1 = ['Dodge Jorney', 'Motor 3.0 32v', 2010, 99197, False, ['Vidrios eléctricos', 'Piloto automá
tico', 'Techo panorámico', 'Aire acondicionado'], 120716.27]
carro_2 = ['Carens', 'Motor 5.0 V8 Bi-Turbo', 2011, 37978, False, ['Aire acondicionado', 'Panel digit
al', 'Central multimedia', 'Cambio automático'], 76566.49]
carro_3 = ['Ford Edge', 'Motor Diesel V6', 2002, 12859, False, ['Sensor crepuscular', 'Llantas de ale
ación', 'Techo panorámico', 'Sensor de lluvia'], 71647.59]

carros = [carro_1, carro_2, carro_3]
```

1. Seleccione la lista de accesorios de cada vehículo y crea una nueva lista con la concatenación de los accesorios de los tres vehículos. Llame esta lista de **`accesorios`** .

   **Respuesta:**

   ```
   accesorios = carros[0][-2] + carros[1][-2] + carros[2][-2]
   accesorios
   ```

   **Salida:**

   ```
   ['Vidrios eléctricos',
   'Piloto automático',
   'Techo panorámico',
   'Aire acondicionado',
   'Aire acondicionado',
   'Panel digital',
   'Central multimídia',
   'Cambio automático',
   'Sensor crepuscular',
   'LLantas de aleación',
   'Techo panorámico',
   'Sensor de lluvia']
   ```

2. Coloque los ítems de la lista creada arriba en orden alfabética

   Observación: - Recuerda que los métodos de secuencias funcionan apenas con secuencias mutables.

   **Respuesta:**

   ```
   accesorios.sort()
   accesorios
   ```

   **Salida:**

   ```
   ['Aire condicionado',
   'Aire condicionado',
   'Central multimedia',
   'Cambio automático',
   ```

```
'Llantas de aleación',
'Panel digital',
'Piloto automático',
'Sensor crepuscular',
'Sensor de lluvia',
'Techo panorámico',
'Techo panorámico',
'Vidrios eléctricos']
```

3. Note que los ítems **'Aire acondicionado'** y **'Techo panorámico'** están duplicados en nuestra lista de accesorios. Elimina el ítem **'Aire acondicionado'** con el uso del método ***remove***.

   **Resposta:**

   ```
   accesorios.remove('Aire acondicionado')
   accesorios
   ```

   **Salida:**

   ```
   ['Aire acondicionado',
   'Central multimídia',
   'Cambio automático',
   'Llantas de aleación',
   'Panel digital',
   'Piloto automático',
   'Sensor crepuscular',
   'Sensor de lluvia',
   'Techo panorámico',
   'Techo panorámico',
   'Vidrios eléctricos']
   ```

4. Elimine el ítem **'Techo panorámico'** con el uso del método ***pop***.

   **Respuesta:**

   ```
   accesorios.pop(-2)
   accesorios
   ```

   **Salida:**

   ```
   ['Aire acondicionado',
   'Central multimídia',
   'Cambio automático',
   'Llantas de aleación',
   'Panel digital',
   'Piloto automático',
   'Sensor crepuscular',
   'Sensor de lluvia',
   'Techo panorámico',
   'Vidrios eléctricos']
   ```

5. Agrega el ítem **'Bancos de cuero'** en nuestra lista de accesorios en la segunda posición

   **Respuesta:**

   ```
   accesorios.insert(1, 'Bancos de cuero')
   accesorios
   ```

   **Salida:**

```
['Aire acondicionado',
 'Bancos de cuero',
 'Central multimedia',
 'Cambio automático',
 'Llantas de aleación',
 'Panel digital',
 'Piloto automático',
 'Sensor crepuscular',
 'Sensor de lluvia',
 'Techo panorámico',
 'Vidrios eléctricos']
```

6. Agregue al final de la lista de accesorios el ítem **'Cámara de estacionamiento'**.

**Respuesta:**

```
accessorios.append('Cámara de estacionamiento')
accessorios
```

**Salida:**

['Aire acondicionado',

```
 'Bancos de cuero',
 'Central multimedia',
 'Cambio automático',
 'Llantas de aleación',
 'Panel digital',
 'Piloto automático',
 'Sensor crepuscular',
 'Sensor de lluvia',
 'Techo panorámico',
 'Cámara de estacionamiento']
```

## 2.6 PARA SABER MÁS...

### Desempaquetado de tuplas

En Python podemos asignar variables distintas a cada elemento de una tupla utilizando un recurso bastante simple. Observe el ejemplo donde creamos la tupla **`carros`**

```
carros = ('C4', 'Fit', 'Focus')
carros
```

**Salida:**

```
('C4', 'Fit', 'Focus')
```

Para colocar cada nombre de vehículo de la tupla carros en una variable distinta basta utilizar el recurso de desempaquetado de tuplas de la siguiente forma:

```
carro_1, carro_2, carro_3 = carros
```

Verificando el valor de cada variable:

```
print(carro_1)
```

```
print(carro_2)
print(carro_3)
```

**Salida:**

```
'C4'
'Fit'
'Focus'
```

Este recurso es bastante útil en la iteración de secuencias (Capítulo 4) y en la construcción y utilización de funciones que retorna múltiples valores (Capítulo 5).

### `zip()`

**Documentación**: https://docs.python.org/3.6/library/functions.html#zip

La función `zip()` es una herramienta bastante útil del lenguaje Python pues retorna un iterador de tuplas, donde la i-ésima tupla contiene el i-ésimo elemento de cada una de las secuencias de argumentos o iterables. Veamos un ejemplo para entender mejor.

Considere las dos listas abajo:

```
carros = ['C4', 'Fit', 'Focus']
fabricantes = ['Citroën', 'Honda', 'Ford']
```

Pasando las lista **`carros`** y **`fabricantes`** como argumentos para la función `zip()` creamos un iterador que agrega los elementos de cada una de las listas.

```
zip(carros, fabricantes)
```

**Salida:**

```
<zip at 0x22ee0240d48>
```

Materializando este iterador con el uso de la función `list()` podemos entender mejor el procedimiento realizado por la función.

```
list(zip(carros, fabricantes))
```

**Salida:**

```
[('C4', 'Citroën'), ('Fit', 'Honda'), ('Focus', 'Ford')]
```

Note que los elementos de cada lista fueron agrupados en una secuencia de tuplas con pares de valor. Este recurso es bastante útil en procesos de iteración (Capítulo 4) y también en la construcción de diccionarios Python (Capítulo 3).

### `set()`

**Documentación**: https://docs.python.org/3.6/library/stdtypes.html#types-set

El lenguaje Python también permite la creación de conjuntos con el uso de la función `set()` . Los conjuntos son colecciones de elementos únicos y no ordenados. El siguiente código crea una tupla para

ayudarnos en un ejemplo sobre conjuntos.

```
carros = ('C4', 'Fit', 'Focus', 'Gol', 'Focus', 'C4', 'Up!')
carros
```

**Salida:**

```
('C4', 'Fit', 'Focus', 'Gol', 'Focus', 'C4', 'Up!')
```

Existen dos formas de crear un conjunto. La primera es con el uso de función `set()` y el segundo utilizando las llaves `{}` .

Un uso bastante común de los conjuntos es para la eliminación de valores duplicados. Observe lo que pasa con la tupla **`carros`** cuando la utilizamos para crear un conjunto.

```
set(carros)
```

**Salida:**

```
{'C4', 'Fit', 'Gol', 'Focus', 'Up!'}
```

Note que los valores repetidos fueron eliminados, restando apenas los valores distintos.

## 2.7 EJERCICIOS

En el último ejercicios hicimos una concatenación de accesorios de tres vehículos de nuestro *dataset* y después identificamos y retiramos los ítems duplicados de esta lista.

1. Realiza esta misma tarea considerando que ahora conocemos el tipo `set` . Los dados están en las celdas de abajo y para cada vehículo las informaciones están disponibles en el siguiente orden:

   - Nombre del vehículo
   - Motorización
   - Año de fabricación
   - Kilometraje
   - ¿Es cero kilómetro?
   - Accesorios
   - Valor de mercado

   ```
   carro_1 = ['Dodge Jorney', 'Motor 3.0 32v', 2010, 99197, False, ['Vidrios eléctricos', 'Piloto automático', 'Techo panorámico', 'Aire acondicionado'], 120716.27]
   carro_2 = ['Carens', 'Motor 5.0 V8 Bi-Turbo', 2011, 37978, False, ['Aire acondicionado', 'Panel digital', 'Central multimedia', 'Cambio automático'], 76566.49]
   carro_3 = ['Ford Edge', 'Motor Diesel V6', 2002, 12859, False, ['Sensor crepuscular', 'Llantas de aleación', 'Techo panorámico', 'Sensor de lluvia'], 71647.59]

   carros = [carro_1, carro_2, carro_3]
   ```

   **Respuesta:**

```
accesorios = carros[0][-2] + carros[1][-2] + carros[2][-2]
accesorios
```

**Salida:**

['Vidrios eléctricos', 'Piloto automático', 'Techo panorámico', 'Aire acondicionado', 'Aire acondicionado', 'Panel digital', 'Central multimedia', 'Cambio automático', 'Sensor crepuscular', 'LLantas de aleación', 'Techo panorámico', 'Sensor de lluvia']

```
set(accesorios)
```

**Salida:**

{'Aire acondicionado', 'Cambio automático', 'Central multimedia', 'Llantas de aleación', 'Panel digital', 'Piloto automático', 'Sensor crepuscular', 'Sensor de lluvia', 'Techo panorámico', 'Vidrios eléctricos'}

```
accesorios = list(set(accesorios))
accesorios
```

**Salida:**

['Sensor de lluvia', 'Llantas de aleación', 'Panel digital', 'Cambio automático', 'Central multimedia', 'Piloto automático', 'Aire acondicionado', 'Sensor crepuscular', 'Techo panorámico', 'Vidrios eléctricos']

CAPÍTULO 3

# TRABAJANDO CON DICCIONARIOS

Listas son colecciones secuenciales, o sea, los ítems de estas secuencias están ordenados y utilizan índices (números enteros) para acceder a los valores.

Los diccionarios son colecciones un poco diferentes. Son estructuras de datos que representan un tipo de mapeo, que pueden ser entendidos como colecciones de asociaciones entre pares de valores, donde el primer elemento del par es conocido como clave (*key*) y el segundo como valor (*value*). Cada clave está asociada a un único valor y esa asociación recibe el nombre de pareja clave-valor (*key-value*).

## 3.1 CREANDO DICCIONARIOS

**Documentación**: https://docs.python.org/3.6/library/stdtypes.html#typesmapping

Un diccionario puede ser creado de varias formas:

```
diccionario = {'clave_1': valor_1, ..., 'clave_n': valor_n}
diccionario = dict(clave_1=valor_1, ..., clave_n=valor_n)
diccionario = dict(zip(iterable_1, iterable_2))
```

Vamos a comenzar con un diccionario simple para ejemplificar. Suponga que queremos crear un mapeo donde tenemos como clave los nombres de los vehículos de nuestro *dataset* y como valores los precios de venta de cada vehículo.

Para crear este mapeo podemos utilizar las {} que son los delimitadores de un diccionario y dentro de las {} podemos ir listando los pares de clave-valor, como en el código a seguir:

```
datos = {'Jetta Variant': 88078.64, 'Passat': 106161.94, 'Crossfox': 72832.16}
datos
```

**Salida:**

```
{'Jetta Variant': 88078.64, 'Passat': 106161.94, 'Crossfox': 72832.16}
```

Una información importante sobre las claves de un diccionario es que pueden ser creadas con cualquier tipo inmutable (string, tuplas, etc).

Verificando el tipo de datos de la variable **`datos`** , que creamos en el código anterior, tenemos como respuesta que se trata de un `dict` , o sea, un diccionario de Python

```
type(datos)
```

**Salida:**

```
dict
```

También podemos crear un diccionario utilizando la función `dict()`, y con esta función podemos proceder de formas variadas. Observe el ejemplo de abajo con la creación de un diccionario semejante al creado anteriormente.

```
datos = dict(JettaVariant = 88078.64, Passat = 106161.94, Crossfox = 72832.16)
datos
```

**Salida:**

```
{'JettaVariant': 88078.64, 'Passat': 106161.94, 'Crossfox': 72832.16}
```

Aquí declaramos los pares clave-valor como si fuesen asignación de variable. Note que en este método precisamos que las claves obedezcan las restricciones de nombres de variable del lenguaje Python (Capítulo 1).

Otra forma de crear diccionarios en Python es con la combinación entre las funciones `dict()` y `zip()` que conocimos en el capítulo anterior. Este método de creación nos permite transformar pares de listas en diccionarios de forma bastante simple. Considere las dos siguientes listas para nuestro ejemplo:

```
carros = ['Jetta Variant', 'Passat', 'Crossfox']
valores = [88078.64, 106161.94, 72832.16]
```

Utilizando estas dos listas como argumentos de la función `zip()` tenemos como resultado un iterador que agrega los elementos de cada una de las listas en tuplas. Abajo utilizamos la función `list()` apenas para visualizar el resultado.

```
list(zip(carros, valores))
```

**Salida:**

```
[('Jetta Variant', 88078.64), ('Passat', 106161.94), ('Crossfox', 72832.16)]
```

Pasando este iterador como argumento para la función `dict()` tenemos como respuesta un diccionario que utiliza como claves el primer elemento de cada tupla y como valores el segundo elemento de cada tupla.

```
datos = dict(zip(carros, valores))
datos
```

**Salida:**

```
{'Jetta Variant': 88078.64, 'Passat': 106161.94, 'Crossfox': 72832.16}
```

## 3.2 OPERACIONES CON DICCIONARIOS

**Documentación**: https://docs.python.org/3.6/library/stdtypes.html#mapping-types-dict

Así como las secuencias, los diccionarios también soportan un conjunto de operaciones que están listados en la tabla de abajo. En la tabla la letra ***d*** representa un diccionario cualquiera, ***key*** es la clave de un ítem especifico del diccionario y puede ser de cualquier tipo inmutable y ***valor*** es el valor de un ítem especifico del diccionario.

| Operaciones | Resultado |
|---|---|
| `d[key]` | Retorna el ítem del diccionario *d* correspondiente a la clave *key*. |
| `key in d` | Retorna *True* si el diccionario *d* tiene un ítem con la clave *key*. |
| `len(d)` | Retorna el número de ítems en el diccionario *d*. |
| `d[key] = value` | Asigna el valor *value* a la clave *key* del diccionario *d*. |
| `del d[key]` | Elimina el ítem con la clave *key* del diccionario *d*. |

Para los ejemplos sobre operaciones con diccionarios vamos a utilizar el diccionario **datos** que creamos en la sección anterior.

```
datos
```

**Salida:**

```
{'Jetta Variant': 88078.64, 'Passat': 106161.94, 'Crossfox': 72832.16}
```

## 3.3 D[KEY]

Cuando trabajamos con la secuencia y queremos acceder a un determinado ítem de una lista, por ejemplo, necesitamos saber la posición de este elemento dentro de la lista, o mejor, necesitamos saber su índice. Con diccionarios eso no es necesario.

Para acceder a un valor específico de un diccionario *d* basta informar para el operador [] la clave (*key*) del ítem.

En el código a seguir estamos accediendo el valor de venta del vehículo "Passat"

```
datos['Passat']
```

**Salida:**

```
106161.94
```

**`key() in d`**

El operador `in` también funciona con diccionarios cuando queremos verificar la existencia de una determinada clave (*key*). La línea del código `key in d` retorna *True* si la clave (*key*) se encuentra en el diccionario *d*.

```
'Passat' in datos
```

**Salida:**

```
True
```

La palabra clave `not` también puede ser utilizado para hacer verificación contraria

```
'Volkswagen' not in datos
```

**Salida:**

```
True
```

**`len(d)`**

La función `len(d)` retorna el número de ítems del diccionario, o sea, la cantidad de pares clave-valor.

```
len(datos)
```

**Salida:**

```
3
```

## 3.4 D[KEY]=VALUE

Para incluir un nuevo clave-valor en el diccionario podemos proceder de la siguiente forma:

```
datos['DS5'] = 124549.07
```

En el código de encima informamos la nueva clave y atribuimos a ella un valor. Caso la clave ya exista en el diccionario este procedimiento irá sustituir el valor de esta clave por la nueva asignación.

```
datos
```

**Salida:**

```
{'Jetta Variant': 88078.64,
 'Passat': 106161.94,
 'Crossfox': 72832.16,
 'DS5': 124549.07}
```

**`del d[key]`**

Para excluir un ítem de un diccionario podemos utilizar el comando `del` y luego especificar el diccionario y la clave que deseamos eliminar.

```
datos
```

**Salida:**

```
{'Jetta Variant': 88078.64,
 'Passat': 106161.94,
 'Crossfox': 72832.16,
 'DS5': 124549.07}
```

La instrucción `del` debe ser utilizada con cuidado, pues si no especificamos la clave que será eliminada, todo el diccionario será excluido.

```
del datos['Passat']
datos
```

**Salida:**

```
{'Jetta Variant': 88078.64, 'Crossfox': 72832.16, 'DS5': 124549.07}
```

## 3.5 EJERCICIOS

Un diccionario es una estructura de datos muy importante en el lenguaje Python. Son colecciones de pares clave-valor que representa un tipo de mapeo.

Aprendimos hasta ahora las formas de crear diccionarios y algunas operaciones básicas. Vamos a entrenar un poco este conocimiento con un diccionario un poco más elaborado.

Observe que el diccionario de abajo es un ejemplo de diccionario anidado, o sea, diccionario que tiene como valores otros diccionarios.

```
dataset = {
    'Lamborghini': {
        'motor': 'Motor V12',
        'año': 2018,
        'km': 25400,
        'accesorios': ['Llantas de aleación', 'Cambio automático', 'Cerraduras eléctricas'],
        'valor': 133529.84
    },
    'Dodge': {
        'motor': 'Motor 6.7',
        'año': 2017,
        'km': 45757,
        'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia'],
        'valor': 92612.1
    },
    'Pajero': {
        'motor': 'Motor 2.4 Turbo',
        'Cero_km': True,
        'km': 80000,
        'accesorios': ['Control de tracción', 'Bancos de cuero', 'Control de estabilidad'],
        'valor': 51606.59
    }
}
```

1. Seleccione el diccionario que contiene los datos del vehículo **Dodge**.

    **Respuesta:**

    ```
     dataset['Dodge']
    ```

    **Salida:**

    ```
     {'motor': 'Motor 6.7',
     'año': 2017,
    ```

```
'km': 45757,
'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia'],
'valor': 92612.1}
```

2. Seleccione ahora solamente los accesorios del vehículo **Dodge**.

   **Respuesta:**

   ```
   dataset['Dodge']['accesorios']
   ```

   **Salida:**

   ```
   ['Llantas de aleación', '4 X 4', 'Central multimedia']
   ```

3. Verifique si el diccionario de datos del vehículo **Pajero** posee la clave **año**.

   **Respuesta:**

   ```
   'año' in dataset['Pajero']
   ```

   **Salida:**

   ```
   False
   ```

4. Agregue la clave **año** al diccionario de datos del vehículo **Pajero** con el valor de 2012. Visualice el diccionario nuevamente.

   **Respuesta:**

   ```
   dataset['Pajero']['año'] = 2012
   dataset
   ```

   **Salida:**

   ```
   {'Lamborghini': {'motor': 'Motor V12',
   'año': 2018,
   'km': 25400,
   'accesorios': ['Llantas de aleación', 'Cambio automático', 'Cerraduras eléctricas'],
   'valor': 133529.84},
   'Dodge': {'motor': 'Motor 6.7',
   'año': 2017,
   'km': 45757,
   'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia'],
   'valor': 92612.1},
   'Pajero': {'motor': 'Motor 2.4 Turbo',
   'Cero_km': True,
   'km': 80000,
   'accesorios': ['Control de tracción',
   'Bancos de cuero',
   'Control de estabilidad'],
   'valor': 51606.59,
   'año': 2012}}
   ```

5. Elimina la clave **Cero_km** del diccionario de datos del vehículo **Pajero** para mantener la compatibilidad con las informaciones de los demás vehículos. Visualice el diccionario nuevamente.

   **Respuesta:**

```
del dataset['Pajero']['Cero_km']
dataset
```

**Salida:**

```
{'Lamborghini': {'motor': 'Motor V12',
'año': 2018,
'km': 25400,
'accesorios': ['Llantas de aleación', 'Cambio automático', 'Cerraduras eléctricas'],
'valor': 133529.84},
'Dodge': {'motor': 'Motor 6.7',
'año': 2017,
'km': 45757,
'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia'],
'valor': 92612.1},
'Pajero': {'motor': 'Motor 2.4 Turbo',
'km': 80000,
'accesorios': ['Control de tracción',
'Bancos de cuero',
'Control de estabilidad'],
'valor': 51606.59,
'año': 2012}}
```

1. Agregue un nuevo ítem al diccionario ***dataset***. Abajo tenemos las informaciones para el vehículo **Ford Edge**:

```
claves = ['motor', 'año', 'km', 'accesorios', 'valor']
valores = ['Motor Diesel V6', 2002, 128590, ['Sensor crepuscular', 'Sensor de lluvia', 'Aire acon
dicionado'], 71647.59]
```

Utilice las funciones ***dict*** y ***zip*** para realizar la tarea.

**Respuesta:**

```
claves = ['motor', 'año', 'km', 'accesorios', 'valor']
valores = ['Motor Diesel V6', 2002, 128590, ['Sensor crepuscular', 'Sensor de lluvia', 'Aire acon
dicionado'], 71647.59]

dataset['Ford Edge'] = dict(zip(claves, valores))
dataset
```

**Salida:**

```
{'Lamborghini': {'motor': 'Motor V12',
'año': 2018,
'km': 25400,
'accesorios': ['Llantas de aleación', 'Cambio automático', 'Cerraduras eléctricas'],
'valor': 133529.84},
'Dodge': {'motor': 'Motor 6.7',
'año': 2017,
'km': 45757,
'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia'],
'valor': 92612.1},
'Pajero': {'motor': 'Motor 2.4 Turbo',
'km': 80000,
'accesorios': ['Control de tracción',
'Bancos de cuero',
'Control de estabilidad'],
'valor': 51606.59,
'año': 2012}}
```

```
    'Ford Edge': {'motor': 'Motor Diesel V6',
    'año': 2002,
    'km': 128590,
    'accesorios': ['Sensor crepuscular', 'Sensor de lluvia', 'Aire acondicionado'],
    'valor': 71647.59}}
```

## 3.6 MÉTODOS DE DICCIONARIOS

**Documentación**: https://docs.python.org/3.6/library/stdtypes.html#mapping-types-dict

La siguiente tabla muestra algunos métodos soportados por diccionarios Python. En la tabla la letra ***d*** representa un diccionario cualquiera, ***key*** es la clave de un ítem especifico del diccionario y puede ser cualquier tipo inmutable y ***value*** es un valor de un ítem especifico del diccionario.

| Operaciones | Resultado |
|---|---|
| `d.clear()` | Elimina todos los ítems del diccionario. |
| `d.copy()` | Crea una copia del diccionario. |
| `d.pop(key[, default])` | Si la clave (*key*) es encontrada en el diccionario, el ítem es eliminado y su valor es retornado. Caso contrario, el valor especificado como *default* es retornado. Si el valor *default* no es pasado y la clave no fue encontrada en el diccionario un error será generado. |
| `d.get(key[, default])` | Retorna el valor correspondiente a la clave *key*. Caso contrario, el valor especificado como *default* es retornado. Si el valor *default* no fuera pasado y la clave no fuera encontrada el método retorna *None*, o sea, este método nunca retorna un error. |
| `d.update()` | Actualiza el diccionario. |

Para los ejemplos de abajo vamos a continuar con nuestro diccionario **`datos`** que fue creado y modificado en las secciones anteriores.

```
datos
```

**Salida:**

```
{'Jetta Variant': 88078.64, 'Crossfox': 72832.16, 'DS5': 124549.07}
```

**`d.clear()`**

Para eliminar todos los ítems de un diccionario utilizamos el método `clear()`.

```
datos.clear()
```

En procesos iterativos, métodos como `clear()` son bastante útiles, permitiendo operaciones de acumulo y limpieza utilizando apenas un objeto.

```
datos
```

**Salida:**

```
{}
```

**`d.copy()`**

Funciona de la misma forma que el método `copy()` que aprendimos en el capítulo sobre secuencias, o sea, crea una copia del diccionario.

La primera línea del código de abajo apenas recrea el diccionario **`datos`** que fue limpiado en la sección anterior.

```
datos = {'Crossfox': 72832.16, 'DS5': 124549.07, 'Jetta Variant': 88078.64}
datos_copia = datos.copy()
```

En el final de este capítulo hablaremos un poco más sobre copias de objetos en Python.

```
datos_copia
```

**Salida:**

```
{'Crossfox': 72832.16, 'DS5': 124549.07, 'Jetta Variant': 88078.64}
```

### `d.pop(key[, default])`

El método `pop()` elimina del diccionario el ítem especificado por la clave (*key*). Si la clave fuera encontrada, el ítem es eliminado y su valor es retornado. Caso contrario es retornado el valor especificado como *default*.

```
datos_copia.pop('DS5')
```

**Salida:**

```
124549.07
```

```
datos_copia
```

**Salida:**

```
{'Crossfox': 72832.16, 'Jetta Variant': 88078.64}
```

Si el valor *default* no es pasado y la clave no es encontrada en el diccionario, será generado un error

```
datos_copia.pop('Passat')
```

**Salida:**

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-75-c9e3566c9c39> in <module>
----> 1 dados_copia.pop('Passat')

KeyError: 'Passat'
```

Para solucionarlo basta pasar un valor de retorno, luego del nombre de la clave, para cuando esta no sea localizada.

```
datos_copia.pop('Passat', 'clave no encontrada.')
```

**Salida:**

```
'clave no encontrada.'
```

### `d.get(key[, default])`

El método `d.get(key[, default])` retorna el valor correspondiente a la clave *key*. Caso contrario, el valor especificado como *default* es retornado.

```
datos.get('DS5')
```

**Salida:**

```
124549.07
```

Si el valor *default* no es pasado y la clave no fuera encontrada,el método retorna *None*,o sea, este método nunca retorna un error.

```
datos.get('DS5x')
```

Por más que no retorne un error podemos definir un valor de retorno caso la llave no sea encontrada.

```
datos.get('DS5x', 'clave no encontrada.')
```

**Salida:**

```
'clave no encontrada.'
```

### `d.update()`

El método **update** permite actualizar total y parcialmente un diccionario.

Podemos agregar un nuevo ítem:

```
datos.update({'Passat': 106161.94})
datos
```

**Salida:**

```
{'Crossfox': 72832.16,
 'DS5': 124549.07,
 'Jetta Variant': 88078.64,
 'Passat': 106161.94}
```

Actualizar un ítem existente y al mismo tiempo agregar un nuevo ítem:

```
datos.update({'Passat': 96161.95, 'Volkswagen': 50000})
datos
```

**Salida:**

```
{'Crossfox': 72832.16,
 'DS5': 124549.07,
 'Jetta Variant': 88078.64,
 'Passat': 96161.95,
 'Volkswagen': 50000}
```

Podemos también realizar estas actualizaciones e inclusiones utilizando el formato de asignación,

como en el ejemplo abajo.

```
datos.update(Passat = 25000, Volkswagen = 50000)
datos
```

**Salida:**

```
{'Crossfox': 72832.16,
 'DS5': 124549.07,
 'Jetta Variant': 88078.64,
 'Passat': 25000,
 'Volkswagen': 50000}
```

Recordando que para este tipo de formato debemos obedecer las reglas de formación de nombre de variables del lenguaje Python.

## 3.7 EJERCICIOS

Ahora vamos a mezclar un poco los conocimientos adquiridos sobre secuencias y diccionarios. Como vimos en nuestro último ejercicio, es posible tener secuencias con valores en un diccionario, pero también es posible tener diccionarios con ítems de secuencias. Sabiendo eso y considerando el diccionario **`dataset`** de abajo, desarrolle los ejercicios.

```
dataset = {
    'Dodge': {
        'motor': 'Motor 6.7',
        'año': 2017,
        'km': 45757,
        'accesorios': ['LLantas de aleación', '4 X 4', 'Central multimedia'],
        'valor': 92612.1
    },
    'Pajero': {
        'motor': 'Motor 2.4 Turbo',
        'Cero_km': True,
        'km': 80000,
        'accesorios': ['Control de tracción', 'Bancos de cuero', 'Control de estabilidad'],
        'valor': 51606.59
    }
}
```

1. Crea una copia del diccionario **`dataset`** y llámalo de **`dataset_copia`** .

**Respuesta:**

```
dataset_copia = dataset.copy()
dataset_copia
```

**Salida:**

```
{'Dodge': {'motor': 'Motor 6.7',
'año': 2017,
'km': 45757,
'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia'],
'valor': 92612.1},
'Pajero': {'motor': 'Motor 2.4 Turbo',
'Cero_km': True,
```

```
'km': 80000,
'accesorios': ['Control de tracción',
'Bancos de cuero',
'Control de estabilidad'],
'valor': 51606.59}}
```

2. Utilizando el diccionario **`dataset_copia`** , agrega el ítem **Airbag** a los accesorios del vehículo **Dodge**. Observa que estamos trabajando con una lista dentro de un diccionario.

   **Respuesta:**

   ```
   dataset_copia['Dodge']['accesorios'].append('Airbag')
   dataset_copia
   ```

   **Salida:**

   ```
   {'Dodge': {'motor': 'Motor 6.7',
   'año': 2017,
   'km': 45757,
   'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia', 'Airbag'],
   'valor': 92612.1},
   'Pajero': {'motor': 'Motor 2.4 Turbo',
   'Cero_km': True,
   'km': 80000,
   'accesorios': ['Control de tracción',
   'Bancos de cuero',
   'Control de estabilidad'],
   'valor': 51606.59}}
   ```

3. Utilizando el método ***get***, obtén una visualización del campo **accesorios** del vehículo **Dodge** de los diccionarios **`dataset`** y **`dataset_copia`** . ¿Notas alguna inconsistencia con lo que aprendimos hasta ahora?

   **Respuesta:**

   ```
   dataset_copia['Dodge'].get('accesorios', 'clave no encontrada.')
   ```

   **Salida:**

   ```
   ['Llantas de aleación', '4 X 4', 'Central multimedia', 'Airbag']
   ```

   ```
   dataset['Dodge'].get('accesorios', 'clave no encontrada.')
   ```

   **Salida:**

   ```
   ['Llantas de aleación', '4 X 4', 'Central multimedia', 'Airbag']
   ```

4. Actualiza los diccionarios de informaciones de los vehículos **Dodge** y **Pajero**. Para el vehículo **Dodge** el **año** debe ser alterado para **2016** y para el vehículo **Pajero** el **motor** debe ser alterado para **Motor Diesel V8**. Utiliza el método `update()` para esta tarea.

   **Respuesta:**

   ```
   # Forma ERRADA de hacer la actualización
   dataset_copia.update({'Dodge': {'año': 2016}, 'Pajero': {'motor': 'Motor Diesel V8'}})
   dataset_copia
   ```

**Salida:**

```
{'Dodge': {'año': 2016}, 'Pajero': {'motor': 'Motor Diesel V8'}}
```

```
dataset
```

**Salida:**

```
{'Dodge': {'motor': 'Motor 6.7',
'año': 2017,
'km': 45757,
'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia'],
'valor': 92612.1},
'Pajero': {'motor': 'Motor 2.4 Turbo',
'Cero_km': True,
'km': 80000,
'accesorios': ['Control de tracción',
'Bancos de cuero',
'Control de estabilidad'],
'valor': 51606.59}}
```

```
dataset['Dodge'].update({'año': 2016})
dataset['Pajero'].update(motor = 'Motor Diesel V8')
dataset
```

**Salida:**

```
{'Dodge': {'motor': 'Motor 6.7',
'año': 2016,
'km': 45757,
'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia', 'Airbag'],
'valor': 92612.1},
'Pajero': {'motor': 'Motor Diesel V8',
'Zero_km': True,
'km': 80000,
'accesorios': ['Control de tracción',
'Bancos de couro',
'Control de estabilidad'],
'valor': 51606.59}}
```

## 3.8 PARA SABER MÁS...

### `deepcopy()`

**Documentación**: https://docs.python.org/3.6/library/copy.html

Vimos en este capítulo y en el anterior que para crear una copia de una secuencia o diccionario en Python necesitamos utilizar el método `copy()`.

El problema en la utilización de este método se da a la hora de trabajar con objetos compuestos (anidados) como es el caso del diccionario **`dataset`** declarado en el código de abajo. Este es un diccionario que tiene como valores otros dos diccionarios que por su vez tienen como valores listas.

```
dataset = {
    'Dodge': {
        'motor': 'Motor 6.7',
        'año': 2017,
```

```
        'km': 45757,
        'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia'],
        'valor': 92612.1
    },
    'Pajero': {
        'motor': 'Motor 2.4 Turbo',
        'Cero_km': True,
        'km': 80000,
        'accesorios': ['Control de tracción', 'Bancos de cuero', 'Control de estabilidad'],
        'valor': 51606.59
    }
}
```

Vimos anteriormente que para crear una copia basta utilizar el método `copy()` y a partir de allí podríamos realizar alteraciones en el objeto resultante sin afectar el original. A continuación el procedimiento:

```
dataset_copia = dataset.copy()
```

El siguiente código altera el diccionario **`dataset_copia`** agregando el accesorio '**Airbag**' a la lista de accesorios del vehículo '**Dodge**'.

```
dataset_copia['Dodge']['accesorios'].append('Airbag')
dataset_copia
```

**Salida:**

```
{
    'Dodge': {
        'motor': 'Motor 6.7',
        'año': 2017,
        'km': 45757,
        'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia', 'Airbag'],
        'valor': 92612.1
    },
    'Pajero': {
        'motor': 'Motor 2.4 Turbo',
        'Cero_km': True,
        'km': 80000,
        'accesorios': ['Control de tracción', 'Bancos de cuero', 'Control de estabilidad'],
        'valor': 51606.59
    }
}
```

Hasta aquí no tenemos novedades, pero cuando consultamos el diccionario **`dataset`** verificamos que la alteración se refleja en este diccionario también, o sea, el método `copy()` parece no funcionar para objetos compuestos.

```
dataset
```

**Salida:**

```
{
    'Dodge': {
        'motor': 'Motor 6.7',
        'año': 2017,
        'km': 45757,
        'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia', 'Airbag'],
```

```
        'valor': 92612.1
    },
    'Pajero': {
        'motor': 'Motor 2.4 Turbo',
        'Cero_km': True,
        'km': 80000,
        'accesorios': ['Control de tracción', 'Bancos de cuero', 'Control de estabilidad'],
        'valor': 51606.59
    }
}
```

Como vimos en el capítulo anterior las instrucciones de atribución en Python no copian objetos, ellas crean referencias. Cuando utilizamos el método `copy()` de secuencias y diccionarios estamos apenas creando una copia superficial del objeto, o mejor, este tipo de copia alcanza los objetos mutables dentro del objeto copiado.

Para solucionar esto podemos utilizar la biblioteca `copy` de Python que pone a disposición métodos de copia superficial y profunda. La diferencia entre este dos tipos de copia solo tiene relevancia para objetos compuestos (objetos que tienes otros objetos).

Una copia superficial construye un nuevo objeto anidado e inserta en él referencias a los objetos encontrados en el original. En una copia profunda es construido un nuevo objeto anidado y entonces, recursivamente, son insertados en copias de los objetos encontrados en el original.

Para utilizar los métodos de la biblioteca `copy` necesitamos importar el paquete para nuestro proyecto. Para eso basta utilizar la instrucción `import` seguida del nombre de la biblioteca que deseamos cargar en nuestro proyecto.

```
import copy
```

Para crear una copia profunda usamos el método `deepcopy()` como presentado en el código de abajo:

```
dataset_copia = copy.deepcopy(dataset)
```

A partir de ahora podemos realizar alteraciones en cualquier parte del nuevo diccionario **`dataset_copia`** que las alteraciones no serán reflejadas en el diccionario original.

```
dataset_copia['Dodge']['acessorios'].append('Bancos de cuero')
dataset_copia
```

**Salida:**

```
{
    'Dodge': {
        'motor': 'Motor 6.7',
        'año': 2017,
        'km': 45757,
        'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia', 'Airbag','Bancos de cuer
o'],
        'valor': 92612.1
    },
    'Pajero': {
```

```
        'motor': 'Motor 2.4 Turbo',
        'Cero_km': True,
        'km': 80000,
        'accesorios': ['Control de tracción', 'Bancos de cuero', 'Control de estabilidad'],
        'valor': 51606.59
    }
}

dataset
```

**Salida:**

```
{
    'Dodge': {
        'motor': 'Motor 6.7',
        'año': 2017,
        'km': 45757,
        'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia', 'Airbag'],
        'valor': 92612.1
    },
    'Pajero': {
        'motor': 'Motor 2.4 Turbo',
        'Cero_km': True,
        'km': 80000,
        'accesorios': ['Control de tracción', 'Bancos de cuero', 'Control de estabilidad'],
        'valor': 51606.59
    }
}
```

CAPÍTULO 4

# ESTRUCTURAS CONDICIONALES Y DE REPETICIÓN

## 4.1 INSTRUCCIÓN IF

Durante la construcción de un programa surgen momentos en que necesitamos verificar algunas condiciones y modificar el comportamiento del programa con base en la respuesta obtenida. Para eso utilizamos instrucciones de control de flujo y vamos a comenzar con la forma más simple, la instrucción `if`.

El formato patrón de una instrucción `if` simple es presentado a seguir. Esta instrucción verifica una condición que, siendo verdadera, ejecutará el bloque de instrucciones indentado, caso contrario, ninguna acción será realizada.

**Formato patrón**

```
if <condición>:
    <instrucciones caso la condición sea verdadera>
```

Para crear las condiciones que serán verificadas por una estructura condicional vamos a necesitar de los conjuntos de operadores:

- Operadores de comparación:

| Operador | Descripción |
|---|---|
| == | Igualdad |
| != | Diferencia |
| > | Mayor que |
| < | Menor que |
| >= | Mayor o igual |
| <= | Menor o igual |

- Operadores lógicos:

| Operador | Descripción |
|---|---|
| and | Y |

| | |
|---|---|
| `or` | O |
| `not` | Negación |

Vamos a comenzar con ejemplos simples con operadores de comparación y en las próximas secciones avanzamos para los operadores lógicos. Considere el código de abajo donde creamos la variable **`anho`** , que representa el año de fabricación de un vehículo, y asignamos a esta variable el valor 2018.

```
anho = 2018
```

El próximo código crea una prueba que va generar como respuesta un booleano. Es como si estuviésemos haciendo la siguiente pregunta: "¿La variable **`anho`** es mayor o igual a 2019?". Como sabemos la variable **`anho`** tiene como valor 2018 y, por lo tanto, esa comparación no es verdadera por tanto retorna *False* .

```
anho >= 2019
```

**Salida:**

```
False
```

Vamos a utilizar la misma idea solo que ahora con una instrucción `if` simple para comprobar la condición y ejecutar una determinada tarea caso la condición sea verdadera. En este ejemplo estamos creando un clasificador de vehículos según los años de fabricación.

```
anho = 2020

if(anho >= 2019):
    print('Vehículo nuevo')
```

**Salida:**

```
Vehículo nuevo
```

El código anterior solamente imprime "Vehículo nuevo" si la condición fuera verdadera (*True*), caso contrario nada es ejecutado y el programa termina.

## 4.2 INSTRUCCIONES IF-ELSE Y IF-ELIF-ELSE

En algunas situaciones puede ser necesaria alguna acción específica para el caso contrario en una instrucción `if` , o sea, cuando la condición no retorna *True*. Para esto utilizamos la palabra-clave `else` que posibilita la creación de un conjunto de instrucciones para ser ejecutadas caso la condición retorna `False` . Observa la siguiente sintaxis:

**Formato patrón**

```
if <condición>:
    <instrucciones caso la condición sea verdadera>
else:
```

```
    <instrucciones caso la condición no sea verdadera>
```

Para ejemplificar suponga que queremos colocar una nueva clasificación para los carros en nuestro clasificador. Para esto vamos a verificar si el vehículo fue fabricado en 2019 o después, y para estos casos imprimimos "Vehículo nuevo", en los demás casos, o sea, para vehículos fabricados en años anteriores al 2019, imprimimos "Vehículo seminuevo".

```
anho = 2018

if(anho >= 2019):
    print('Vehículo nuevo')
else:
    print('Vehículo seminuevo')
```

**Salida:**

```
Vehículo seminuevo
```

Una tercera posibilidad de uso de estructuras condicionales es cuando tenemos más de dos alternativas, o mejor, cuando necesitamos realizar más de una prueba en una única estructura condicional. Esto es posible con el uso de la palabra-clave `elif` (una unión de `else` y `if`) que puede ser utilizada una o más veces dentro de una estructura condicional. El formato patrón de este tipo de estructura puede ser visto a seguir.

**Formato patrón**

```
if <condición 1>:
    <instrucciones caso la condición 1 sea verdadera>
elif <condición 2>:
    <instrucciones caso la condición 2 seja verdadera>
elif <condición 3>:
    <instrucciones caso la condición 3 seja verdadera>
                        .
                        .
                        .
else:
    <instrucciones caso las condiciones anteriores no sean verdaderas>
```

Antes de ejemplificar el uso de `if-elif-else` vamos a regresar a los operadores lógicos que vimos en la sección anterior. Las tablas abajo son conocidas como tablas de verdad y son un recurso muy utilizado en lógica matemática para definir el valor lógico de una preposición. Las dos tablas presentan los resultados posibles para el uso de los operadores `and` y `or` con dos condiciones A y B.

Utilizando el operador lógico `and` solo tendremos un retorno *True* si la condición A fuera *True* y la condición B también fuera *True*, en los demás casos siempre tendremos como retorno un *False*.

**Tabla verdad - operador lógico *AND***

| A | B | A and B |
|---|---|---|
| True | True | True |
| True | False | False |

| | | |
|---|---|---|
| False | True | False |
| False | False | False |

Ya para el operador lógico `or` la única forma de que la expresión retorne *False* es si la condición A fuera *False* y la condición B también fuera *False*, en las demás situaciones la expresión siempre será verdadera.

**Tabla verdad - operador lógico *OR***

| A | B | A or B |
|---|---|---|
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

Regresando al ejemplo anterior, vamos a asumir ahora que queremos clasificar los carros en tres grupos según el año de fabricación. Para carros con año de fabricación igual o superior la 2019 clasificamos como "Vehículo nuevo", para carros con año de fabricación entre 2016 (exclusive) y 2018 (inclusive) clasificamos como "Vehículo seminuevo" y para los demás vehículos clasificamos como "Vehículo usado". El código abajo ejecuta estas tareas.

Observe que en la línea donde tenemos la instrucción `elif` tenemos la siguiente condición: `anho > 2016 and anho <= 2018` . Aquí probamos dos condiciones y después verificamos la tabla de verdad para evaluar si la expresión es verdadera o falsa.

```
anho = 2017

if(anho >= 2019):
    print('Vehículo nuevo')
elif(anho > 2016 and anho <= 2018):
    print('Vehículo seminuevo')
else:
    print('Vehículo usado')
```

**Salida:**

```
Vehículo seminuevo
```

En una estructura `if-elif-else` podemos tener un número indeterminado de cláusulas `elif` y si hubiera una cláusula `else` , ella debe estar siempre al final del bloque, pero no es obligatoria la utilización de una cláusula `else` .

## 4.3 INSTRUCCIÓN FOR

En esta sección vamos a conocer el bucle `for` que es una de las principales construcciones de *loop*

de Python. Esta instrucción nos permite recorrer los elementos de los objetos de secuencias y ejecutar un conjunto de tareas para cada elemento de estos objetos.

El formato patrón de un bucle `for` comienza con el encabezado que define una variable de destino para cada ítem de la iteración (), juntamente con el objeto que será recorrido (). Después del encabezado sigue el bloque de instrucciones indentado.

**Formato patrón**

```
for <item> in <iterable>:
    <instrucciones>
```

## *Loop* simples

El código de abajo ejemplifica un *loop* simple que imprime el cuadrado de todos los ítems de una secuencia de números. Note que utilizamos un `range()` como iterable para este bucle `for` .

```
for i in range(1, 11):
    print(i ** 2)
```

**Salida:**

```
1
4
9
16
25
36
49
64
81
100
```

## *Loops* con secuencias

En secuencias, como las listas y tuplas, también es posible iterar por los ítems con el uso de los bucles `for` . Considere la lista **`accesorios`** para nuestro ejemplo.

```
accesorios = ['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero',
'Aire acondicionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia']
```

En la instrucción `for` a seguir, el nombre **`item`** es asignado a cada elemento de la lista **`accesorios`** y la instrucción `print` es ejecutada para cada uno.

```
for item in accesorios:
    print(item)
```

**Salida:**

```
Llantas de aleación
Cerraduras eléctricas
Piloto automático
Bancos de cuero
```

```
Aire acondicionado
Sensor de estacionamiento
Sensor crepuscular
Sensor de lluvia
```

Regresando a la idea del clasificador de vehículos de acuerdo con el año de fabricación, suponga que tenemos un conjunto de informaciones sobre los vehículos organizadas en listas. La variable **`datos`** que tenemos a seguir es una lista que contiene listas con datos sobre los vehículos.

```
# 1º item de la lista - Nombre del vehículo
# 2º item de la lista - Año de fabricación
# 3º item de la lista - ¿Vehículo es cero km?

datos = [
    ['Jetta Variant', 2003, False],
    ['Passat', 1991, False],
    ['Crossfox', 1990, False],
    ['DS5', 2019, True],
    ['Aston Martin DB4', 2006, False],
    ['Palio Weekend', 2017, False],
    ['A5', 2019, True],
    ['Série 3 Cabrio', 2009, False],
    ['Dodge Jorney', 2018, False],
    ['Carens', 2011, False]
]
```

Cuando utilizamos un bucle `for` para iterar por los ítems de la lista **`datos`** tenemos la siguiente situación:

```
for item in datos:
    print(item)
```

**Salida:**

```
['Jetta Variant', 2003, False]
['Passat', 1991, False]
['Crossfox', 1990, False]
['DS5', 2019, True]
['Aston Martin DB4', 2006, False]
['Palio Weekend', 2017, False]
['A5', 2019, True]
['Série 3 Cabrio', 2009, False]
['Dodge Jorney', 2018, False]
['Carens', 2011, False]
```

Como los elementos de la lista **`datos`** también son listas, para acceder a sus elementos podemos utilizar el operador de indexación []. En el siguiente ejemplo accedemos apenas a los años de fabricación de cada vehículo del *dataset*.

```
for item in datos:
    print(item[1])
```

**Salida:**

```
2003
1991
1990
2019
```

```
2006
2017
2019
2009
2018
2011
```

Así, para crear nuestro clasificador basta colocar la estructura condicional que creamos en la sección anterior dentro del bucle `for` y hacer algunas adaptaciones.

```
for item in datos:
    if(item[1] >= 2019):
        print('Vehículo nuevo')
    elif(item[1] > 2016 and item[1] <= 2018):
        print('Vehículo seminuevo')
    else:
        print('Vehículo usado')
```

**Salida:**

```
Vehículo usado
Vehículo usado
Vehículo usado
Vehículo nuevo
Vehículo usado
Vehículo seminuevo
Vehículo nuevo
Vehículo usado
Vehículo seminuevo
Vehículo usado
```

## *Loops* con diccionarios

**Documentación**: https://docs.python.org/3.6/library/stdtypes.html#mapping-types-dict

Con diccionarios las iteraciones pueden ser hechas a partir de métodos específicos. Abajo tenemos algunos métodos soportados por diccionarios Python que nos ayudan en la hora de hacer iteraciones. En la tabla la letra ***d*** representa un diccionario cualquiera.

| Operaciones | Resultado |
|---|---|
| `d.keys()` | Retorna una nueva visualización de las claves del diccionario. |
| `d.values()` | Retorna una nueva visualización de los valores del diccionario. |
| `d.items()` | Retorna una nueva visualización de los pares (clave, valor) del diccionario. |

El código a seguir construye el diccionario **`datos`** que tiene como claves los nombres de los vehículos y como valores el valor de venta de cada vehículo.

```
datos = {'Crossfox': 72832.16, 'DS5': 124549.07, 'Volkswagen': 150000, 'Jetta Variant': 88078.64, 'Passat': 106161.95}
datos
```

**Salida:**

```
{'Crossfox': 72832.16,
```

```
 'DS5': 124549.07,
 'Volkswagen': 150000,
 'Jetta Variant': 88078.64,
 'Passat': 106161.95}
```

El método `keys()` retorna una nueva visualización de las claves de un diccionario. Observe el código de abajo.

```
datos.keys()
```

**Salida:**

```
dict_keys(['Crossfox', 'DS5', 'Volkswagen', 'Jetta Variant', 'Passat'])
```

Con esto podemos recorrer las claves de un diccionario y también acceder a sus valores.

```
for clave in datos.keys():
    print(datos[clave])
```

**Salida:**

```
72832.16
124549.07
150000
88078.64
106161.95
```

El método `values()` funciona de forma semejante, retornando una nueva visualización de los valores del diccionario.

```
datos.values()
```

**Salida:**

```
dict_values([72832.16, 124549.07, 150000, 88078.64, 106161.95])
```

Esto nos permite acceder directamente todos los valores de un diccionario. El código de abajo hace una sumatoria de los valores de los vehículos del diccionario **`datos`** .

```
valorTotal = 0
for valor in datos.values():
    valorTotal += valor

valorTotal
```

**Salida:**

```
541621.82
```

En el código anterior utilizamos el operador de asignación +=. Este operador es utilizado cuando deseamos hacer sumas acumuladas con una misma variable. Las dos líneas de código de abajo producen el mismo resultado:

```
valorTotal += valor

valorTotal = valorTotal + valor
```

El método `items()` es más completo, ofreciendo una nueva visualización de los pares (clave, valor) de un diccionario.

```
datos.items()
```

**Salida:**

```
dict_items([('Crossfox', 72832.16), ('DS5', 124549.07), ('Volkswagen', 150000), ('Jetta Variant', 880
78.64), ('Passat', 106161.95)])
```

Utilizando la técnica de desempaquetado podemos generar dos variables de iteración en apenas un bucle `for` , de la siguiente manera:

```
for nombre, valor in datos.items():
    print(f'El vehículo {nombre} cuesta $ {valor}')
```

**Salida:**

```
El vehículo Crossfox cuesta $ 72832.16
El vehículo DS5 cuesta $ 124549.07
El vehículo Volkswagen cuesta $ 150000
El vehículo Jetta Variant cuesta $ 88078.64
El vehículo Passat cuesta $ 106161.95
```

## Instrucciones `continue` e `break`

La instrucción `continue` es utilizada para saltar el bloque de instrucciones actual y retornar a la instrucción `for` . Eso hace que el bucle `for` avance para la próxima iteración, ignorando todo lo que venga después de la instrucción `continue` .

Como ejemplo, suponga que necesitamos consultar una lista de accesorios e imprimir apenas los accesorios que comienzan con determinada letra. El código de abajo hace eso para los accesorios que comienzan con la letra 'S'. Recordando que *strings* son secuencias inmutables de caracteres y por eso podemos tener acceso a cada carácter de una *string* utilizando el operador de indexación []. Como queremos acceder al primer carácter de todos los accesorios utilizamos `item[0]` .

El siguiente código utiliza la instrucción `continue` dentro de una estructura condicional `if` que verifica si la primera letra de cada accesorio es diferente de 'S'. Si fuera *True* ignoramos el resto del bloque ( `print(item)` ) y avanzamos para la próxima iteración.

```
accesorios = ['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero',
'Aire acondicionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia']

for item in accesorios:
    if(item[0] != 'S'):
        continue
    print(item)
```

**Salida:**

```
Sensor de estacionamiento
Sensor crepuscular
```

```
Sensor de lluvia
```

Con la instrucción `break` podemos salir totalmente de un bucle `for`.

Aprovechando el ejemplo anterior, considera que nuestro buscador quiere apenas encontrar la primera ocurrencia de un accesorio que comienza con determinada letra. Para eso después de la instrucción `print(item)` utilizamos la palabra reservada `break`.

```
for item in accesorios:
    if(item[0] != 'S'):
        continue
    print(item)
    break
```

**Salida:**

```
Sensor de estacionamiento
```

La instrucción `break` es bastante utilizada en bucles o *loops* anidados (próxima sección), pues ella aplica unicamente al *loop* donde está inserida, manteniendo la iteración de los *loops* con mayor control externo.

## 4.4 INSTRUCCIÓN FOR ANIDADA

Esta es una forma de crear un bucle o *loop* dentro de otro *loop*. Bastante útil para iterar en estructuras de datos más complejas como secuencias y diccionarios anidados.

Abajo tenemos un formato patrón apenas para mostrar que los *loops* más internos deben estar siempre indentados. Eso hace que el Python entienda que se trata de un bucle `for` dentro de otro bucle `for`.

**Formato patrón**

```
for <item_1> in <iterável_1>:
    for <item_2> in <iterável_2>:
        <instrucciones>
```

Considera la lista anidada **`datos`** con el conjunto de accesorios de tres vehículos. El objetivo es transformar esta lista anidada en una lista simple con todos los accesorios de los tres vehículos.

```
datos = [
    ['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero', 'Aire ac
ondicionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia'],
    ['Central multimedia', 'Techo panorámico', 'Frenos ABS', '4 X 4', 'Panel digital', 'Piloto automá
tico', 'Bancos de cuero', 'Cámara de estacionamiento'],
    ['Piloto automático', 'Control de estabilidad', 'Sensor crepuscular', 'Frenos ABS', 'Cambio autom
ático', 'Bancos de cuero', 'Central multimedia', 'Vidrios elétricos']
]
```

Haciendo un *loop* simple tenemos acceso a las tres listas individualmente.

```
for lista in datos:
```

```
    print(lista)
```

**Salida:**

```
['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero', 'Aire acondi
cionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia']
['Central multimedia', 'Techo panorámico', 'Frenos ABS', '4 X 4', 'Panel digital', 'Piloto automático
', 'Bancos de cuero', 'Cámara de estacionamiento']
['Piloto automático', 'Control de estabilidad', 'Sensor crepuscular', 'Frenos ABS', 'Cambio automátic
o', 'Bancos de cuero', 'Central multimedia', 'Vidrios elétricos']
```

Para acceder a los ítems de cada lista en un mismo procedimiento de iteración podemos utilizar un nuevo bucle `for` que tendrá como `<iterable>` la lista proveniente del bucle `for` más externo (anterior). Vea el siguiente código:

```
for lista in datos:
    for item in lista:
        print(item)
```

**Salida:**

```
Llantas de aleación
Cerraduras eléctricas
Piloto automático
Bancos de cuero
Aire acondicionado
Sensor de estacionamiento
Sensor crepuscular
Sensor de lluvia
Central multimedia
Techo panorámico
Frenos ABS
4 X 4
Panel digital
Piloto automático
Bancos de cuero
Cámara de estacionamiento
Piloto automático
Control de estabilidad
Sensor crepuscular
Frenos ABS
Cambio automático
Bancos de cuero
Central multimedia
Vidrios elétricos
```

Para crear una nueva lista con todos los accesorios del *dataset* basta ejecutar el código abajo.

```
accesorios = []

for lista in datos:
    for item in lista:
        accesorios.append(item)

accesorios
```

**Salida:**

```
['Llantas de aleación',
 'Cerraduras eléctricas',
```

```
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia',
 'Central multimedia',
 'Techo panorámico',
 'Frenos ABS',
 '4 X 4',
 'Panel digital',
 'Piloto automático',
 'Bancos de cuero',
 'Cámara de estacionamiento',
 'Piloto automático',
 'Control de estabilidad',
 'Sensor crepuscular',
 'Frenos ABS',
 'Cambio automático',
 'Bancos de cuero',
 'Central multimedia',
 'Vidrios elétricos']
```

## 4.5 EJERCICIOS

Estamos aprendiendo en este capítulo a utilizar las herramientas básicas del lenguaje Python para control de flujo. Esta herramienta es encontrada en otros lenguajes de programación y envuelve lógica condicional y bucles de repetición.

Utilice el diccionario de abajo para solución de los ítems del problema.

```
dataset = {
    'Lamborghini': {
        'motor': 'Motor V12',
        'año': 2018,
        'km': 25400,
        'accesorios': ['Llantas de aleación', 'Cambio automático', 'Cerraduras eléctricas', 'Bancos d
e cuero'],
        'valor': 133529.84
    },
    'Dodge': {
        'motor': 'Motor 6.7',
        'año': 2017,
        'km': 45757,
        'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia', 'Cambio automático'],
        'valor': 92612.1
    },
    'Pajero': {
        'motor': 'Motor 2.4 Turbo',
        'año': 2012,
        'km': 80000,
        'accesorios': ['Control de tracción', 'Bancos de cuero', 'Cambio automático', 'Control de est
abilidad', '4 X 4'],
        'valor': 51606.59
    }
}
```

1. Crea un bucle de repetición ( `for` ) para imprimir las informaciones de los tres vehículos del ***dataset***.

**Respuesta:**

```
for valor in dataset.values():
    print(valor)
```

**Salida:**

```
{'motor': 'Motor V12', 'año': 2018, 'km': 25400, 'accesorios': ['Llantas de aleación', 'Cambio au
tomático', 'Cerraduras eléctricas', 'Bancos de cuero'], 'valor': 133529.84}
{'motor': 'Motor 6.7', 'año': 2017, 'km': 45757, 'accesorios': ['Llantas de aleación', '4 X 4', '
Central multimedia', 'Cambio automático'], 'valor': 92612.1}
{'motor': 'Motor 2.4 Turbo', 'año': 2012, 'km': 80000, 'accesorios': ['Control de tracción', 'Ban
cos de cuero', 'Cambio automático', 'Control de estabilidad', '4 X 4'], 'valor': 51606.59}
```

2. Desarrolla el código del ítem anterior para posibilitar la impresión de las listas de accesorios de cada vehículo.

**Respuesta:**

```
for valor in dataset.values():
    print(valor['accesorios'])
```

**Salida:**

```
['Llantas de aleación', 'Cambio automático', 'Cerraduras eléctricas', 'Bancos de cuero']
['Llantas de aleación', '4 X 4', 'Central multimedia', 'Cambio automático']
['Control de tracción', 'Bancos de cuero', 'Cambio automático', 'Control de estabilidad', '4 X 4'
]
```

3. Crea una lista llamada **`accesorios`** y almacene en ella todos los accesorios de los tres vehículos. Utiliza un bucle `for` anidado para esta tarea.

**Respuesta:**

```
accesorios = []
for valor in dataset.values():
    for item in valor['accesorios']:
        accesorios.append(item)

accesorios
```

**Salida:**

```
['Llantas de aleación',
'Cambio automático',
'Cerraduras eléctricas',
'Bancos de cuero',
'Llantas de aleación',
'4 X 4',
'Central multimedia',
'Cambio automático',
'Control de tracción',
'Bancos de cuero',
'Cambio automático',
'Control de estabilidad',
'4 X 4']
```

4. Crea una nueva lista **`accesorios`** sin los ítems repetidos.

**Respuesta:**

```
accesorios = list(set(accesorios))
accesorios
```

**Salida:**

```
['Cambio automático',
'Central multimedia',
'Llantas de aleación',
'Cerraduras eléctricas',
'4 X 4',
'Control de tracción',
'Control de estabilidad',
'Bancos de cuero']
```

## 4.6 DESAFÍO

Ahora vamos a organizar nuestra lista de vehículos en un diccionario que clasifica estos vehículos de acuerdo con sus primeras letras. Este ejercicio nos va permitir entrenar los conocimientos sobre control de flujo, métodos y operaciones con secuencias y diccionarios. Utiliza la lista de la siguiente celda para resolver el problema.

```
carros = ['Lamborghini', 'Versa', '207 Sedan', 'Dodge Charger', 'Aston Martin', 'Pajero TR4', 'Tiggo'
, 'Kangoo Express', 'Mohave', 'Livina', '207 SW', 'Palio Adventure', 'Tucson', 'Doblò', '500 Abarth',
'Jetta Variant', 'Range Rover Evoque', 'Up!', 'Grand Cherokee', 'HB20', 'Boxer', 'NX 200t', 'Bongo',
'RAM', 'Parati', 'Tiida Hatch', 'CR-V', 'Boxster', 'Golf', '308 CC', 'Fox', 'Z4 Roadster', 'Sandero',
'Polo Sedan', 'Veloster', 'Cadillac', 'Ford Edge', 'Discovery 4', 'EcoSport', 'Hilux', 'Logan', 'Gol
G4', 'Sorento', 'Santa Fe', 'Vantage Volante', 'Xsara Picasso', 'Volkswagen', 'Crossfox', 'Touareg',
'Gallardo LP 560 – 4', 'Optima', 'Jumper', 'Uno', 'J3', 'Edge', 'C3 Picasso', 'Voyage', 'Cayenne', 'P
assat', 'Sonata', 'Aircross', 'Linea', 'Veracruz', 'Fit', 'Idea', 'A3', 'SpaceFox', 'New Fiesta', 'DS
5', 'Ford EcoSport', 'Jeep Renegade', 'Accord']
```

1. Declara un diccionario sin contenido (vació) y llámalo de **`nombres_carros`** .

   **Respuesta:**

   ```
   nombres_carros = {}
   ```

2. Construye un bucle `for` para leer cada ítem de la lista **`carros`** . Dentro del bucle crea una variable llamada **`primera_letra`** y almacena en ella la primera letra del nombre de cada vehículo.

   **Respuesta:**

   ```
   nombres_carros = {}

   for carro in carros:
       primera_letra = carro[0]
   ```

3. Nuestro objetivo ahora es obtener un diccionario en el siguiente formato:

   ```
   {
       '2': ['207 Sedan', '207 SW'],
       '3': ['308 CC'],
       '5': ['500 Abarth'],
   ```

```
    'A': ['Aston Martin', 'Aircross', 'A3', 'Accord'],
    'B': ['Boxer', 'Bongo', 'Boxster'],
                                    .
                                    .
                                    .
    'T': ['Tiggo', 'Tucson', 'Tiida Hatch', 'Touareg'],
    'U': ['Up!', 'Uno'],
    'V': ['Versa', 'Veloster', 'Vantage Volante', 'Voyage', 'Veracruz'],
    'X': ['Xsara Picasso'],
    'Z': ['Z4 Roadster']
}
```

Donde las **claves** sean las letras iniciales de los nombres de los vehículos y los **valores** una lista con los nombres de todos los vehículos iniciados por la respectiva letra. Elabore el resto del código para obtener el resultado de arriba.

Tip: Crea una cláusula `if-else` , dentro del bucle `for` , que verifica si la clave con la primera letra de la palabra ya existe en el diccionario. Caso no exista, crea la clave y atribuye como valor una nueva lista conteniendo el nombre del vehículo. Caso exista, haga el *append* del nombre del vehículo en la lista existente.

**Respuesta:**

```
nombres_carros = {}

for carro in carros:
    primera_letra = carro[0]
    if(primera_letra in nombres_carros):
        nombres_carros[primera_letra].append(carro)
    else:
        nombres_carros[primera_letra] = [carro]

nombres_carros
```

**Salida:**

```
{'L': ['Lamborghini', 'Livina', 'Logan', 'Linea'],
'V': ['Versa', 'Veloster', 'Vantage Volante', 'Voyage', 'Veracruz'],
'2': ['207 Sedan', '207 SW'],
'D': ['Dodge Charger', 'Doblò', 'Discovery 4', 'DS5'],
'A': ['Aston Martin', 'Aircross', 'A3', 'Accord'],
'P': ['Pajero TR4', 'Palio Adventure', 'Parati', 'Polo Sedan', 'Passat'],
'T': ['Tiggo', 'Tucson', 'Tiida Hatch', 'Touareg'],
'K': ['Kangoo Express'],
'M': ['Mohave'],
'5': ['500 Abarth'],
'J': ['Jetta Variant', 'Jumper', 'J3', 'Jeep Renegade'],
'R': ['Range Rover Evoque', 'RAM'],
'U': ['Up!', 'Uno'],
'G': ['Grand Cherokee', 'Golf', 'Gol G4', 'Gallardo LP 560 – 4'],
'H': ['HB20', 'Hilux'],
'B': ['Boxer', 'Bongo', 'Boxster'],
'N': ['NX 200t', 'New Fiesta'],
'C': ['CR-V', 'Cadillac', 'Crossfox', 'C3 Picasso', 'Cayenne'],
'3': ['308 CC'],
'F': ['Fox', 'Ford Edge', 'Volkswagen', 'Fit', 'Ford EcoSport'],
'Z': ['Z4 Roadster'],
'S': ['Sandero', 'Sorento', 'Santa Fe', 'Sonata', 'SpaceFox'],
```

```
    'E': ['EcoSport', 'Edge'],
    'X': ['Xsara Picasso'],
    'O': ['Optima'],
    'I': ['Idea']}
```

# 4.7 LIST COMPREHENSIONS

## *List comprehensions* patrón

**Documentación**: https://docs.python.org/3.6/tutorial/datastructures.html#list-comprehensions

*List comprehension* es un recurso del Python que permite la creación de una nueva lista de forma más concisa.

### Formato patrón

```
[ <expresión> for <item> in <iterable> if <condición> ]
```

El formato patrón para una *list comprehension*, como mostrado arriba, es equivalente al siguiente bucle `for` :

```
lista = []
for <item> in <iterable>:
    if <condición>:
        lista.append(<expresión>)
```

Podemos tener algunas variaciones en este formato, pero la idea básica continúa siendo la de permitir la creación de listas de manera concisa (apenas una línea de código) con la aplicación de filtros y transformaciones en los elementos.

Veamos un ejemplo simple. El siguiente bucle `for` crea la lista **`cuadrado`** conteniendo los elementos del iterable `range(10)` elevados al cuadrado.

```
cuadrado = []
for i in range(10):
    cuadrado.append(i ** 2)

cuadrado
```

**Salida:**

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Utilizando *list comprehension* basta ejecutar el siguiente código para obtener el mismo resultado.

```
[i ** 2 for i in range(10)]
```

**Salida:**

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Como segundo ejemplo vamos a crear la misma lista del código anterior apenas para los números pares del iterable `range(10)` . Para esto necesitamos incluir una cláusula `if` para probar si el número

es par o no.

```
cuadrado_pares = []
for i in range(10):
    if(i % 2 == 0):
        cuadrado_pares.append(i ** 2)

cuadrado_pares
```

**Salida:**

```
[0, 4, 16, 36, 64]
```

con *list comprehension* el código seria el siguiente:

```
[i ** 2 for i in range(10) if (i % 2 == 0)]
```

**Salida:**

```
[0, 4, 16, 36, 64]
```

## *List comprehensions* con cláusulas `if-elif-else`

Como vimos en las secciones anteriores, en algunos casos necesitamos verificar y tratar más de una condición durante la construcción de una lista. Para eso aprendimos como utilizar las instrucciones `if-elif-else` . Con *list comprehension* también es posible utilizar este tipo de estructura condicional.

### Formato patrón

```
[ <expresión 1> if <condición> else <expresión 2> for <item> in <iterable> ]
```

El formato patrón para una *list comprehension*, como el mostrado arriba, es equivalente al siguiente bucle `for` :

```
lista = []
for <item> in <iterável>:
    if <condición>:
        lista.append(<expressão 1>)
    else:
        lista.append(<expressão 2>)
```

Volviendo al ejemplo del clasificador de vehículos según los años de fabricación, considere la lista anidada **datos**.

```
datos = [
    ['Jetta Variant', 2003, False],
    ['Passat', 1991, False],
    ['Crossfox', 1990, False],
    ['DS5', 2019, True],
    ['Aston Martin DB4', 2006, False],
    ['Palio Weekend', 2017, False],
    ['A5', 2019, True],
    ['Série 3 Cabrio', 2009, False],
    ['Dodge Jorney', 2018, False],
    ['Carens', 2011, False]
]
```

Como vimos en las sesiones anteriores, utilizando estructuras condicionales y de repetición creamos un clasificador simple con el siguiente código:

```
tipo = []
for item in datos:
    if(item[1] >= 2019):
        tipo.append('Vehículo nuevo')
    else:
        tipo.append('Vehículo usado')

tipo
```

**Salida:**

```
['Vehículo usado',
 'Vehículo usado',
 'Vehículo usado',
 'Vehículo nuevo',
 'Vehículo usado',
 'Vehículo usado',
 'Vehículo nuevo',
 'Vehículo usado',
 'Vehículo usado',
 'Vehículo usado']
```

Usando *list comprehension* el código se reduce a una línea conforme lo vemos más abajo. Observa que la estructura condicional, en este caso, viene antes del bucle `for` . Diferente del ejemplo de la sección anterior ( `[i ** 2 for i in range(10) if (i % 2 == 0)]` ) que solamente ejecuta el bucle `for` si la condición es verdadera.

```
['Vehículo nuevo' if (item[1] >= 2019) else 'Vehículo usado' for item in datos]
```

**Salida:**

```
['Vehículo usado',
 'Vehículo usado',
 'Vehículo usado',
 'Vehículo nuevo',
 'Vehículo usado',
 'Vehículo usado',
 'Vehículo nuevo',
 'Vehículo usado',
 'Vehículo usado',
 'Vehículo usado']
```

Para utilizar la instrucción `elif` en una *list comprehension* necesitamos modificar un poco la estructura condicional. El código del clasificador con las estructuras básicas sería como mostrado abajo.

```
tipo = []
for item in datos:
    if(item[1] >= 2019):
        tipo.append('Vehículo nuevo')
    elif(item[1] > 2016 and item[1] <= 2018):
        tipo.append('Vehículo seminuevo')
    else:
        tipo.append('Vehículo usado')
```

```
tipo
```

**Salida:**

```
['Vehículo Usado',
 'Vehículo Usado',
 'Vehículo Usado',
 'Vehículo nuevo',
 'Vehículo Usado',
 'Vehículo Seminuevo',
 'Vehículo nuevo',
 'Vehículo Usado',
 'Vehículo Seminuevo',
 'Vehículo Usado']
```

En una *list comprehension* en el lugar de `elif` utilizamos un `else` seguido del resultado caso la condición sea verdadera y luego después colocamos la instrucción `if` con la condición a ser verificada. Sería como presentado abajo:

**Formato patrón**

```
[ <expresión 1> if <condición 1> else <expresión 2> if <condición 2> else <expresión 3> for <item> in
 <iterable> ]
```

Nuestro clasificador completo a partir de una *list comprehension* sería construido de la siguiente forma:

```
['Vehículo nuevo' if (item[1] >= 2019) else 'Vehículo seminuevo' if (item[1] > 2016 and item[1] <= 20
18) else 'Vehículo usado' for item in datos]
```

**Salida:**

```
['Vehículo Usado',
 'Vehículo Usado',
 'Vehículo Usado',
 'Vehículo nuevo',
 'Vehículo Usado',
 'Vehículo Seminuevo',
 'Vehículo nuevo',
 'Vehículo Usado',
 'Vehículo Seminuevo',
 'Vehículo Usado']
```

## *List comprehensions* con bucles o *loops* anidados

También es posible construir una estructura de bucles anidados con el uso de *list comprehensions*. Utilizando el ejemplo de almacenamiento de todos los accesorios de los tres vehículos de la lista **`datos`** en una lista simple, tendríamos el siguiente código con las estructuras patrones.

```
datos = [
    ['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero', 'Aire ac
ondicionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia'],
    ['Central multimedia', 'Techo panorámico', 'Frenos ABS', '4 X 4', 'Panel digital', 'Piloto automá
tico', 'Bancos de cuero', 'Cámara de estacionamiento'],
    ['Piloto automático', 'Control de estabilidad', 'Sensor crepuscular', 'Frenos ABS', 'Cambio autom
ático', 'Bancos de cuero', 'Central multimedia', 'Vidrios eléctricos']
```

```
]

accesorios = []
for lista in datos:
    for item in lista:
        accesorios.append(item)

accesorios
```

**Salida:**

```
['Llantas de aleación',
'Cerraduras eléctricas',
'Piloto automático',
'Bancos de cuero',
'Aire acondicionado',
'Sensor de estacionamiento',
'Sensor crepuscular',
'Sensor de lluvia',
'Central multimedia',
'Techo panorámico',
'Frenos ABS',
'4 X 4',
'Panel digital',
'Piloto automático',
'Bancos de cuero',
'Cámara de estacionamiento',
'Piloto automático',
'Control de estabilidad',
'Sensor crepuscular',
'Frenos ABS',
'Cambio automático',
'Bancos de cuero',
'Central multimedia',
'Vidrios eléctricos']
```

Con *list comprehension* tenemos un código mucho más simple.

```
[item for lista in datos for item in lista]
```

**Salida:**

```
['Llantas de aleación',
'Cerraduras eléctricas',
'Piloto automático',
'Bancos de cuero',
'Aire acondicionado',
'Sensor de estacionamiento',
'Sensor crepuscular',
'Sensor de lluvia',
'Central multimedia',
'Techo panorámico',
'Frenos ABS',
'4 X 4',
'Panel digital',
'Piloto automático',
'Bancos de cuero',
'Cámara de estacionamiento',
'Piloto automático',
'Control de estabilidad',
'Sensor crepuscular',
'Frenos ABS',
```

```
'Cambio automático',
'Bancos de cuero',
'Central multimedia',
'Vidrios eléctricos']
```

El recurso de *list comprehension* para *loops* anidados puede ser un poco más difícil de comprender en el primer momento, pero nota que las instrucciones `for` son organizadas en el orden en que están anidadas. El ejemplo arriba puede ser mejor entendido con la representación gráfica abajo.

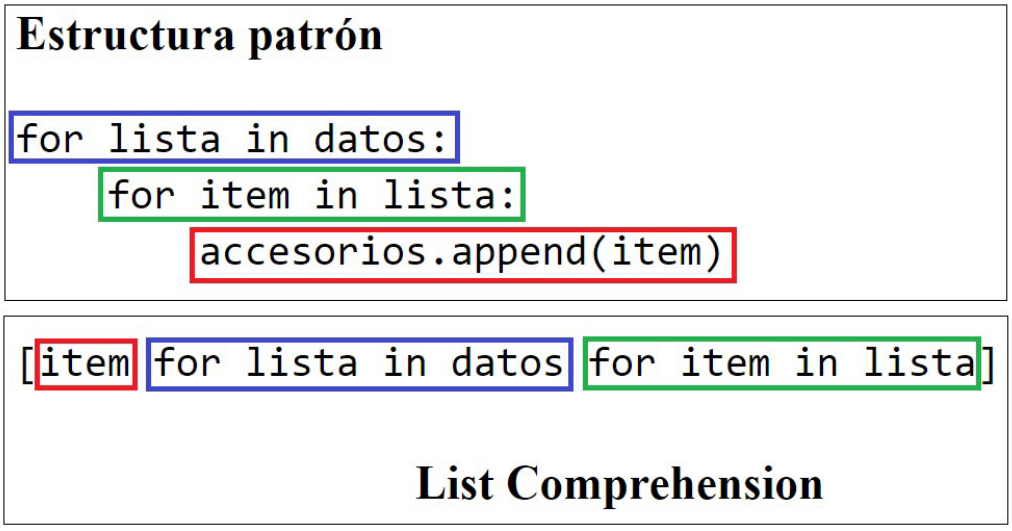



Figura 4.1: Laços for X List Comprehension

Para expresiones simples, el recurso de *list comprehension* vuelve el código conciso y de fácil lectura, también impactando en el performance cuando comparado con los bucles `for` equivalentes.

## 4.8 EJERCICIOS

Los ítems de abajo muestran códigos que generan listas. Evalúa cada uno de ellos y crea las mismas listas utilizando el recurso de *list comprehensions*.

1. Seleccionando letras:

```
letras = []
for letra in 'Data Science':
    letras.append(letra)
letras
```

**Salida:**

```
['D', 'a', 't', 'a', ' ', 'S', 'c', 'i', 'e', 'n', 'c', 'e']
```

**Respuesta:**

```
[letra for letra in 'Data Science']
```

**Salida:**

```
['D', 'a', 't', 'a', ' ', 'S', 'c', 'i', 'e', 'n', 'c', 'e']
```

2. ¿Par o impar?

```
lista = []
for i in range(10):
    if(i % 2 == 0):
        lista.append(2 ** i)
    else:
        lista.append(3 ** i)
lista
```

**Salida:**

```
[1, 3, 4, 27, 16, 243, 64, 2187, 256, 19683]
```

**Respuesta:**

```
[2 ** i if (i % 2 == 0) else 3 ** i for i in range(10)]
```

**Salida:**

```
[1, 3, 4, 27, 16, 243, 64, 2187, 256, 19683]
```

3. Estirando listas.

```
secuencia = []
for lista in listas:
    for item in lista:
        secuencia.append(item)
secuencia
```

**Salida:**

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

Obs.: los datos para solución del problema están en la siguiente celda.

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

**Respuesta:**

```
[item for lista in listas for item in lista]
```

**Salida:**

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

4. Haciendo consultas.

```
consulta = []
for clave, valor in dataset.items():
    if (clave[0] == 'D'):
        for item in valor['accesorios']:
            consulta.append(item)
consulta
```

### Salida:

```
['Llantas de aleación',
'4 X 4',
'Central multimedia',
'Cambio automático',
'Control de tracción',
'Bancos de cuero',
'Control de estabilidad']
```

Obs.: los datos para solución del problema están en la siguiente celda.

```
    dataset = {
        'Lamborghini': {
            'motor': 'Motor V12',
            'año': 2018,
            'km': 25400,
            'accesorios': ['Llantas de aleación', 'Cambio automático', 'Cerraduras eléctricas', '
Bancos de cuero'],
            'valor': 133529.84
        },
        'Dodge': {
            'motor': 'Motor 6.7',
            'año': 2017,
            'km': 45757,
            'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia', 'Cambio automáti
co'],
            'valor': 92612.1
        },
        'DS5': {
            'motor': 'Motor 2.4 Turbo',
            'año': 2012,
            'km': 80000,
            'accesorios': ['Control de tracción', 'Bancos de cuero', 'Control de estabilidad'],
            'valor': 51606.59
        }
    }
```

### Respuesta:

```
    [item for clave, valor in dataset.items() if (clave[0] == 'D') for item in valor['accesorios'
]]
```

### Salida:

```
['Llantas de aleación',
'4 X 4',
'Central multimedia',
'Cambio automático',
'Control de tracción',
'Bancos de cuero',
'Control de estabilidad']
```

CAPÍTULO 5

# FUNCIONES

Funciones son unidades de código reutilizables que realizan una tarea específica, pueden recibir alguna entrada y también pueden retornar algún resultado. Utilizar funciones ayuda en la organización y reutilización del código haciendo que nuestro programa tenga un número menor de líneas de código y por lo tanto sea de fácil lectura y depuración.

El lenguaje Python ya pone a disposición un conjunto de funciones precargadas y listas para su utilización, pero también es posible crear nuestras propias funciones como veremos en este capítulo.

## 5.1 BUILT-IN FUNCTIONS

**Documentación**: https://docs.python.org/3.6/library/functions.html

El lenguaje Python posee varias funciones integradas que están siempre accesibles. Algunas ya utilizamos en nuestro entrenamiento, como a `type()`, `print()`, `zip()`, `len()`, `set()`, etc.

La tabla de abajo presenta las *built-in functions* de Python y una descripción detallada de cada una puede ser encontrada en el *link* informado más arriba.

| | | **Built-in Functions** | | |
|---|---|---|---|---|
| `abs()` | `dict()` | `help()` | `min()` | `setattr()` |
| `all()` | `dir()` | `hex()` | `next()` | `slice()` |
| `any()` | `divmod()` | `id()` | `object()` | `sorted()` |
| `ascii()` | `enumerate()` | `input()` | `oct()` | `staticmethod()` |
| `bin()` | `eval()` | `int()` | `open()` | `str()` |
| `bool()` | `exec()` | `isinstance()` | `ord()` | `sum()` |
| `bytearray()` | `filter()` | `issubclass()` | `pow()` | `super()` |
| `bytes()` | `float()` | `iter()` | `print()` | `tuple()` |
| `callable()` | `format()` | `len()` | `property()` | `type()` |
| `chr()` | `frozenset()` | `list()` | `range()` | `vars()` |
| `classmethod()` | `getattr()` | `locals()` | `repr()` | `zip()` |
| `compile()` | `globals()` | `map()` | `reversed()` | `__import__()` |
| `complex()` | `hasattr()` | `max()` | `round()` | |
| `delattr()` | `hash()` | `memoryview()` | `set()` | |

En algunas situaciones, utilizar las *built-in functions* de Python puede reducir nuestro código y hasta genera una mejora en el performance. Vamos a ver los ejemplos de cómo el uso de *bult-in functions* puede mejorar y reducir nuestro código, considere el diccionario **`datos`** a seguir:

```
datos = {'Jetta Variant': 88078.64, 'Passat': 106161.94, 'Crossfox': 72832.16}
```

Este diccionario contiene los nombres y valores de tres vehículos. Si quisiéramos obtener una lista simple de Python solamente con los valores de cada vehículo del diccionario **`datos`** tendríamos que proceder de la siguiente forma:

```
valores = []
for valor in datos.values():
    valores.append(valor)
valores
```

**Salida:**

```
[88078.64, 106161.94, 72832.16]
```

Ahora, observe el mismo procedimiento con el uso de la *built-in function* `list()`. Apenas con una línea de código obtenemos el mismo resultado del código anterior.

```
list(datos.values())
```

**Salida:**

```
[88078.64, 106161.94, 72832.16]
```

Todavía con el mismo diccionario, suponga que ahora necesitamos obtener una sumatoria de los valores de los tres vehículos. Utilizando nuestro conocimiento en Python podríamos sugerir el siguiente código:

```
suma = 0
for valor in datos.values():
    suma += valor
suma
```

**Salida:**

```
267072.74
```

Con el uso de la *built-in function* `sum()` obtenemos el mismo resultado con apenas una línea de código y sin el uso de bucles *for*.

```
sum(datos.values())
```

**Salida:**

```
267072.74
```

Con estos ejemplos básico ya podemos notar la importancia del uso de las *built-in functions*, pero es importante entender que no es necesario conocer todas las *built-in functions* de Python para poder desarrollar programas en proyectos de *Data Science*. Lo importante aquí es tener en mente que siempre

que surja un problema que demande el desarrollo de tareas más elaboradas, busca antes para saber si una de las funciones disponibles de Python pueden ayudarte en la solución. Y para encontrar esas informaciones podemos utilizar la ayuda de la comunidad Python en el internet y también la propia documentación del lenguaje.

Existen algunos trucos para acceder a la documentación vía *notebook*. Una de ellas es justamente con el uso de la *built-in function* `help()` . Para utilizar esta función basta llamarla y pasar como argumento el nombre del recurso del cual queremos obtener información de ayuda. En el siguiente ejemplo utilizamos `help()` para acceder a la documentación de la *built-in function* `print()` .

```
help(print)
```

**Salida:**

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Otra forma de acceder a la documentación es utilizando el carácter "?" antes o después del nombre del recurso del cual queremos obtener información ( `print?` o `?print` ).

## 5.2 CREANDO FUNCIONES

Como mencionamos al inicio de este capítulo, es posible crear nuestras propias funciones en Python. Este recurso es bastante útil en programación pues nos permite nombrar unidades de código haciendo nuestro programa más fácil de leer y depurar. Usar funciones también permite eliminar el código repetitivo, posibilitando la reutilización de código de forma limpia y simple en otras partes de nuestro programa.

### Funciones sin argumentos

Para definir una función, en su forma más simple, necesitamos apenas especificar un nombre y una secuencia de instrucciones que serán ejecutadas cuando la función es llamada.

**Formato patrón**

```
def <nombre>():
    <instrucciones>
```

En Python, las funciones son definidas con la instrucción `def` seguida del nombre escogido para la

función. Las reglas para nombres de variables que aprendimos en el Capítulo 1 también se aplican para los nombres de funciones. Debemos también evitar tener una función y una variable con el mismo nombre en el código.

Después del nombre de la función utilizamos los paréntesis () que cuando están vacíos indican que se trata de una función sin argumentos. La primera línea de la definición de la función (cabecera de la función) termina con el uso del carácter ":" y luego en la siguiente línea tenemos las instrucciones (cuerpo de la función) que necesitan ser indentadas, así como en los bucles *for*.

Para ejemplificar, considere la función `media()` abajo. Esta función obtiene la media aritmética de los números 1, 2 y 3 y apenas imprime este resultado en la pantalla.

```
def media():
    valor = (1 + 2 + 3) / 3
    print(valor)
```

Note que no obtenemos ningún retorno de la ejecución del código arriba. Este código apenas crea un objeto de función del tipo *function*. Para llamar la función `media()` utilizamos la misma sintaxis de las *built-in functions*.

```
media()
```

**Salida:**

```
2.0
```

## Funciones con argumentos

Algunas funciones exigen uno o más argumentos para poder ejecutar sus instrucciones. Estos argumentos pueden ser obligatorios o opcionales.

**Formato patrón**

```
def <nombre>(<arg_1>, <arg_2>, ..., <arg_n>):
    <instrucciones>
```

Los argumentos obligatorios no vienen previamente configurados, o sea, con un valor *default*, por eso necesitan ser especificados cuando la función es llamada. Veamos un ejemplo de función que recibe argumentos.

```
def media(valor_1, valor_2, valor_3):
    valor = (valor_1 + valor_2 + valor_3) / 3
    print(valor)
```

La función `media()` de arriba ahora tiene tres argumentos (valor_1, valor_2 y valor_3) y necesitan ser informados siempre que la función es llamada, o sea, son obligatorios.

```
media(1, 2, 3)
```

**Salida:**

```
2.0
```

La utilización de argumentos en funciones amplía su uso para diversos tipos de casos, como nuestra función `media()` que ahora permite el cálculo de la media aritmética para cualquier conjunto de tres números.

```
media(23, 45, 67)
```

**Salida:**

```
45.0
```

La función `media()` que definimos tiene tres argumentos obligatorios pues si llamamos esa función y pasamos menos de tres argumentos o más de tres argumentos ocurrirá un error. Para que tengamos argumentos con valores *default* (opcionales) en una función precisamos asignar valores a estos argumentos a la hora de definir la función. Veamos un ejemplo nuevamente con la función `media()`.

```
def media(valor_1, valor_2, valor_3 = 3):
    valor = (valor_1 + valor_2 + valor_3) / 3
    print(valor)
```

Observe que para el tercer argumento (valor*3) asignamos el valor 3 como _default* y ahora cuando llamamos la función `media()` podemos suprimir el valor del último argumento, pues este ya está definido.

```
media(1, 2)
```

**Salida:**

```
2.0
```

Pero también podemos llamar a la función `media()` con tres argumentos y asignar un nuevo valor para el último argumento.

```
media(23, 45, 67)
```

**Salida:**

```
45.0
```

Un punto importante cuando trabajamos con funciones de múltiples argumentos es el orden de declaración de estos argumentos cuando llamamos a función. Ellos deben ser pasados en el orden en que fueron definidos por la instrucción `def` y caso sea necesario informar en otro orden, eso debe ser hecho utilizando los nombres de los argumento de la siguiente forma: `media(valor_2 = 3, valor_1 = 4)`.

Los argumentos de una función pueden también tener formatos diversos (listas, tuplas, diccionarios, etc.). El ejemplo de abajo modifica la función `media()` para calcular la media aritmética de los valores de una lista cualquiera.

```
def media(lista):
    valor = sum(lista) / len(lista)
```

```
    print(valor)
```

Note que en la construcción de la nueva función `media()` utilizamos tres *built-in functions* de Python: `sum()`, `len()` y `print()`.

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

**Salida:**

```
5.0
```

En algunas situaciones puede ser necesario definir una función en la cual no se sabe cuantos argumentos el usuario va pasar. En estos casos, podemos usar la forma especial `*args` para capturar todos los argumentos que serán pasados para la función.

Suponga, como ejemplo, que necesitamos crear una función que sume los valores de todos los argumentos pasados y nos retorne el valor de esta sumatoria, pero no queremos restringir el número de partes de esta suma, o sea, el usuario podrá informar cualquier cantidad de números que la función irá retornar el resultado de la suma de todos ellos. El código de abajo crea esta función.

```
def suma(*args):
    suma = 0
    for i in args:
        suma += i
    print(suma)
```

En este tipo de definición de función el nombre **args** no es importante, pero sí el carácter *que le precede. El nombre **args** es apenas una convención por ser la abreviación de "argument". El carácter* antes de una variable indica que el contenido de esta variable debe ser expandido como una secuencia y por eso conseguimos utilizar **args** como un iterador en un bucle *for* en el código arriba.

```
suma(1, 2)
```

**Salida:**

```
3
```

Vea que ahora podemos llamar a la función `suma()` con cualquier número de argumentos y el valor de la suma de estos valores es retornada.

```
suma(1, 2, 3)
```

**Salida:**

```
6
```

```
suma(1, 2, 3, 4)
```

**Salida:**

```
10
```

## 5.3 PARA SABER MÁS...

**`datetime`**

**Documentación**: https://docs.python.org/3.6/library/datetime.html

El módulo `datetime` pone a disposición clases para manipulación de datas y horas en Python.

```
import datetime

now = datetime.datetime.now()

print(now)
print(now.year)
print(now.month)
print(now.day)
print(now.hour)
print(now.minute)
print(now.second)
```

**Salida:**

```
2020-10-19 09:37:52.519277
2020
10
19
9
37
52
```

## 5.4 EJERCICIOS

Una tarea importante durante el proceso de investigación de un conjunto de datos es la transformación y creación de variables. En este paso es posible crear variables a partir de las variables ya existentes en el *dataset*. Un ejemplo de eso, en nuestro *dataset*, sería la construcción de la variable que indica el kilometraje medio anual de cada vehículo. Para construir esta variable basta calcular la razón entre el kilometraje total del vehículo y la diferencia entre el año corriente y el año de fabricación del vehículo.

En este ejercicio vamos a crear una función para obtener el kilometraje medio anual y actualizar nuestro diccionario de datos sobre los vehículos. Lo que esta función va a necesitar hacer es:

- iterar por los ítems del diccionario;
- seleccionar los campos que serán utilizados en los cálculos;
- realizar los cálculos; y
- actualizar el diccionario de información.

Para eso la función debe recibir como parámetro el diccionario de datos que está en la celda de abajo.

```
dataset = {
```

```
    'Lamborghini': {
        'motor': 'Motor V12',
        'año': 2018,
        'km': 25400,
        'accesorios': ['Llantas de aleación', 'Cambio automático', 'Cerraduras eléctricas', 'Bancos d
e cuero'],
        'valor': 133529.84
    },
    'Dodge': {
        'motor': 'Motor 6.7',
        'año': 2017,
        'km': 45757,
        'accesorios': ['Llantas de aleación', '4 X 4', 'Central multimedia', 'Cambio automático'],
        'valor': 92612.1
    },
    'Pajero': {
        'motor': 'Motor 2.4 Turbo',
        'año': 2012,
        'km': 80000,
        'accesorios': ['Control de tracción', 'Bancos de cuero', 'Cambio automático', 'Control de est
abilidad', '4 X 4'],
        'valor': 51606.59
    }
}
```

1. Crea la estructura básica de nuestra función. Llama la función de **`kmMedia`** , que recibe como argumento **`datos`** y apenas imprime este argumento. Ejecute la función **`kmMedia`** en la segunda celda.

   **Respuesta:**

   ```
   def kmMedia(datos):
       print(datos)

   kmMedia(dataset)
   ```

   **Salida:**

   ```
   {'Lamborghini': {'motor': 'Motor V12', 'año': 2018, 'km': 25400, 'accesorios': ['Llantas de aleac
   ión', 'Cambio automático', 'Cerraduras eléctricas', 'Bancos de cuero'], 'valor': 133529.84}, 'Dodg
   e': {'motor': 'Motor 6.7', 'año': 2017, 'km': 45757, 'accesorios': ['Llantas de aleación', '4 X 4'
   , 'Central multimedia', 'Cambio automático'], 'valor': 92612.1}, 'Pajero': {'motor': 'Motor 2.4 Tu
   rbo', 'año': 2012, 'km': 80000, 'accesorios': ['Control de tracción', 'Bancos de cuero', 'Cambio a
   utomático', 'Control de estabilidad', '4 X 4'], 'valor': 51606.59}}
   ```

2. Dentro de la función especificada en el ítem anterior, elabora el bucle *for* que hará lectura, cálculo del kilometraje medio anual y actualización del diccionario de datos de cada vehículo. Ejecuta la función **`kmMedia`** en la segunda celda.

   **Respuesta:**

   ```
   def kmMedia(datos):
       for llave, valor in datos.items():
           datos[llave]['km_media'] = int(valor['km'] / (2020 - valor['año']))

       print(datos)

   kmMedia(dataset)
   ```

**Salida:**

```
 {'Lamborghini': {'motor': 'Motor V12', 'año': 2018, 'km': 25400, 'accesorios': ['Llantas de aleac
ión', 'Cambio automático', 'Cerraduras eléctricas', 'Bancos de cuero'], 'valor': 133529.84, 'km_me
dia': 12700}, 'Dodge': {'motor': 'Motor 6.7', 'año': 2017, 'km': 45757, 'accesorios': ['Llantas de
 aleación', '4 X 4', 'Central multimedia', 'Cambio automático'], 'valor': 92612.1, 'km_media': 152
52}, 'Pajero': {'motor': 'Motor 2.4 Turbo', 'año': 2012, 'km': 80000, 'accesorios': ['Control de t
racción', 'Bancos de cuero', 'Cambio automático', 'Control de estabilidad', '4 X 4'], 'valor': 516
06.59, 'km_media': 10000}}
```

3. Utiliza el conocimiento adquirido en la sección "para saber más" para mejorar la función **`kmMedia`**.

**Respuesta:**

```
import datetime

def kmMedia(datos):
    data = datetime.datetime.now()

    for llave, valor in datos.items():
        datos[llave]['km_media'] = int(valor['km'] / (data.year - valor['año']))

    print(datos)

kmMedia(dataset)
```

**Salida:**

```
 {'Lamborghini': {'motor': 'Motor V12', 'año': 2018, 'km': 25400, 'accesorios': ['Llantas de aleac
ión', 'Cambio automático', 'Cerraduras eléctricas', 'Bancos de cuero'], 'valor': 133529.84, 'km_me
dia': 12700}, 'Dodge': {'motor': 'Motor 6.7', 'año': 2017, 'km': 45757, 'accesorios': ['Llantas de
 aleación', '4 X 4', 'Central multimedia', 'Cambio automático'], 'valor': 92612.1, 'km_media': 152
52}, 'Pajero': {'motor': 'Motor 2.4 Turbo', 'año': 2012, 'km': 80000, 'accesorios': ['Control de t
racción', 'Bancos de cuero', 'Cambio automático', 'Control de estabilidad', '4 X 4'], 'valor': 516
06.59, 'km_media': 10000}}
```

```
dataset
```

**Salida:**

```
{'Lamborghini': {'motor': 'Motor V12',
'año': 2018,
'km': 25400,
'accesorios': ['Llantas de aleación',
'Cambio automático',
'Cerraduras eléctricas',
'Bancos de cuero'],
'valor': 133529.84,
'km_media': 12700},
'Dodge': {'motor': 'Motor 6.7',
'año': 2017,
'km': 45757,
'accesorios': ['Llantas de aleación',
'4 X 4',
'Central multimedia',
'Cambio automático'],
'valor': 92612.1,
'km_media': 15252},
'Pajero': {'motor': 'Motor 2.4 Turbo',
'año': 2012,
'km': 80000,
```

```
    'accesorios': ['Control de tracción',
    'Bancos de cuero',
    'Cambio automático',
    'Control de estabilidad',
    '4 X 4'],
    'valor': 51606.59,
    'km_media': 10000}}
```

## 5.5 DEFINIENDO FUNCIONES QUE RETORNAN VALORES

Considere la última definición de la función `media()` que utilizamos en la sección anterior. Como podemos ver en el código de abajo esta función recibe una lista de valores, calcula la media aritmética de estos valores y después imprime el resultado de esta media para el usuario.

```
def media(lista):
    valor = sum(lista) / len(lista)
    print(valor)

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

**Salida:**

```
5.0
```

Ahora, suponga que estás interesado en utilizar el resultado de esta media en un otro lugar de tu código. Podrías pensar que para eso bastaría asignar el resultado de la función `media()` a una variable y después utilizar esta variable en otra parte del código.

```
resultado = media([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

**Salida:**

```
5.0
```

Pero cuando intentamos acceder a la variable para ver su contenido notamos que ningún valor es retornado, o sea, ningún valor fue atribuido a la variable **resultado** que acabamos de crear.

```
resultado
```

Eso ocurre porque la función `media()` apenas imprime y no retorna el resultado de la operación. Funciones como esta, son conocidas como funciones nulas, pues pueden exhibir algún resultado en la pantalla, pero no tienen un valor de retorno.

En la siguiente sección hablaremos sobre cómo definir funciones que retornan uno o más valores como respuesta.

### Funciones que retornan un valor

Para definir una función con retorno utilizamos la instrucción `return` y pasamos el resultado que deseamos que la función retorne, luego después. Observe el formato patrón de este tipo de función.

**Formato patrón**

```
def <nombre>(<arg_1>, <arg_2>, ..., <arg_n>):
    <instrucciones>
    return <resultado>
```

Vamos entonces a definir nuevamente nuestra función para el cálculo de las medias, solo que ahora retornando el resultado y no apenas imprimiendo.

```
def media(lista):
  valor = sum(lista) / len(lista)
  return valor
```

Note que la función continua funcionando como antes, solo que ahora podemos asignar su resultado a una variable o hasta mismo llamarla dentro de una expresión u otra función.

```
resultado = media([1, 2, 3, 4, 5, 6, 7, 8])
resultado
```

**Salida:**

```
4.5
```

## Funciones que retornan más de un valor

Algunos paquetes de Python poseen funciones que retornan más de un valor. Estos retornos, en general, vienen en el formato de una tupla con dos o más valores.

El lenguaje Python también nos permite definir funciones que retornan múltiples valores. Observe la sintaxis de definición de una función de este tipo.

**Formato patrón**

```
def <nombre>(<arg_1>, <arg_2>, ..., <arg_n>):
    <instrucciones>
    return <resultado_1>, <resultado_2>, ..., <resultado_n>
```

Mejorando nuestra función `media()`, ahora vamos a definirla para retornar dos informaciones, la media y también el número de valores involucrados en el cálculo de la media, o mejor, el tamaño de la lista pasada como argumento para la función.

```
def media(lista):
  valor = sum(lista) / len(lista)
  return valor, len(lista)
```

Vea que después del `return` basta pasar los valores que deseamos retornar separados por comas. Llamando la nueva función `media()` tenemos como retorno una tupla con los resultados de la operación.

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

**Salida:**

```
(5.0, 9)
```

Como vimos en el capítulo sobre secuencias, podemos utilizar el recurso de desempaquetado de tuplas para asignar los dos valores de retorno a dos variables distintas.

```
resultado, n = media([1, 2, 3, 4, 5, 6, 7, 8, 9])
print(resultado)
print(n)
```

**Salida:**

```
5.0
9
```

## 5.6 FUNCIONES LAMBDA

Las funciones `lambda` son una otra forma de generar objetos de funciones a parte de la instrucción `def` . Así como `def` , esta expresión crea una función que puede ser llamada luego, pero retorna la función en vez de asignarle a un nombre, por eso son llamadas de anónimas.

La forma patrón es la palabra-llave `lambda` , seguida por uno o más argumentos, seguida por una expresión después de los dos puntos.

**Formato patrón**

```
lambda <arg_1>, <arg_2>, ..., <arg_n>: <expresión>
```

`Lambda` siempre retorna un valor (una nueva función) que puede, opcionalmente, recibir un nombre. En el ejemplo de abajo **`cuadrado`** es atribuido al objeto de función que la expresión `lambda` crea, en el caso una función para obtener el cuadrado de un número. Con `def` funciona de la misma forma, pero la asignación es automática.

```
cuadrado = lambda x: x ** 2
```

Para llamar a la función `cuadrado()` procedemos de la misma forma que con una función creada con el uso de `def` .

```
cuadrado(3)
```

**Salida:**

```
9
```

Una función `lambda` puede recibir cualquier número de argumentos, pero está limitada a apenas una expresión.

```
multiplica = lambda x, y: x * y
```

```
multiplica(2, 3)
```

**Salida:**

```
6
```

Las funciones `lambda` son bastante útiles en análisis de datos, porque en diversos casos algunas funciones de transformación de datos aceptan funciones como argumentos. Regresaremos a este asunto cuando veamos la biblioteca Pandas.

## 5.7 PARA SABER MÁS...

### Métodos de *string* - `lower()` y `upper()`

**Documentación**: https://docs.python.org/pt-br/3.6/library/stdtypes.html#text-sequence-type-str

Como ya sabemos, los datos en formato de texto en Python son tratados como *strings*. *Strings* son secuencias inmutables de códigos Unicode y aceptan todas las operaciones comunes de secuencias que vimos en el Capítulo 2 donde trabajamos con listas y tuplas, conjuntamente con algunos métodos adicionales de la clase *str*.

Ahora vamos a conocer dos de estos métodos: `str.lower()` y `str.upper()`.

Estos métodos son bastante útiles en tareas de comparación entre *strings* o partes de *strings*. Porque el lenguaje Python es *case sensitive*, o sea, la *string* "a" es diferente de la *string* "A".

```
"Python" == "PyThOn"
```

**Salida:**

```
False
```

El método `lower()` retorna una copia de la *string* original con todos los caracteres convertidos para letras minúsculas y el método `upper()` retorna una copia de la *string* original con todos los caracteres convertidos para letras mayúsculas.

```
"PyThOn".lower()
```

**Salida:**

```
'python'
```

```
"PyThOn".upper()
```

**Salida:**

```
'PYTHON'
```

Con el uso de estos métodos podemos proceder con comparaciones entre *strings* de forma más segura.

```
"Python".lower() == "PyThOn".lower()
```

**Salida:**

```
True
```

```
"Python".upper() == "PyThOn".upper()
```

**Salida:**

```
True
```

## 5.8 EJERCICIOS

Vamos a crear algunas funciones.

1. Crea una función que reciba como argumento una lista de números y retorne las siguientes estadísticas descriptivas:

   - Media aritmética;
   - Valor máximo; y
   - Valor mínimo;

   Obs.: No utilice las *built-in functions* `min()` y `max()`.

   **Respuesta:**

   ```
   def estatDesc(lista):
     media = sum(lista) / len(lista)
     lista.sort()
     minimo = lista[0]
     maximo = lista[-1]
     return (media, minimo, maximo)

   estatDesc([2, 56, 78, 23, 89, 83, 12, 43])
   ```

   **Salida:**

   (48.25, 2, 89)

2. Crea una función que reciba dos argumentos. El primer argumento debe ser una lista de *strings* (ex.: `['Ana', 'Bob', 'Ted']`) y el segundo debe ser una *string* con apenas un carácter (ex.: `'b'`). Lo que la función debe hacer es retornar una lista con apenas los ítems de la lista de entrada (primer argumento de la función) que inicien con el carácter pasado como segundo argumento. Prueba la función con la lista de nombres de abajo.

   Consejo: Utiliza el conocimiento adquirido en la sección "para saber más" en la construcción de esta función.

   ```
   nombres = ['Mariana', 'Nika', 'Fernando', 'Lúcia', 'Clara', 'Jacques', 'Miracema', 'Calista', 'Jussara', 'Adoniram', 'Gal', 'Amanda', 'Juma', ' ngela', 'Cecília', 'Oriana', 'Liliane', 'Édson', 'Matilde', 'Axel', 'Douglas', 'Mada', 'Ray', 'Ofélia', 'Darlene', 'Dominique', 'Ludmila', 'Luiza', 'Augusta', 'Zélia', 'Cleide', 'Patrício', 'Bóris', 'Bridget', 'Isabelle', 'Luiz', 'Elisabete', 'Salma', 'Zenon', 'Rúbia', 'Zulima', 'Nadya', 'Dante', 'Ararê', 'Tadeu', 'Eloy', 'Yuri', 'Aracy', 'Gildo', 'Zoraide']
   ```

### Respuesta:

```
def busca(lista, letra='A'):
    if (len(letra) != 1):
        return "El segundo parámetro debe ser una string de tamaño 1."
    resultado = []
    for item in lista:
        if (item.lower()[0] == letra.lower()):
            resultado.append(item)
    return resultado

busca(nombres, letra='r')
```

### Salida:

```
['Ray', 'Rúbia']
```

Capítulo 6

# INTRODUCCIÓN A LA BIBLIOTECA NUMPY

## 6.1 ¿POR QUÉ USAR LA BIBLIOTECA NUMPY PARA DATA SCIENCE?

NumPy, abreviación de *Numerical Python*, es uno de las bibliotecas más importantes para procesamiento numérico en Python, ofreciendo la base para la mayoría de otras bibliotecas de aplicaciones científicas que utilizan datos numéricos en el lenguaje. Podemos destacar los siguientes recursos:

- Un poderoso objeto *array* multidimensional;
- Funciones matemáticas sofisticadas para operaciones con *arrays* sin la necesidad de utilización de bucles *for*;
- Recursos de álgebra lineal y generación de números aleatorios.

Fuera de los recursos numéricos y de procesamiento eficiente de *arrays*, la biblioteca NumPy también ejerce un papel importante en análisis de datos como un eficiente contenedor para transporte de datos multidimensionales entre algoritmos y diversas bibliotecas del lenguaje. En la manipulación y almacenamiento de datos numéricos, los *arrays* NumPy son significativamente más eficientes que las estructuras básicas de datos de Python (listas, tuplas etc.).

Aunque NumPy no ofrezca funcionalidades específicas del área de análisis de datos como herramientas de modelaje estadístico, consideramos importante desarrollar el conocimiento de cómo trabajar con *arrays* NumPy y el procesamiento orientado a *arrays* ya que nos ayudará a entender más fácilmente las herramientas con semántica orientada a *arrays*, como la biblioteca Pandas que veremos más adelante.

### Importando toda la biblioteca

El lenguaje Python posee una biblioteca patrón que viene con un conjunto de herramientas importantes para atender una amplia variedad de tareas. Algunas de esas herramientas fueron exploradas en este curso, como las *built-in functions* que utilizamos en el capítulo anterior.

A parte de eso, el lenguaje Python nos permite tener acceso a un amplio ecosistema de herramientas y bibliotecas de terceros que amplían las funcionalidades del lenguaje para áreas y tareas más

específicas, como es el caso de la biblioteca NumPy.

Para cargar módulos de la biblioteca patrón o de terceros en nuestro proyecto basta utilizar la declaración `import` seguida del nombre que identifica la biblioteca de interés. En el caso del paquete NumPy seria de la siguiente forma:

```
import numpy
```

Con la declaración de arriba estamos importando todo el contenido de la biblioteca NumPy en nuestro proyecto y para acceder a cualquier recurso del paquete necesitamos utilizar el nombre del módulo ( `numpy` ) seguido del carácter "." y después el nombre del recurso.

```
numpy.arange(10)
```

**Salida:**

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

## Importando parte de la biblioteca

Es posible también importar algunos recursos específicos de una biblioteca en vez de importar todo su contenido. Para eso basta utilizar a declaración `from` en conjunto con la declaración `import` , de la siguiente forma:

```
from numpy import arange
```

**Salida:** El código de arriba permite que la función `arange` sea accesada sin la necesidad de especificar a qué biblioteca pertenece.

```
arange(10)
```

**Salida:**

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

Este método debe ser utilizado con cautela pues puede sobrescribir nombres de funciones no *namespace* de nuestro proyecto. Un *namespace* es básicamente un sistema para garantizar que todos los nombres en un programa sean exclusivos y puedan ser usados sin ningún conflicto.

## Importando toda la biblioteca y atribuyendo un nuevo nombre

Una práctica bastante común a la hora de realizar el `import` de un paquete es atrbuir un *alias* a él, o sea, un apodo o seudónimo. Por convención la comunidad acostumbra utilizar el seudónimo `np` para el *namespace* `numpy` , de la siguiente forma:

```
import numpy as np
```

Para acceder a un recurso de la biblioteca utilizando este tipo de `import` basta digitar el *alias* ( `np` ) seguido del carácter "." y después el nombre del recurso.

```
np.arange(10)
```

**Salida:**

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

## 6.2 PARA SABER MÁS...

**`numpy.arange([start, ]stop, [step, ]dtype=None)`**

**Documentación**: https://numpy.org/doc/1.18/reference/generated/numpy.arange.html

Retorna valores espaciados de forma uniforme dentro de un determinado intervalo. Los valores son generados dentro del intervalo semiabierto [*start*, *stop*), o sea, el intervalo inclui *start* y exclui *stop*. Para argumentos enteros, la función es equivalente a la *built-in function* `range()` de la biblioteca patrón de Python, pero retornando un *array* NumPy.

## 6.3 ARRAYS NUMPY

La biblioteca NumPy tiene como recurso principal su objeto *array* n-dimensional, el *ndarray*, que puede ser entendido como una colección de "ítems" del mismo tipo. Este objeto funciona como un contenedor multidimensional que permite la realización de operaciones matemáticas de forma rápida, simple y en grandes conjuntos de datos.

Objetos *ndarray* presentan algunos atributos, entre ellos tenemos los atributos `shape` y `dtype`. El atributo `shape` retorna una tupla con el tamaño de cada dimensión del *array* y el atributo `dtype` retorna el tipo de dato del *array*.

Otra característica importante de los *arrays* NumPy es que estos son homogéneos, o sea, solo pueden almacenar un tipo de dato.

### Creando *arrays* NumPy

La forma más básica de creación de *arrays* NumPy es con la utilización de la función `array()`. Esta función puede ser utilizada para convertir otras estructuras de Python, como listas y tuplas, en *arrays* NumPy. En verdad la función `array()` acepta como argumento cualquier tipo de secuencia Python, inclusive otros *arrays*.

#### Creando *arrays* a partir de secuencias

Veamos algunos ejemplos de cómo son realizadas las conversiones de secuencias Python en *arrays* NumPy con el uso de la función `array()`.

```
lista = [1000, 2300, 4987, 1500]
array = np.array(lista)
array
```

**Salida:**

```
array([1000, 2300, 4987, 1500])
```

Utilizando tuplas tenemos el mismo resultado.

```
tupla = 1000, 2300, 4987, 1500
array = np.array(tupla)
array
```

**Salida:**

```
array([1000, 2300, 4987, 1500])
```

Consultando el atributo `shape` del *array* creado en los trechos de código anteriores tenemos el siguiente resultado:

```
array.shape
```

**Salida:**

```
(4,)
```

Indicando que se trata de un *array* unidimensional y que esta dimensión única tiene tamaño 4. Otro atributo importante es el `dtype` que retorna el tipo de dato almacenado en el *array*.

```
array.dtype
```

**Salida:**

```
dtype('int64')
```

Observe que no informamos explícitamente para la función `array()` el tipo de dato del *array*. Eso puede ser hecho con el argumento opcional `dtype` . Cuando el argumento `dtype` no es pasado, la función `array()` intentará inferir el mejor tipo de dato para posibilitar la creación del *array*.

**Creando *arrays* a partir de datos externos**

El paquete NumPy, con el uso de la función `loadtxt()` , posibilita que sean creados *arrays* a partir de archivos externos, como por ejemplo, cargar las informaciones de un archivo con extensión **TXT** y crear un *array* NumPy con este contenido. Suponga el siguiente contenido para el archivo "archivo.txt":

```
0
1
2
3
4
```

Para cargar el contenido de este archivo en un *array* NumPy basta utilizar la función `loadtxt()` de la siguiente forma:

```
np.loadtxt(fname = 'archivo.txt')
```

**Salida:**

```
array([0., 1., 2., 3., 4.])
```

Observa que teníamos originalmente, en el archivo **TXT**, números enteros y después de la importación de su contenido en un *array* NumPy tenemos como resultado un *array* con `dtype('float64')`. Eso pasa porque el argumento `dtype` de la función `loadtxt()` tiene como configuración *default* el valor `'float'`. Para modificar eso basta declarar explícitamente un valor diferente para el argumento `dtype`.

```
np.loadtxt(fname = 'archivo.txt', dtype = int)
```

**Salida:**

```
array([0, 1, 2, 3, 4])
```

Siguiendo con los *arrays* unidimensionales, considera el mismo contenido del archivo **TXT** anterior, pero ahora con una organización diferente.

```
0 1 2 3 4
```

La utilización de la función `loadtxt()` con el contenido de este archivo genera el mismo resultado obtenido anteriormente, o sea, un *array* NumPy unidimensional.

```
np.loadtxt(fname = 'archivo.txt', dtype = int)
```

**Salida:**

```
array([0, 1, 2, 3, 4])
```

Fue mencionado en la introducción de esta sección que el paquete NumPy tiene como recurso principal su objeto *array* n-dimensional. En nuestro curso vamos a concentrarnos apenas en *arrays* de una y dos dimensiones y hasta ahora trabajamos solamente con *arrays* unidimensionales. Ahora vamos a crear un *array* NumPy bidimensional utilizando la función `loadtxt()`. Considere el siguiente archivo **TXT**:

```
0 1
2 3
4 5
6 7
8 9
```

Para generar un *array* bidimensional, con la función `loadtxt()`, cada línea en el archivo de texto debe tener el mismo número de valores.

```
array = np.loadtxt(fname = 'archivo.txt', dtype = int)
array
```

**Salida:**

```
array([[0, 1],
       [2, 3],
```

```
       [4, 5],
       [6, 7],
       [8, 9]])
```

Consultando el atributo `shape` de este nuevo *array* tenemos el siguiente resultado:

```
array.shape
```

**Salida:**

```
(5, 2)
```

El tamaño de la tupla indica el número de dimensiones del *array* (dos) y los valores indican el tamaño de cada dimensión. Para facilitar nuestro entendimiento podemos decir que en el ejemplo de arriba tenemos un array con 5 líneas y 2 columnas de datos.

Una característica común de los tres archivos de texto utilizados como ejemplo es que los valores numéricos dentro de cada uno están separados por un espacio en blanco. Este es el carácter utilizado como delimitador *default* de la función `loadtxt()`. Caso sea necesario asumir otro tipo de delimitador basta configurar el argumento `delimiter`, como en el siguiente ejemplo.

```
1;2;3
4;5;6
7;8;9
10;11;12
13;14;15
```

```
np.loadtxt(fname = 'archivo.txt', delimiter=';', dtype = int)
```

**Salida:**

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12],
       [13, 14, 15]])
```

## *Arrays* con dos dimensiones

En la sección anterior vimos cómo crear *arrays* NumPy de dos dimensiones a partir de datos externos. Ahora vamos a ver como hacer eso utilizando secuencias. Considere el siguiente código:

```
datos = [
    ['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático'],
    ['Central multimedia', 'Techo panorámico', 'Frenos ABS'],
    ['Piloto automático', 'Sensor crepuscular', 'Frenos ABS']
]
```

Para informar a la función `array()` que queremos crear un *array* bidimensional basta pasar como argumento una secuencia anidada, o sea, secuencia(s) dentro de una secuencia. En el ejemplo de arriba la lista `datos` es una lista que contiene otras tres listas.

```
accesorios = np.array(datos)
```

**Salida:**

```
array([['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático'],
       ['Central multimedia', 'Techo panorámico', 'Frenos ABS'],
       ['Piloto automático', 'Sensor crepuscular', 'Frenos ABS']],
      dtype='<U18')
```

Consultando el atributo `shape` de la lista `accesorios` tenemos el siguiente resultado: `(3, 3)`. Eso indica que creamos un *array* de dos dimensiones y que cada dimensión tiene tamaño 3, o sea, una matriz con 3 líneas y 3 columnas.

## 6.4 TIPOS DE DATOS

NumPy posee algunos tipos de datos extras cuando comparamos con los tipos básicos del lenguaje Python (`str`, `int`, `float` y `bool`). La siguiente tabla tiene los códigos y una descripción simple de cada tipo de dato en el NumPy. El tipo de dato *complex* no fue abordado en este curso por no tener aplicación práctica en nuestro campo de estudio.

| Tipo | Código | Descripción |
|---|---|---|
| int8 | i1 | Enteros de 8 bits con signo |
| int16 | i2 | Enteros de 16 bits con signo |
| int32 | i4 | Enteros de 32 bits con signo |
| int64 | i8 | Enteros de 64 bits con signo |
| uint8 | u1 | Enteros de 8 bits sin signo |
| uintl6 | u2 | Enteros de 16 bits sin signo |
| uint32 | u4 | Enteros de 32 bits sin signo |
| uint64 | u8 | Enteros de 64 bits sin signo |
| float16 | f2 | Punto flotante de media precisión |
| float32 | f4 o f | Punto flotante |
| float64 | f8 o d | Punto flotante de doble precisión (compatible con el objeto `float` de Python) |
| float128 | f16 o g | Punto flotante de precisión extendida |
| bool | ? | Tipo booleano |
| object | El | Tipo objeto de Python |
| string | S | Tipo *string* |
| unicode | U | Tipo Unicode |

Podemos hacer algunas observaciones sobre las informaciones de la tabla de arriba. En el caso de los enteros tenemos dos tipos distintos que son los `int` y `uint`. Los `int` consideran los signos de los números y los `uint` (*unsigned integer*) no consideran los signos, o sea, no representan números negativos. Para ejemplificar la diferencia vamos a utilizar los enteros de 8 bits que pueden codificar 256

números (n bits disponibles pueden codificar $2^n$ números). Con eso deducimos que los enteros de 8 bits ( `int8` ) representan los números de -128 hasta 127 y los enteros sin signo de 8 bits ( `uint8` ) representan los números de 0 hasta 255.

Para los números de punto flotante tenemos cuatro tipos de precisión diferentes, siendo que el `float64` es el tipo compatible con el objeto `float` de Python que aprendimos en el Capítulo 1 del curso.

Los dos últimos tipos `string` y `unicode` tienen tamaños fijos y para que creemos un `dtype` con tamaño 15, por ejemplo, debemos utilizar 'S15' como código para *string* y 'U15' para unicode.

```
np.array(['AB', 'CD'], dtype='U2')
```

**Salida:**

```
array(['AB', 'CD'], dtype='<U2')
```

## 6.5 OPERACIONES ARITMÉTICAS CON ARRAYS NUMPY

Un recurso importante en la utilización de *arrays* NumPy es la posibilidad de realizar operaciones aritméticas rápidas sin utilizar ningún bucle `for` . Veremos en esta sección la aplicación de este recurso en *arrays* de misma dimensión y en *arrays* de dimensiones diferentes (*broadcasting*).

### Operaciones aritméticas entre *arrays* de misma dimensión

Operaciones aritméticas aplicadas entre *arrays* de misma dimensión serán replicadas para todos los elementos de estos *arrays*. Esa forma de cálculo es conocida como vectorización en la biblioteca NumPy y veremos algunos ejemplos utilizando *arrays* de una y dos dimensiones.

#### *Arrays* con una dimensión

Vamos a suponer que estamos trabajando con informaciones sobre el mercado inmobiliario y necesitamos generar contenido con los datos que tenemos disponibles. Considere que tenemos las informaciones sobre el precio de venta de un conjunto de cinco inmuebles en una lista llamada **`valor`** y sus respectivas superficies (em $m^2$) en una lista llamada **`m2`** .

```
valor = [450000, 1500000, 280000, 650000, 325000]
m2 = [90, 150, 70, 100, 65]
```

Una información interesante que puede ser retirada de estos dos datos es el valor del metro cuadrado para cada inmueble. Esa información es fácilmente obtenida a partir de la razón entre valor y metro cuadrado, o sea, dividiendo el valor de venta de cada inmueble por su respectiva superficie o área. Para obtener el valor del metro cuadrado de cada inmueble utilizando listas sería necesario un bucle `for` para realizar esa operación de división en cada ítem de las dos listas y almacenar el resultado en una nueva lista.

```
valor_m2 = []
for valor, m2 in zip(valor, m2):
    valor_m2.append(valor / m2)
valor_m2
```

**Salida:**

```
[5000.0, 10000.0, 4000.0, 6500.0, 5000.0]
```

Ahora vamos a crear dos *arrays* NumPy para sustituir las listas creadas anteriormente.

```
valor = np.array([450000, 1500000, 280000, 650000, 325000])
m2 = np.array([90, 150, 70, 100, 65])
```

Este mismo procedimiento de cálculo del valor del metro cuadrado utilizando *arrays* NumPy puede ser realizado con apenas una línea de código.

```
valor_m2 = valor / m2
valor_m2
```

**Salida:**

```
array([ 5000., 10000.,  4000.,  6500.,  5000.])
```

Esto muestra que la operación aritmética entre los *arrays* **`valor`** y **`m2`** fue aplicada en todos los elementos sin utilizar ningún bucle `for`.

#### *Arrays* de dos dimensiones

Considere ahora que tenemos informaciones sobre el valor de venta de cada inmueble, el valor de locación de cada inmueble, las tasas incidentes sobre la venta y las tasas incidentes sobre la locación. Todas estas informaciones organizadas en *arrays* NumPy conforme el código de abajo:

```
venta = np.array([450000, 1500000, 280000, 650000, 325000])
alquiler = np.array([1500, 2000, 1800, 2500, 1250])
tasa_venda = np.array([13500, 51000, 11200, 22750, 12350])
tasa_alquiler = np.array([900, 1000, 1100, 850, 950])
```

Suponga también que las informaciones de valores y tasas sean agrupadas en dos *arrays* bidimensionales ( **`valores`** y **`tasas`** ).

```
valores = np.array([venta, alquiler])
valores
```

**Salida:**

```
array([[ 450000, 1500000,  280000,  650000,  325000],
       [   1500,    2000,    1800,    2500,    1250]])
```

```
tasas = np.array([tasa_venda, tasa_alquiler])
tasas
```

**Salida:**

```
array([[13500, 51000, 11200, 22750, 12350],
       [  900,  1000,  1100,   850,   950]])
```

El objetivo aquí es obtener un nuevo *array* llamado **`total`** con el resultado de las sumas entre los valores y sus respectivas tasas, o sea, el nuevo *array* debe tener dos líneas, y la primera línea debe contener el resultado de la suma entre el valor de venta y tasas incidentes sobre la venta y en la segunda línea el valor de locación más las tasas incidentes sobre a locación.

```
total = valores + tasas
total
```

**Salida:**

```
array([[ 463500, 1551000,  291200,  672750,  337350],
       [   2400,    3000,    2900,    3350,    2200]])
```

Note que funciona de la misma forma con *arrays* de dos dimensiones. El primer elemento del *array* **`valores`** es sumado con el primer elemento del *array* **`tasas`** y el resultado de esa suma es colocado en el primer elemento del *array* **`total`** . Este proceso es repetido para todos los elementos de los *arrays* involucrados, como ilustrado abajo.

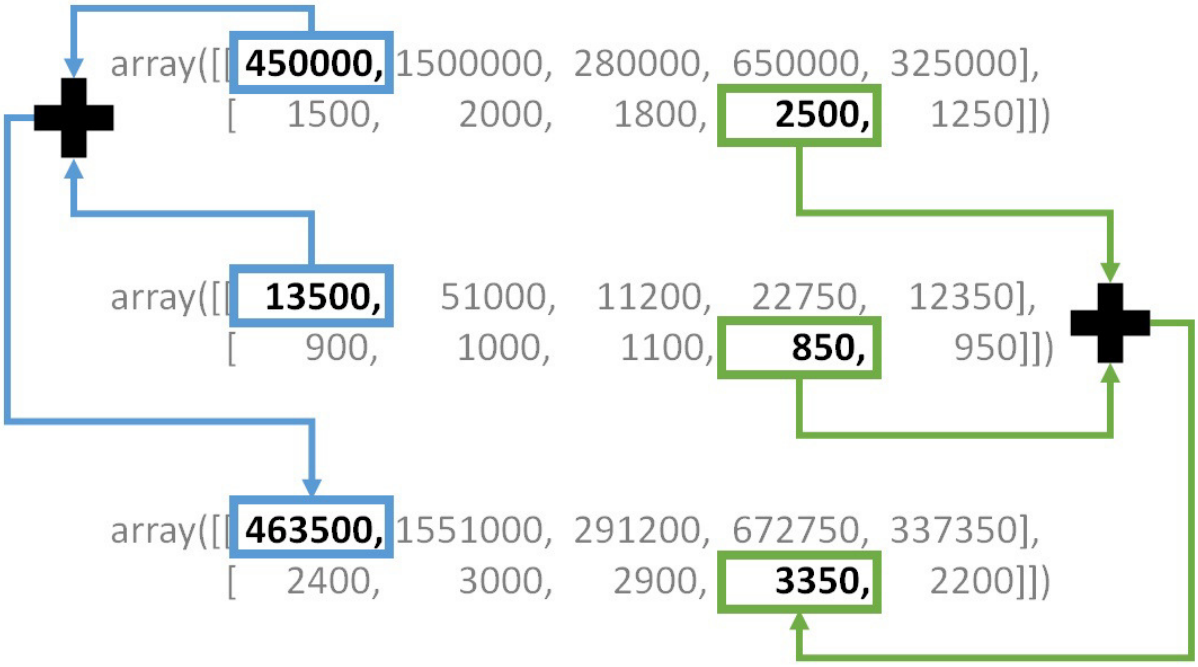



Figura 6.1: Operación Aritmética entre Arrays de misma Dimensión

## Operaciones aritméticas entre *arrays* de dimensiones diferentes - *Broadcasting*

El término *broadcasting* es utilizado para describir cómo la biblioteca NumPy trata *arrays* con *shapes* (dimensiones) diferentes durante operaciones aritméticas. Es un recurso bastante útil que permite, con algunas restricciones, que un *array* menor sea **"transmitido"** por el *array* mayor para que tengan dimensiones compatibles y así posibilitando las operaciones aritméticas entre ellos.

Puede ser un poco confuso en algunos casos, pero en nuestro curso no vamos a profundizar mucho en nuestra técnica. Usaremos apenas ejemplos simples, como la realización de una operación aritmética entre un *array* y una constante que es la forma más simple y utilizada de *broadcasting*.

```
numeros = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
```

Considerando el *array* **`numeros`** de arriba, suponga que sea necesario sumar el número 2 en todos los elementos de este *array*. Utilizando listas tendríamos que barrer la lista de números con un bucle `for` e ir haciendo la suma de la constante ítem a ítem. Con *arrays* NumPy basta ejecutar el siguiente código:

```
numeros = numeros + 2
numeros
```

**Salida:**

```
array([[ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

La operación de arriba es equivalente a crear un *array* con dimensiones (3, 3), llenar todos los elementos con el número 2 y después realizar la sumatoria, como puede ser visto en la figura abajo.

Lo que podemos decir en este caso es que hicimos el *broadcast* (transmisión) de la constante 2 para todos los elementos del *array* **`numeros`** en la operación de suma.

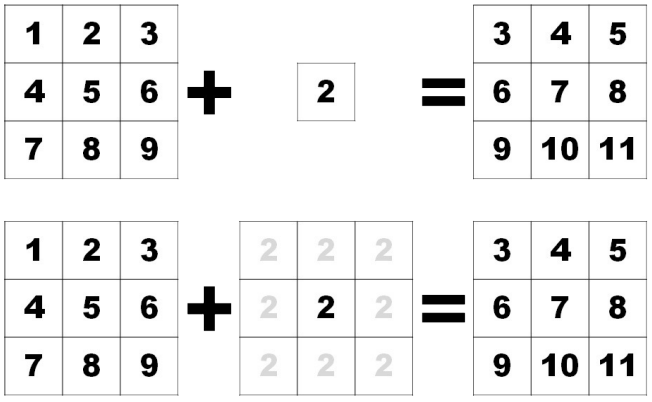



Figura 6.2: Broadcast entre Arrays y Constantes

Ahora vamos a crear una ejemplo donde necesitamos realizar una operación aritmética entre dos *arrays* de dimensiones diferentes. Para esto considere el nuevo *array* **`numeros`** definido en el código de abajo.

```
numeros = np.array([
    [1, 2],
    [3, 4],
```

```
    [5, 6]
])
```

Una operación bastante útil, y que hace parte del cálculo de estadísticas descriptivas como la varianza y la desviación estándar, es el cálculo de los desvíos en relación a la media. Para realizar este ejercicio vamos a asumir que el *array* **`numeros`** tiene dos conjuntos de datos distintos y están representados por las dos columnas del *array*, o sea, el primer conjunto de datos compuesto por los valores 1, 3 y 5, y el segundo conjunto de datos compuesto por los valores 2, 4 y 6. Vamos a considerar también el *array* **`medias`** que contiene los valores de las medias aritméticas de cada conjunto de datos. En las próximas secciones de este curso vamos a aprender a obtener las medias con *arrays* NumPy utilizando funciones propias de la biblioteca.

```
medias = np.array([3, 4])
medias
```

**Salida:**

```
array([3, 4])
```

Para obtener los desvíos en relación a la media es necesario substraer cada elemento de una variable por la media de esta variable. En nuestro caso sería el siguiente conjunto de operaciones:

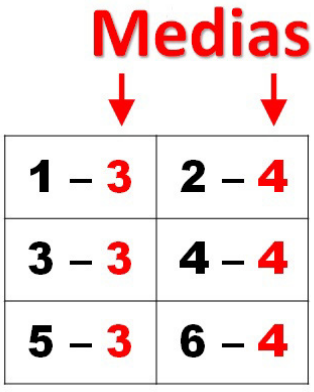



Figura 6.3: Conjunto de Operaciones

El *broadcasting* con *arrays* de dimensiones diferentes nos permite realizar esas operaciones de forma bastante simple.

```
desvios = numeros - medias
desvios
```

**Salida:**

```
array([[-2., -2.],
       [ 0.,  0.],
       [ 2.,  2.]])
```

La figura de abajo ilustra el procedimiento de *broadcasting* para este ejemplo.

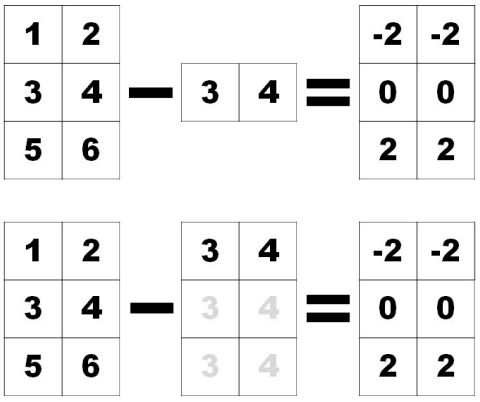



Figura 6.4: Procedimento de Broadcasting

El próximo ejemplo es similar al anterior solo que ahora vamos a tener el *array* **`numeros`** con el *shape* diferente. Nuevamente tenemos dos conjuntos de datos distintos solo que ahora ellos están representados por las dos líneas del *array*, o sea, el primer conjunto de datos es compuesto por los valores 1, 3 y 5, y el segundo conjunto de datos es compuesto por los valores 2, 4 y 6.

```
numeros = np.array([
    [1, 3, 5],
    [2, 4, 6]
])
```

Utilizando el mismo *array* de medias para los dos conjuntos de datos.

```
medias = np.array([3, 4])
medias
```

**Salida:**

```
array([3, 4])
```

Al ejecutar la operación aritmética entre el nuevo *array* **`numeros`** y el *array* **medias** vamos a tener un `ValueError`. Este error nos informa que no fue posible realizar el *broadcast* con los *arrays* **`numeros`** y **`medias`**.

```
desvios = numeros - medias
desvios
```

**Salida:**

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-146-5c3c7e0ace73> in <module>()
----> 1 desvios = numeros - medias
    2 desvios

ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

Esto ocurrió porque no respetamos las reglas de *broadcasting*, que pueden ser resumidas de la siguiente forma:

Para verificar si dos *arrays* son compatibles para una operación de *broadcasting*, NumPy compara sus dimensiones (*shapes*). La comparación comienza con las dimensiones finales. Dos dimensiones son compatibles para *broadcasting* cuando ellas son iguales o una de ellas es igual a 1 o ausente. El *broadcasting* será realizado en la dimensión ausente o de tamaño 1.

Vamos a ilustrar algunos ejemplos para dejar más claras estas reglas. En la imagen de abajo suponga que A y B son dos *arrays* de máximo dos dimensiones. La dimensión ausente, en caso de *arrays* unidimensionales, será representada por el número cero, o sea, un *array* con *shape* (2,) será representado en la ilustración por el *shape* (0, 2) indicando que el eje de las líneas está ausente en este *array*. El eje 0 es el eje de las líneas y el eje 1 es el de las columnas.

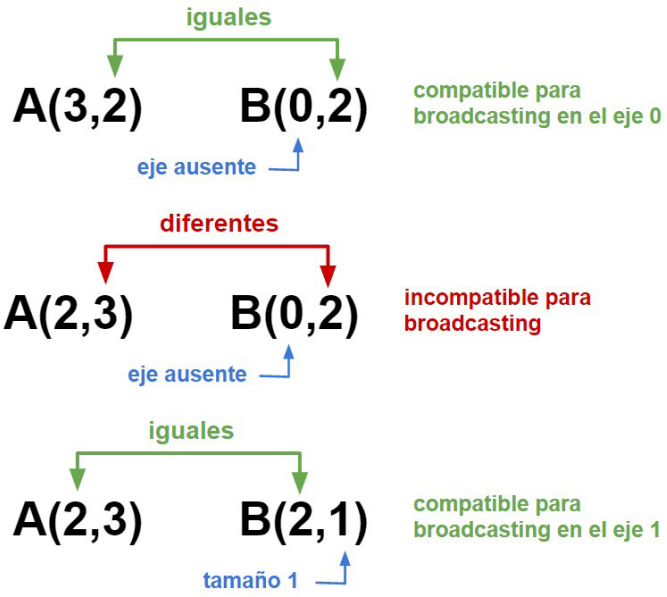



Figura 6.5: Entendendo las Regras de Broadcasting

De acuerdo con las reglas de *broadcasting* vamos a necesitar modificar el *shape* del nuestro *array* de medias para hacer los *arrays* **numeros** y **medias** compatibles. El siguiente código genera un nuevo *array* **`medias`** con dimensiones (2, 1).

```
medias = np.array([[3], [4]])
medias
```

**Salida:**

```
array([[3],
       [4]])
```

```
print(numeros.shape, medias.shape)
```

**Salida:**

```
(2, 3) (2, 1)
```

Ahora tenemos una igualdad en las dimensiones del eje 0 de ambos los *arrays* y también tenemos una dimensión 1 en el eje 1 del *array* **`medias`** . Eso indica compatibilidad para el *broadcasting* que será efectuado en el eje 1 (ver figura más abajo).

```
desvios = numeros - medias
desvios
```

**Salida:**

```
array([[-2.,  0.,  2.],
       [-2.,  0.,  2.]])
```

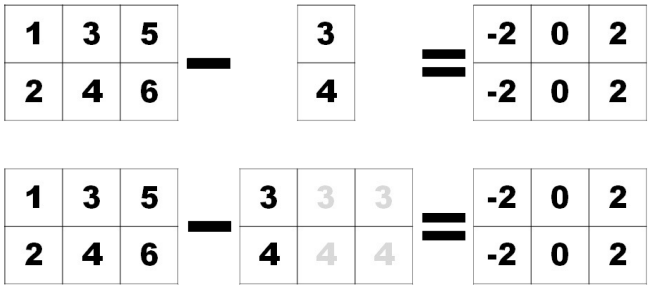



Figura 6.6: Procedimento de Broadcasting

## 6.6 PARA SABER MÁS...

**`numpy.nan_to_num`**

**Documentación:** https://numpy.org/doc/1.18/reference/generated/numpy.nan_to_num.html

Este procedimiento sustituye valores NaN por cero o por el valor definido por el usuario en el argumento `nan` . Considere el *array* de ejemplo de abajo:

```
ejemplo = np.array([1, 2, 3, np.nan, 5])
ejemplo
```

**Salida:**

```
array([ 1.,  2.,  3., nan,  5.])
```

En el código de arriba creamos un *array* con uno de los elementos `np.nan` , es una representación de punto flotante de *Not a Number*.

Para sustituir este valor por un número basta utilizar el método `np.nan_to_num` de la siguiente forma:

```
np.nan_to_num(ejemplo)
```

**Salida:**

```
array([1., 2., 3., 0., 5.])
```

Este método no modifica el *array in-place* y para hacer eso necesitamos utilizar el argumento `copy` y configurarlo con *False*. Podemos también escoger otro valor para sustituir los `nan` del *array*. El valor *default* del método es 0, pero configurando el argumento `nan` es posible escoger un valor diferente.

```
np.nan_to_num(ejemplo, nan=9, copy=False)
```

**Salida:**

```
array([1., 2., 3., 9., 5.])
```

## 6.7 EJERCICIOS

La celda de abajo tiene la construcción de dos listas con las informaciones de kilometraje recorrido **`lista_km`** y año de fabricación **`lista_anhos`** de un conjunto de vehículos. Considere estas informaciones para solucionar los ejercicios:

```
lista_km = [44410., 5712., 37123., 0., 25757.]
lista_anhos = [2003, 1991, 1990, 2020, 2006]
```

1. Crea dos *arrays* a partir de las listas **`lista_km`** y **`lista_anhos`**. Llama los nuevos *arrays* de **`km`** y **`anhos`**.

   **Respuesta:**

   ```
   km = np.array(lista_km)
   anhos = np.array(lista_anhos)
   ```

2. Crea un *array* NumPy que represente la edad (en años) de los vehículos y llama este *array* de **`edades`**

3. . Considere **2020** como año corriente.

   **Respuesta:**

   ```
   edades = 2020 - anhos
   edades
   ```

   **Salida:**

   ```
   array([17, 29, 30,  0, 14])
   ```

4. Crea un *array* NumPy que represente el kilometraje medio anual de cada vehículo, utiliza la fórmula de abajo para realizar los cálculos. Llama este nuevo *array* de **`km_medias`**.

$$km_{medio} = \frac{km_{total}}{(año_{actual} - año_{fabricacion})}$$

Figura 6.7: Fórmula

**Respuesta:**

```
km_medias = km / edades
km_medias
```

**Salida:**

```
array([2612.35294118,  196.96551724, 1237.43333333, nan, 1839.78571429])
```

```
np.around(km_medias, 2)
```

**Salida:**

```
array([2612.35,  196.97, 1237.43, nan, 1839.79]
```

5. Utilizando los consejos de la última sección **para saber más**, sustituya el valor `nan` del *array* **`km_medias`** por el valor 0 (cero).

**Respuesta:**

```
np.nan_to_num(np.around(km_medias, 2), copy=False)
```

**Salida:**

```
array([2612.35,  196.97, 1237.43, 0., 1839.79])
```

## 6.8 SELECCIONES CON ARRAYS NUMPY

En esta sección vamos a hablar sobre indexación y selección con *arrays* NumPy. Tratamos sobre este asunto cuando trabajamos con secuencias (listas, tuplas, etc.) que, en este sentido, tienen bastante similaridad con *arrays* NumPy.

Para facilitar nuestro entendimiento vamos a utilizar un *array* bidimensional conteniendo *strings* que representan letras y números.

```
letras = np.array(['A', 'B', 'C', 'D', 'E'])
numeros = np.arange(1, 6)
datos = np.array([letras, numeros])
datos
```

**Salida:**

```
array([['A', 'B', 'C', 'D', 'E'],
       ['1', '2', '3', '4', '5']], dtype='<U21')
```

## Indexación

Así como en las secuencias integradas de Python, la indexación en *arrays* NumPy tiene origen en el cero. La figura de abajo nos ayuda a entender la indexación en nuestro *array* bidimensional de ejemplo.

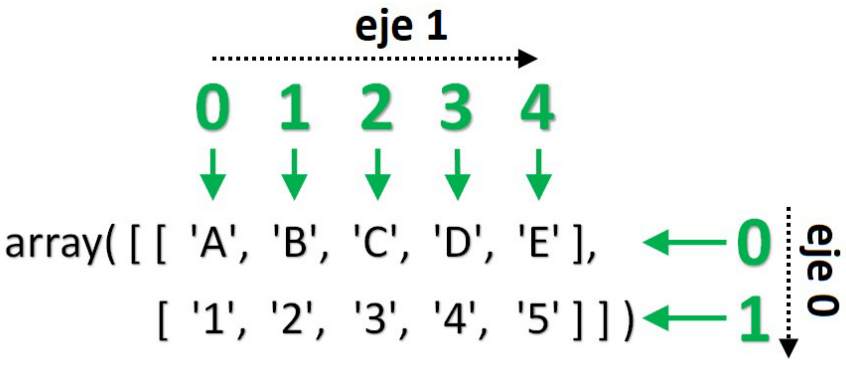



Figura 6.8: Indexación

En un *array* unidimensional el proceso de selección de ítems es bastante simple, bastando indicar el índice del elemento que se quiere seleccionar entre corchetes luego después del nombre del *array*. El ejemplo de abajo muestra cómo seleccionar el primer elemento del *array* **`letras`** .

```
letras[0]
```

**Salida:**

```
'A'
```

El uso de índices negativos también es permitido en las selecciones con *arrays* NumPy. Recordando que para seleccionar el último elemento de un *array* utilizamos el índice -1, para seleccionar el penúltimo elemento utilizamos el índice -2 y así sucesivamente. Abajo veremos un ejemplo de selección del último elemento del *array* **`letras`** .

```
letras[-1]
```

**Salida:**

```
'E'
```

Para selecciones en arrays bidimensionales necesitamos realizar un paso extra, pero de fácil entendimiento. Eso porque cuando accedemos a un array bidimensional utilizando un único índice estamos accediendo a las informaciones del eje 0 (ver figura de arriba), o sea, estamos accediendo a arrays unidimensionales.

```
datos[0]
```

**Salida:**

```
array(['A', 'B', 'C', 'D', 'E'], dtype='<U21')
```

```
datos[1]
```

**Salida:**

```
array(['1', '2', '3', '4', '5'], dtype='<U21')
```

Para acceder a un ítem específico dentro de un array bidimensional podemos proceder de dos formas.

```
datos[1][2]
```

**Salida:**

```
'3'
```

```
datos[1, 2]
```

**Salida:**

```
'3'
```

En este tipo de indexación, primero pasamos el índice del eje 0 (líneas) y después el índice del eje 1 (columnas).

**SELECCIÓN EN ARRAYS BIDIMENSIONALES:**

```
array[ línea ][ columna ]
```

o

```
array[ línea, columna ]
```

## Slices

En los *slices* seleccionamos trechos de *arrays* con el uso de índices. La sintaxis para realizar un *slice* en un *array* NumPy unidimensional es `array[i : j : k]` donde `i` es el índice inicial, `j` es el índice de parada, y `k` es el indicador de paso ( $k \neq 0$ ). Importante también saber que en los *slices* el ítem con índice `i` es **incluido** y el ítem con índice `j` **no es incluido** en el resultado.

Vamos a verificar algunos ejemplos de slices con *arrays* unidimensionais. El código de abajo selecciona los elementos de índices 1, 2 y 3 del *array* **`letras`** . Note que, como fue observado más arriba, el elemento del índice inicial en el resultado de los *slices* y el elemento del índice de parada no entra en el resultado.

```
letras[1:4]
```

**Salida:**

```
array(['B', 'C', 'D'], dtype='<U1')
```

Caso estemos interesados en hacer un *slice* que incluya desde el primer elemento del *array* hasta determinado punto de parada `j`, basta suprimir el primer índice en el *slice*. El *slice* tendría el siguiente código: `array[:j]`.

```
letras[:4]
```

**Salida:**

```
array(['A', 'B', 'C', 'D'], dtype='<U1')
```

El mismo puede ser hecho caso estemos interesados en hacer *slices* a partir de determinado punto de partida `i` hasta el final del *array*. Para eso basta suprimir el índice de parada del *slice*. El código tendría la siguiente forma: `array[i:]`.

```
letras[1:]
```

**Salida:**

```
array(['B', 'C', 'D', 'E'], dtype='<U1')
```

Suprimiendo los índices inicial y de parada tenemos como retorno el *array* completo.

```
letras[:]
```

**Salida:**

```
array(['A', 'B', 'C', 'D', 'E'], dtype='<U1')
```

Ahora imagina que necesitamos hacer una selección de elementos que no están puestos de forma continua en un *array*. Por ejemplo, suponga que necesitamos seleccionar los valores pares o impares de una secuencias de números. Para eso utilizamos, en la notación patrón, el indicador de paso, que es un número que informa cuántas posiciones debemos saltar entre una selección y otra. Observa el ejemplo de abajo que muestra cómo seleccionar solamente los números impares del *array* **`numeros`**.

```
numeros[::2]
```

**Salida:**

```
array([1, 3, 5])
```

Note que como queremos todos los números impares, no fue necesario informar el índice de partida y ni el índice de parada, también podríamos haber utilizado el código `numeros[0:5:2]` para obtener el mismo resultado.

Para seleccionar solamente los números pares del *array* `numeros` el código sería el siguiente:

```
numeros[1::2]
```

**Salida:**

```
array([2, 4])
```

En los códigos de arriba estamos haciendo una selección con intervalos de tamaño 2 (indicador de paso), teniendo como única diferencia el punto de partida. En el ejemplo de los números impares comenzamos la selección en el índice 0 y en el ejemplo de los números pares comenzamos en el índice 1.

La idea es la misma en *arrays* bidimensionales, salvo algunas pequeñas diferencias por estar trabajando con dos ejes. Observa el ejemplo de abajo.

```
datos[:, 1:3]
```

**Salida:**

```
array([['B', 'C'],
       ['2', '3']], dtype='<U21')
```

El primer punto que debe quedar claro es que las notaciones `array[ línea ][ columna ]` y `array[ línea, columna ]` no son equivalentes cuando estamos haciendo *slices* en *arrays* bidimensionales. Ellos sólo son equivalentes en la selección de valores únicos en un *array* bidimensional.

En *arrays* bidimensionales usamos la notación `array[ i : j: k, x : y : z]` donde `i` y `x` son los índices iniciales, `j` y `y` son los índices de parada, y `k` y `z` son los indicadores de paso ( `k` y `z` $\neq$ `0` ) para los ejes 0 y 1 respectivamente, como mostrado en el ejemplo de arriba.

En este ejemplo estamos informando que queremos en nuestra selección todos los elementos del eje 0 ([:]) y apenas los elementos con índices 1 y 2 del eje 1 ([1:3]).

Regresando al ejemplo de los números pares, vamos a aplicar *slice* al *array* **`datos`** para obtener una visualización de los elementos que están en las columnas con números pares.

```
datos[:, 1::2]
```

**Salida:**

```
array([['B', 'D'],
       ['2', '4']], dtype='<U21')
```

La misma idea del ejemplo unidimensional solo que aquí fue necesario informar que queríamos todos los elementos del eje 0 (líneas) en nuestra visualización.

## Indexación con *array* booleano

Del mismo modo que realizamos operaciones aritméticas entre *arrays*, las operaciones de

comparación entre ellos también son posibles (==, >, >=, <, <=, !=). La diferencia es que en estas operaciones de comparación tendremos como resultado un *array* booleano. Sigue un ejemplo simples:

```
contador = np.arange(5)
contador
```

**Salida:**

```
array([0, 1, 2, 3, 4])
```

```
contador >= 2
```

**Salida:**

```
array([False, False, True, True, True])
```

Note que este procedimiento ya fue tratado en este capítulo cuando hablamos sobre *broadcasting*. Aquí tenemos el mismo procedimiento solo que ahora estamos utilizando operadores de comparación y no operadores aritméticos.

Lo interesante de tener como respuesta *arrays* booleanos es que podemos utilizar estes *arrays* cuando estemos indexando otro *array*. Regresando al ejemplo que acabamos de ver. Suponga que necesitamos apenas los ítems mayores o iguales a 2 del *array* **`contador`** . Para eso basta crear el siguiente código:

```
contador[[False, False,  True,  True,  True]]
```

**Salida:**

```
array([2, 3, 4])
```

O de una forma más simple:

```
contador[contador >= 2]
```

**Salida:**

```
array([2, 3, 4])
```

En *arrays* bidimensionales funciona de la misma forma que aprendimos en la sección anterior, pudiendo hasta mezclar *arrays* booleanos e índices enteros. Podemos también asignar el *array* booleano en una nueva variable y pasar esta variable como índice.

```
select = datos == 'A'
select
```

**Salida:**

```
array([[ True, False, False, False, False],
       [False, False, False, False, False]])
```

```
datos[select]
```

**Salida:**

```
array(['A'], dtype='<U21')
```

Podemos utilizar operadores lógicos para hacer las selecciones más complejas. Los operadores lógicos `and`, `or` y `not` de Python no funcionan con *arrays* booleanos, para eso utilizamos `&` (`and`), `|` (`or`) y `~` (`not`). En este caso las expresiones antes y después de los operadores lógicos deben estar siempre entre paréntesis, como en la primera línea del código a seguir.

```
select = (datos == 'A') | (datos == 'D') | (datos == '2')
select
```

**Salida:**

```
array([[ True, False, False,  True, False],
       [False,  True, False, False, False]])
```

```
datos[select]
```

**Salida:**

```
array(['A', 'D', '2'], dtype='<U21')
```

```
datos[~select]
```

**Salida:**

```
array(['B', 'C', 'E', '1', '3', '4', '5'], dtype='<U21')
```

Para tener como retorno un *array* en el formato bidimensional necesitamos pasar las informaciones de *slices* por eje, como fue hecho en la sección anterior.

Vamos a suponer que necesitamos seleccionar las columnas que contienen los elemento 'A', 'D' y '2'. Observa las diferencias entre el código anterior y el código de abajo.

```
select = (datos[0] == 'A') | (datos[0] == 'D') | (datos[1] == '2')
select
```

**Salida:**

```
array([ True,  True, False,  True, False])
```

```
datos[:, select]
```

**Salida:**

```
array([['A', 'B', 'D'],
       ['1', '2', '4']], dtype='<U21')
```

```
datos[:, ~select]
```

**Salida:**

```
array([['C', 'E'],
       ['3', '5']], dtype='<U21')
```

Notamos que el *array* booleano ahora es unidimensional y debe ser pasado como índice del eje 1 (columnas). Y aquí cabe una observación importante: **El *array* booleano debe tener el mismo tamaño**

**del eje del *array* que está siendo indexado**. En nuestro ejemplo tenemos un *array* booleano con cinco elementos y estamos indexando el eje 1 que también tiene cinco elementos.

Otro punto que debe ser notado es que el código de selección ahora informa el *slice* para el eje 0 (todas las líneas) y para el eje 1 (array booleano).

Como ejemplo final de esta sección vamos a crear un *slice* del *array* **`datos`** para obtener una visualización de los elementos que están en las columnas con *strings* que representen números pares, y vamos a hacer eso con *arrays* booleanos. Para eso podemos utilizar el *array* **`numeros`** que está en el formato numérico para generar el *array* booleano que necesitamos.

```
select = numeros % 2 == 0
select
```

**Salida:**

```
array([False, True, False, True, False])
```

Ahora basta hacer la selección considerando todos los elementos del eje 0, o sea, todas las líneas. Recordando que el operador % retorna el resto de la división entre dos números.

```
datos[:, select]
```

**Salida:**

```
array([['B', 'D'],
       ['2', '4']], dtype='<U21')
```

O de forma resumida:

```
datos[:, numeros % 2 == 0]
```

**Salida:**

```
array([['B', 'D'],
       ['2', '4']], dtype='<U21')
```

## 6.9 EJERCICIOS

1. En el ejercicio anterior calculamos el kilometraje medio anual para un conjunto de vehículos utilizando la combinación entre *arrays* unidimensionales. Haga el mismo cálculo utilizando el *array* bidimensional de abajo:

```
km = np.array([44410., 5712., 37123., 0., 25757.])
anhos = np.array([2003, 1991, 1990, 2020, 2006])
datos = np.array([km, anhos])
datos
```

**Salida:**

```
array([[44410.,  5712., 37123.,     0., 25757.],
    [ 2003.,  1991.,  1990.,  2020.,  2006.]])
```

**Respuesta:**

```
km_medias = np.around(datos[0] / (2020 - datos[1]), 2)
km_medias
```

**Salida:**

```
array([2612.35, 196.97, 1237.43, nan, 1839.79])
```

2. A partir del *array* **datos**, crea un *slice* considerando solamente los vehículos fabricados después del año 2000.

**Respuesta:**

```
datos[:, datos[1] > 2000]
```

**Salida:**

```
array([[44410.,     0., 25757.],
    [ 2003.,  2020.,  2006.]])
```

3. A partir del *array* **datos**, crea un *slice* considerando solamente los vehículos con kilometraje medio anual inferior a 1500 km.

**Respuesta:**

```
km_medias < 1500
```

**Salida:**

```
array([False,  True,  True, False, False])
```

```
np.nan_to_num(km_medias, copy=False)
```

**Salida:**

```
array([2612.35,  196.97, 1237.43,    0.  , 1839.79])
```

```
km_medias < 1500
```

**Salida:**

```
array([False,  True,  True,  True, False])
```

```
datos[:, km_medias < 1500]
```

**Salida:**

```
array([[ 5712., 37123.,     0.],
    [ 1991.,  1990.,  2020.]])
```

# 6.10 ATRIBUTOS Y MÉTODOS DE ARRAYS NUMPY

## Atributos

**Documentación:** https://numpy.org/doc/1.18/reference/arrays.ndarray.html#array-attributes

Los atributos de un *array* posibilitan acceder a informaciones intrínsecas del mismo. En esta sección vamos a conocer los principales atributos de los *arrays* NumPy.

Para ayudar en los ejemplos considere los *arrays* **`letra`** , **`numeros`** y **`datos`** a seguir:

```
letras = np.array(['A', 'B', 'C', 'D', 'E'])
numeros = np.arange(1, 6)
datos = np.array([letras, numeros])
datos
```

**Salida:**

```
array([['A', 'B', 'C', 'D', 'E'],
       ['1', '2', '3', '4', '5']], dtype='<U21')
```

**`ndarray.shape`**

Retorna una tupla con las dimensiones del *array*. El atributo `shape` es utilizado para obtener la forma de un *array*, pero también puede ser utilizado para remodelar un *array in-place*, asignándole una tupla con las nuevas dimensiones.

```
datos.shape
```

**Salida:**

```
(2, 5)
```

Como vimos en las secciones anteriores, el resultado muestra que el *array* **datos** tiene dos dimensiones y que la primera dimensión (eje 0 o líneas) tiene tamaño 2 y la segunda dimensión (eje 1 o columnas) tiene tamaño 5.

El procedimiento de ajuste de las dimensión utilizando la modificación del atributo `shape` necesita que la nueva tupla informada genere un *array* con el mismo número de elementos del *array* de origen. Por ejemplo, el *array* **`datos`** tiene un *shape* igual a (2, 5), indicando que este *array* tiene un total de 10 elementos (2 X 5 = 10). De esta forma para remodelar el *array* **`datos`** utilizando el atributo `shape` será necesario un *shape* que garantice exactamente 10 elementos para el nuevo formato, ejemplos: (1, 10), (10, 1), (5, 2) o (10,).

```
datos.shape = (1, 10)
datos
```

**Salida:**

```
array([['A', 'B', 'C', 'D', 'E', '1', '2', '3', '4', '5']], dtype='<U21')
```

```
datos.shape = (5, 2)
datos
```

**Salida:**

```
array([['A', 'B'],
       ['C', 'D'],
       ['E', '1'],
```

```
       ['2', '3'],
       ['4', '5']], dtype='<U21')
```

```
datos.shape = (2, 5)
datos
```

**Salida:**

```
array([['A', 'B', 'C', 'D', 'E'],
       ['1', '2', '3', '4', '5']], dtype='<U21')
```

**`ndarray.ndim`**

Retorna el número de dimensiones del *array*.

```
datos.ndim
```

**Salida:**

```
2
```

**`ndarray.size`**

Retorna el número de elementos del *array*.

```
datos.size
```

**Salida:**

```
10
```

**`ndarray.dtype`**

Retorna el tipo de dato de los elementos del *array*.

```
letras.dtype
```

**Salida:**

```
dtype('<U1')
```

```
numeros.dtype
```

**Salida:**

```
dtype('int64')
```

```
datos.dtype
```

**Salida:**

```
dtype('<U21')
```

**`ndarray.T`**

Retorna el *array* transpuesto, o sea, convierte líneas en columnas y viceversa. No modifica el *array in-place*.

```
datos.T
```

**Salida:**

```
array([['A', '1'],
       ['B', '2'],
       ['C', '3'],
       ['D', '4'],
       ['E', '5']], dtype='<U21')
```

```
datos # El array no sufrió modificación
```

**Salida:**

```
array([['A', 'B', 'C', 'D', 'E'],
       ['1', '2', '3', '4', '5']], dtype='<U21')
```

El mismo resultado puede ser obtenido con el método `transpose()`.

```
datos.transpose()
```

**Salida:**

```
array([['A', '1'],
       ['B', '2'],
       ['C', '3'],
       ['D', '4'],
       ['E', '5']], dtype='<U21')
```

## Métodos

**Documentación:** https://numpy.org/doc/1.18/reference/arrays.ndarray.html#array-methods

El objeto *array* del NumPy posee un conjunto bien amplio de métodos. Son métodos para la conversión, manipulación, cálculo, etc. En esta sección vamos a conocer algunos de uso más común, dejando los métodos de cálculo para la próxima sesión del curso.

**`ndarray.tolist()`**

Retorna una copia de los datos de un *array* como una lista de Python. Los datos del *array* son convertidos para el tipo de Python compatible.

```
datos.tolist()
```

**Salida:**

```
[['A', 'B', 'C', 'D', 'E'], ['1', '2', '3', '4', '5']]
```

Nota que como **`datos`** es un *array* bidimensional, el retorno del método `tolist()` es una lista anidada, o sea, una lista que contiene listas como ítems.

**`ndarray.reshape(shape, order='C')`**

Altera la forma de un *array* sin alterar sus datos. Considere el *array* **`contador`** a seguir para nuestros ejemplos.

```
contador = np.arange(15)
contador
```

**Salida:**

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Para utilizar el método `reshape()` basta pasar como argumento el nuevo *shape* del array en formato de tupla o los elementos del *shape* como argumentos separados.

```
contador.reshape((5, 3))
```

**Salida:**

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

```
contador.reshape(5, 3)
```

**Salida:**

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

El método `reshape()` posee también el argumento `order` que es opcional y define si los datos serán orientados a las líneas (orden C) o a las columnas (orden Fortran). Eso permite controlar la forma como los datos del *array* quedan organizados en la memoria. *Arrays* NumPy son creados, por patrón, en una orden orientada a líneas y esta es la opción *default* del método `reshape()` .

```
contador.reshape((5, 3), order='C') # Opción default
```

**Salida:**

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

```
contador.reshape((5, 3), order='F')
```

**Salida:**

```
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

**`ndarray.resize(new_shape, refcheck=True)`**

Altera la forma y el tamaño del *array in-place*.

```
datos.resize((3, 5))
```

**Salida:**

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-58-0c0fabdc93b4> in <module>()
----> 1 datos.resize((3, 5))

ValueError: cannot resize an array that references or is referenced
by another array in this way.
Use the np.resize function or refcheck=False
```

El método `resize()` posee el argumento opcional `refcheck` que es un booleano que indica si la verificación del conteo de referencia debe ser hecha o no. El objetivo es verificar si el *buffer* del *array* está relacionado a cualquier otro objeto. Por patrón, es definido como *True*.

Sin embargo, los conteos de referencia pueden aumentar por otros motivos. Por tanto, si tienes seguridad de que la memoria del *array* no comparte memoria con otro objeto Python, podrás definir con seguridad el argumento `refcheck` como *False*, evitando así el error ocurrido en el código de arriba.

```
datos.resize((3, 5), refcheck=False)
datos
```

**Salida:**

```
array([['A', 'B', 'C', 'D', 'E'],
       ['1', '2', '3', '4', '5'],
       ['', '', '', '', '']], dtype='<U21')
```

Nota que el código arriba aumentó una de las dimensiones (eje 0 o líneas) del *array* **datos** y llenó este nuevo espacio con *strings* vacíos. Si el *array* original fuese llenado con datos numéricos el método llenaría los espacios vacíos con ceros, si fuesen datos booleanos los espacios vacíos serían llenados con *False*.

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

**Salida:**

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

```
contador.resize((3, 5), refcheck=False)
contador
```

**Salida:**

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

# 6.11 OBTENIDO ESTADÍSTICAS CON ARRAYS NUMPY

**Documentación:** https://numpy.org/doc/1.18/reference/routines.statistics.html

El paquete NumPy ofrece herramientas de estadística básica para auxiliar en procesos de pruebas y análisis. Considera el código de abajo para nuestros ejemplos.

```
dataset = np.arange(1, 16).reshape((5, 3))
dataset
```

**Salida:**

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12],
       [13, 14, 15]])
```

**`np.mean()`**

Retorna la media de los elementos del *array* a lo largo del eje especificado. Para obtener la media por las líneas de un *array* bidimensional debemos configurar el argumento `axis` como 0.

```
np.mean(dataset, axis = 0)
```

**Salida:**

```
array([7., 8., 9.])
```

Y si quisiéramos obtener la media por las columnas el argumento `axis` debe tener valor 1.

```
np.mean(dataset, axis = 1)
```

**Salida:**

```
array([ 2.,  5.,  8., 11., 14.])
```

**`np.std()`**

Retorna la desviación estándar de los elementos del *array* a lo largo del eje especificado. Para obtener la desviación estándar por las líneas de un *array* bidimensional debemos configurar el argumento `axis` como 0 y para obtener la desviación estándar por las columnas el argumento `axis` debe tener valor 1.

```
np.std(dataset, axis = 0)
```

**Salida:**

```
array([4.24264069, 4.24264069, 4.24264069])
```

```
np.std(dataset, axis = 1)
```

**Salida:**

```
array([0.81649658, 0.81649658, 0.81649658, 0.81649658, 0.81649658])
```

**`np.sum()`**

Retorna la suma de los elementos del *array* a lo largo del eje especificado. Para obtener la sumatoria por las líneas con un *array* bidimensional el argumento `axis` debe ser configurado como 0.

```
np.sum(dataset, axis = 0)
```

**Salida:**

```
array([35, 40, 45])
```

Para una sumatoria por columnas el argumento `axis` debe tener valor 1.

```
np.sum(dataset, axis = 1)
```

**Salida:**

```
array([ 6, 15, 24, 33, 42]  )
```

## 6.12 EJERCICIOS

En un procedimiento de *web scraping* (colectar datos de la web), donde necesitamos colectar datos de una página web de venta de vehículos, obtenemos un conjunto de datos en forma de lista (ver celda más abajo). Los datos representan las informaciones de cinco vehículos, donde los cinco primeros ítems de la lista son las informaciones sobre kilometraje total y los cinco últimos son las informaciones sobre año de fabricación.

```
km_anos = [44410., 5712., 37123., 0., 25757., 2003, 1991, 1990, 2020, 2006]
```

1. Crea un *array* NumPy a partir de la lista **`km_años`** con el formato especificado de abajo. Llama a este nuevo *array* de **`info_carros`**.

```
array([[44410.,  5712., 37123.,     0., 25757.],
       [ 2003.,  1991.,  1990.,  2020.,  2006.]])
```

**Respuesta:**

```
info_carros = np.array(km_años).reshape((2, 5))
info_carros
```

**Salida:**

```
array([[44410.,  5712., 37123.,     0., 25757.],
       [ 2003.,  1991.,  1990.,  2020.,  2006.]])
```

2. Verifica las dimensiones del *array* **`info_carros`**.

**Respuesta:**

```
info_carros.shape
```

**Salida:**

```
(2, 5)
```

3. Crea una copia del *array* **`info_carros`** y llama esta copia de **`info_carros_copia`**.

   **Respuesta:**

```
info_carros_copia = info_carros.copy()
```

4. Agrega una nueva "línea" en el *array* **`info_carros_copia`**, o sea, aumenta en 1 la primera dimensión del *array* **`info_carros_copia`**. El nuevo *shape* de **`info_carros_copia`** debe tener el siguiente formato: (3, 5).

   **Respuesta:**

```
info_carros_copia.resize((3, 5), refcheck=False)
info_carros_copia
```

   **Salida:**

```
array([[44410.,  5712., 37123.,     0., 25757.],
       [ 2003.,  1991.,  1990.,  2020.,  2006.],
       [    0.,     0.,     0.,     0.,     0.]])
```

5. Rellena la última línea del *array* **`info_carros_copia`** con los valores de kilometraje medio anual para cada vehículo.

   **Respuesta:**

```
info_carros_copia[2] = info_carros_copia[0] / (2020 - info_carros_copia[1])
info_carros_copia
```

   **Salida:**

```
array([[44410.        ,  5712.        , 37123.        ,     0.        ,        25757.        ],
       [ 2003.        ,  1991.        ,  1990.        ,  2020.        ,         2006.        ],
       [ 2612.35294118,   196.96551724,  1237.43333333,            nan,         1839.78571429]])
```

6. Redondea los valores obtenidos y sustituya los valores NaN por ceros.

   **Respuesta:**

```
info_carros_copia = np.nan_to_num(np.around(info_carros_copia, 2))
info_carros_copia
```

   **Salida:**

```
array([[44410.  ,  5712.  , 37123.  ,     0.  , 25757.  ],
       [ 2003.  ,  1991.  ,  1990.  ,  2020.  ,  2006.  ],
       [ 2612.35,   196.97,  1237.43,     0.  ,  1839.79]])
```

CAPÍTULO 7

# INTRODUCCIÓN A LA BIBLIOTECA PANDAS

## 7.1 ¿POR QUÉ USAR LA BIBLIOTECA PANDAS PARA DATA SCIENCE?

Proyectos en *Data Science* siempre envuelven algún tipo de conjunto de datos, y antes de que desarrollemos proyectos sofisticados en esta área (*machine learning*, regresión, clasificación, etc.), siempre tendremos que explorar y tratar estos conjuntos de datos. Es en esta parte que Pandas entra en acción.

Pandas es un paquete Python que tiene estructuras de manipulación de datos de alto nivel, teniendo como base la biblioteca de NumPy ya estudiada en el capitulo anterior. El paquete puede ser visto como una combinación de los recursos de manipulación de datos encontrados en planillas y bancos de datos relacionales con el alto desempeño de *arrays* del paquete NumPy.

Otro punto fuerte de Pandas es que sus estructuras de datos son fácilmente utilizadas por las principales bibliotecas de análisis del lenguaje Python.

Con estos puntos y teniendo en mente que preparación y tratamiento de datos son parte fundamental en un proceso de análisis de datos, la biblioteca Pandas se convierte en herramienta indispensable para un profesional de *Data Science*.

### Importando la biblioteca

Del mismo modo que vimos en el capítulo anterior (NumPy) es costumbre a la hora de realizar el `import` de la biblioteca Pandas asignarle un *alias*. Por convención la comunidad acostumbra utilizar el seudónimo `pd` para el *namespace* `pandas`, de la siguiente forma:

```
import pandas as pd
```

Para acceder a un recurso del paquete basta escribir el *alias* ( `pd` ) seguido del carácter "." y después el nombre del recurso. El código de abajo imprime la versión del paquete pandas que estamos utilizando en este curso.

```
pd.__version__
```

**Salida:**

```
'1.0.5'
```

## 7.2 CONOCIENDO LAS ESTRUCTURAS DE DATOS DE PANDAS

### *Series*

**Documentación**: https://pandas.pydata.org/docs/reference/api/pandas.Series.html

*Series* son *arrays* unidimensionales etiquetados capaces de almacenar cualquier tipo de dato. Las etiquetas de las líneas son llamadas de `index` . La forma básica de creación de una *Series* es la siguiente:

```
s = pd.Series(datos, index=index)
```

El argumento `datos` puede ser un diccionario, una lista, un *array* NumPy o una constante.

```
pd.Series(['Jetta Variant', 'Passat', 'Crossfox'])
```

**Salida:**

```
0    Jetta Variant
1           Passat
2         Crossfox
dtype: object
```

Cuando omitimos el parámetro `index` la *Series* queda apenas con el índice patrón que está compuesto por números enteros iniciados en cero. Adicionando el argumento `index` quedaría de la siguiente forma:

```
pd.Series(['Jetta Variant', 'Passat', 'Crossfox'], index=['Carro A', 'Carro B', 'Carro C'])
```

**Salida:**

```
Carro A    Jetta Variant
Carro B           Passat
Carro C         Crossfox
dtype: object
```

Hablaremos más sobre indexación en las próximas secciones del curso.

### *DataFrame*

**Documentación**: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

*DataFrame* es una estructura de datos tabular bidimensional con etiquetas en las línea y columnas. Como las *Series*, los *DataFrames* son capaces de almacenar cualquier tipo de datos. La forma básica de creación de un *DataFrame* es la siguiente:

```
df = pd.DataFrame(datos, index=index, columns=columns)
```

El argumento `datos` puede ser un diccionario, una lista, un *array* NumPy, una *Series* u otro

*DataFrame*.

**Creando un *DataFrame* a partir de una lista de diccionarios**

Considere la lista de diccionarios a seguir:

```
datos = [
    {'Nombre': 'Jetta Variant', 'Motor': 'Motor 4.0 Turbo', 'Año': 2003, 'Kilometraje': 44410.0, 'Cero_km': False, 'Valor': 17615.73},
    {'Nombre': 'Passat', 'Motor': 'Motor Diesel', 'Año': 1991, 'Kilometraje': 5712.0, 'Cero_km': False, 'Valor': 21232.39},
    {'Nombre': 'Crossfox', 'Motor': 'Motor Diesel V8', 'Año': 1990, 'Kilometraje': 37123.0, 'Cero_km': False, 'Valor': 14566.43}
]
```

Con esta estructura de datos podemos crear una *DataFrame* de forma bastante simple, donde los valores de cada diccionario serán los registros del *DataFrame* (líneas) y las claves serán los *labels* de las columnas.

```
dataset = pd.DataFrame(datos)
dataset
```

**Salida:**

| | Nombre | Motor | Año | Kilometraje | Cero_km | Valor |
|---|---|---|---|---|---|---|
| **0** | Jetta Variant | Motor 4.0 Turbo | 2003 | 44410.0 | False | 17615.73 |
| **1** | Passat | Motor Diesel | 1991 | 5712.0 | False | 21232.39 |
| **2** | Crossfox | Motor Diesel V8 | 1990 | 37123.0 | False | 14566.43 |

**Creando un *DataFrame* a partir de un diccionario**

Otra forma bastante utilizada puede ser vista en el código de abajo. Pasando un diccionario, donde sus claves serán los *labels* de las columnas y los valores, en formato de listas, serán los registros de cada columna. El resultado en este caso es idéntico al del ejemplo anterior.

```
datos = {
    'Nombre': ['Jetta Variant', 'Passat', 'Crossfox'],
    'Motor': ['Motor 4.0 Turbo', 'Motor Diesel', 'Motor Diesel V8'],
    'Año': [2003, 1991, 1990],
    'Kilometraje': [44410.0, 5712.0, 37123.0],
    'Cero_km': [False, False, False],
    'Valor': [17615.73, 21232.39, 14566.43]
}
```

Formatos como este y el del ejemplo anterior son respuestas bastante comunes cuando consultamos `APIs Rest` . En este tipo de llamada las respuestas generalmente son retornadas en el formato `JSON` (*Javascript Object Notation*) que guarda algunas semejanzas con diccionarios Python y pueden ser convertidos en un diccionario de forma bastante simple.

```
dataset = pd.DataFrame(datos)
```

```
dataset
```

**Salida:**

| | Nombre | Motor | Año | Kilometraje | Cero_km | Valor |
|---|---|---|---|---|---|---|
| 0 | Jetta Variant | Motor 4.0 Turbo | 2003 | 44410.0 | False | 17615.73 |
| 1 | Passat | Motor Diesel | 1991 | 5712.0 | False | 21232.39 |
| 2 | Crossfox | Motor Diesel V8 | 1990 | 37123.0 | False | 14566.43 |

### Creando un *DataFrame* a partir de un archivo externo

**Documentación**: https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html

Una de las formas más utilizadas para creación de *DataFrames* es a partir de la importación de datos de archivos externos. En nuestro curso vamos a enfocar en *datasets* almacenados en el formato CSV, pero Pandas ofrece soporte para importación de datos en diversos formatos (TXT, XLSX, HTML, etc.).

A lo largo de este curso vamos a realizar el estudio de la etapa inicial de todo proyecto en *Data Science*. Esta etapa está formada por algunos pasos que pueden ser divididos en: colecta e importación de los datos; identificación y entendimiento; transformación; tratamiento; y resumen.

Para eso vamos a utilizar dos conjuntos de datos, uno para nuestros ejemplos y otro como un proyecto individual donde serán aplicados los conocimientos obtenidos en el curso.

El *dataset* de la clase se encuentra en el archivo **"db.csv"** y contiene un conjunto de informaciones sobre vehículos anunciados para la venta. El *dataset* del proyecto individual está en el archivo **"db_proyecto.csv"** y contiene informaciones sobre un conjunto de inmuebles disponibles para locación en la Ciudad de México. Puedes descargar los archivos a través de lo siguientes vínculos:

- Archivo de la clase: **"db.csv"**
- Archivo del proyecto individual: **"db_proyecto.csv"**

Para cargar el contenido de un archivo CSV en un *DataFrame* Pandas pone a disposición el método `read_csv()`. La forma más simple de utilización de este método puede ser vista en el código de más abajo.

Pasamos como primer parámetro el nombre y localización del archivo CSV. Como nuestro archivo CSV está localizado en la misma carpeta del *notebook* que estamos utilizando, basta informar apenas el nombre del archivo, pero suponiendo de que este archivo esté dentro de una carpeta llamada `data` y esta carpeta esté localizada en la misma carpeta de nuestro *notebook*, la dirección deberá ser informada de la siguiente forma: `'./data/db.csv'` o `'data/db.csv'`.

El parámetro `sep` informa que tipo de separador de columnas el archivo CSV está utilizando. El

*default* de este parámetro es el carácter ",". Caso el archivo CSV utilice otro tipo de carácter como separador de columnas, el parámetro `sep` debe ser configurado. Nuestro archivo de la clase utiliza el carácter ";" como separador.

```
dataset = pd.read_csv('db.csv', sep=';')
dataset.head()
```

**Salida:**

| | Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|---|
| **0** | Jetta Variant | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 |
| **1** | Passat | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 |
| **2** | Crossfox | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 |
| **3** | DS5 | Motor 2.4 Turbo | 2020 | NaN | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 |
| **4** | Aston Martin DB4 | Motor 2.4 Turbo | 2006 | 25757.0 | False | ['Llantas de aleación', '4 X 4', 'Pantalla mul... | 18522.42 |

El código de arriba lee el archivo CSV y carga su contenido en un *DataFrame* de Pandas que recibe el nombre de **dataset**. En la última línea del código es utilizada la función `head()` que retorna las primeras *n* líneas de un *DataFrame* o *Series*. Esta funcionalidad es útil para verificar de forma rápida si tu *dataset* posee el tipo cierto de datos o si fue importado de forma correcta. Cuando la función `head()` es utilizada sin parámetro genera una visualización de las cinco primeras líneas de un *DataFrame*. Caso desee visualizar un número diferente de líneas de la parte inicial del *DataFrame*, basta informar el número total de líneas que deseas visualizar.

```
dataset = pd.read_csv('db.csv', sep = ';', index_col = 0)
dataset.head(10)
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 |
| **DS5** | Motor 2.4 Turbo | 2020 | NaN | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 |
| **Aston Martin DB4** | Motor 2.4 Turbo | 2006 | 25757.0 | False | ['Llantas de aleación', '4 X 4', 'Pantalla mul... | 18522.42 |
| **Palio Weekend** | Motor 1.8 16v | 2012 | 10728.0 | False | ['Sensor de estacionamiento', 'Techo panorámic... | 19499.55 |
| **A5** | Motor 4.0 Turbo | 2020 | NaN | True | ['Cambio automático', 'Cámara de estacionamien... | 11289.04 |
| **Série 3 Cabrio** | Motor 1.0 8v | 2009 | 77599.0 | False | ['Control de estabilidad', 'Sensor crepuscular... | 22462.09 |
| **Dodge Jorney** | Motor 3.0 32v | 2010 | 99197.0 | False | ['Vidrios eléctricos', 'Piloto automático', 'T... | 24143.25 |
| **Carens** | Motor 5.0 V8 Bi-Turbo | 2011 | 37978.0 | False | ['Aire acondicionado', 'Tablero digital', 'Pan... | 15313.30 |

Comparando los resultados retornados por los dos códigos anteriores es posible notar una diferencia. En el resultado del primer código para cada línea del *DataFrame* tenemos un índice numérico que comienza en cero. En el segundo resultado no tenemos este índice numérico visible, en lugar de los índices tenemos los nombres de los carros como etiquetas de las líneas del *DataFrame*. Ese cambio ocurre por la utilización del parámetro `index_col` de la función `read_csv()`. Este parámetro indica cual o cuales columnas serán usadas como etiquetas de línea del *DataFrame*. El valor cero informa para la función que la primera columna del archivo CSV debe ser utilizada como etiqueta de las líneas del *DataFrame*. Hablaremos un poco más sobre índices en la próxima sección.

## 7.3 EJERCICIOS:

1. Para iniciar el proyecto haga el *upload* del archivo **"db_proyecto.csv"** para el Colab.
2. Crea un *DataFrame* con el contenido del archivo **"db_proyecto.csv"** y llámalo de **`proyecto`**. El archivo utiliza como separador de columnas el carácter ";".

   **Respuestas:**

   ```
   proyecto = pd.read_csv('db_proyecto.csv', sep=';')
   ```

3. Visualiza apenas los cinco primeros registros del *DataFrame* **proyecto**.

   **Respuestas:**

   ```
   proyecto.head()
   ```

   **Salida:**

| | Tipo | Valor | Barrio | Alcaldia | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | Caracteristicas |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Departamento | 681.50 | Barrio del Niño Jesús | Tlalpan | 85 | 3 | 2 | 1.0 | 40.7 | [Planta baja, Buena iluminación, Cocina integr... |
| 1 | Departamento | 681.55 | Carola | Alvaro Obregón | 60 | 2 | 2 | 2.0 | 0.0 | [Buena iluminación,Sala Comedor,Cocina Equipad... |
| 2 | Departamento | 864.80 | Condesa | Cuauhtémoc | 70 | 2 | 2 | 1.0 | 84.6 | [Vigilancia 24 horas, Gimnasio, Roof Garden co... |
| 3 | Casa | 1363.00 | Contadero | Cuajimalpa de Morelos | 131 | 3 | 2 | 2.0 | 0.0 | [Sala, Comedor, Cocina integral,Área de lavado... |
| 4 | Departamento | 705.00 | Del Valle | Benito Juárez | 79 | 2 | 2 | 1.0 | 61.1 | [Recámara con vestidor y baño, Closet,Comedor ... |

# 7.4 OPERACIONES BÁSICAS CON UN DATAFRAME

## Objetos *Index*

Pandas posee un conjunto de objetos *Index* que son responsables por el almacenamiento de las etiquetas de los ejes de *Series* y *DataFrames*. Estos objetos también pueden guardar otros metadatos como, por ejemplo, los nombres de los ejes.

### *RangeIndex*

Este es el tipo de índice patrón utilizado por el *DataFrame* y *Series* cuando ningún índice explícito es dado por el usuario.

```
datos = {
    'Nombre': ['Jetta Variant', 'Passat', 'Crossfox'],
    'Motor': ['Motor 4.0 Turbo', 'Motor Diesel', 'Motor Diesel V8'],
    'Año': [2003, 1991, 1990],
    'Kilometraje': [44410.0, 5712.0, 37123.0],
    'Cero_km': [False, False, False],
    'Valor': [17615.73, 21232.39, 14566.43]
}
df = pd.DataFrame(datos)
df
```

**Salida:**

| | Nombre | Motor | Año | Kilometraje | Cero_km | Valor |
|---|---|---|---|---|---|---|
| 0 | Jetta Variant | Motor 4.0 Turbo | 2003 | 44410.0 | False | 17615.73 |
| 1 | Passat | Motor Diesel | 1991 | 5712.0 | False | 21232.39 |
| 2 | Crossfox | Motor Diesel V8 | 1990 | 37123.0 | False | 14566.43 |

```
df.index
```

**Salida:**

```
RangeIndex(start=0, stop=3, step=1)
```

*RangeIndex* posee los atributos `start`, `stop` y `step` así como el tipo de secuencia `range()` que

conocimos en el capítulo sobre secuencias de Python.

***Index***

Al crear una *Series* o un *DataFrame* la secuencia de etiquetas que fue utilizada en la construcción de estos objetos será internamente convertida en un *Index*.

Pandas soporta valores de índice no exclusivos, pero caso una operación que no soporta valores duplicados de índice sea ejecutada, una excepción será generada.

```
df = pd.DataFrame(datos, index = ['A', 'B', 'C'])
df
```

**Salida:**

| | Nombre | Motor | Año | Kilometraje | Cero_km | Valor |
|---|---|---|---|---|---|---|
| **A** | Jetta Variant | Motor 4.0 Turbo | 2003 | 44410.0 | False | 17615.73 |
| **B** | Passat | Motor Diesel | 1991 | 5712.0 | False | 21232.39 |
| **C** | Crossfox | Motor Diesel V8 | 1990 | 37123.0 | False | 14566.43 |

```
df.index
```

**Salida:**

```
Index(['A', 'B', 'C'], dtype='object')
```

Objetos *Index* no pueden ser modificados por el usuario, o sea, son inmutables.

```
df.index[1] = 'H'
```

**Salida:**

```
---------------------------------------------------------------------------
TypeError: Index does not support mutable operations
```

**Definiendo como índice una de las columnas existentes**

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.set_index.html

Utilizando el método `set_index()` es posible definir un índice para un *DataFrame* utilizando como etiquetas los valores de una o más columnas del propio *DataFrame*. Considere el mismo *DataFrame* de la sección anterior para el ejemplo de abajo.

```
df.set_index('Nombre')
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Valor |
|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | 17615.73 |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | 21232.39 |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | 14566.43 |

En este ejemplo fueron utilizados los valores de la columna **"Nombre"** como etiquetas para las líneas de nuestro *DataFrame*. El método `set_index()` posee un parámetro llamado `drop` que viene configurando como *True* por *default*. Este parámetro indica si la columna utilizada como nuevo índice debe permanecer ( `drop=False` ) o ser retirada del *DataFrame* (*default*).

Otra observación importante cuando utilizamos métodos de *Series* y *DataFrames* es sobre el parámetro `inplace` . Este parámetro aparece en buena parte de los métodos que vamos a estudiar, y viene configurado como *False* por *default*. El mismo informa si el objeto debe ser modificado *in-place* ( `inplace=True` ) o no (*default*). Cuando `inplace` es *False* apenas una visualización del objeto es creada, permaneciendo inalterado. Observe que cuando accedemos el índice del *DataFrame* **df** (código de abajo) no encontramos las modificaciones que fueron hechas, eso porque el parámetro *inplace* fue mantenido con su configuración *default* (*False*).

```
df.index
```

**Salida:**

```
Index(['A', 'B', 'C'], dtype='object')
```

```
df
```

**Salida:**

| | Nombre | Motor | Año | Kilometraje | Cero_km | Valor |
|---|---|---|---|---|---|---|
| **A** | Jetta Variant | Motor 4.0 Turbo | 2003 | 44410.0 | False | 17615.73 |
| **B** | Passat | Motor Diesel | 1991 | 5712.0 | False | 21232.39 |
| **C** | Crossfox | Motor Diesel V8 | 1990 | 37123.0 | False | 14566.43 |

Para modificar el *DataFrame* basta configurar `inplace` como *True* en el método `set_index()` .

```
df.set_index('Nombre', inplace=True)
```

```
df.index
```

**Salida:**

```
Index(['Jetta Variant', 'Passat', 'Crossfox'], dtype='object', name='Nombre')
```

## Indexación y Selecciones con *DataFrames*

El operador de indexación de Python y NumPy [] ofrecen acceso rápido y fácil a las estructuras de datos de Pandas. Eso facilita el aprendizaje visto que ya sabemos lidiar con secuencias y diccionarios de Python y *arrays* NumPy. Sin embargo, la biblioteca Pandas ofrece también algunos métodos optimizados de acceso a datos que aprenderemos en esta sección.

Considera en nuestros ejemplos de esta sección el *DataFrame* **`dataset`** que cargamos con el contenido del archivo **"db.csv"**.

```
dataset.head()
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 |
| **DS5** | Motor 2.4 Turbo | 2020 | NaN | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 |
| **Aston Martin DB4** | Motor 2.4 Turbo | 2006 | 25757.0 | False | ['Llantas de aleación', '4 X 4', 'Pantalla mul... | 18522.42 |

### Seleccionando columnas

Como citado anteriormente el operador de indexación [] de las estructuras de datos de Pandas funciona de forma semejante a la indexación de *arrays* NumPy, excepto por la posibilidad de utilización de las etiquetas de líneas y columnas en el lugar de apenas números enteros.

```
dataset['Valor']
```

**Salida:**

```
Nombre
Jetta Variant            17615.73
Passat                   21232.39
Crossfox                 14566.43
DS5                      24909.81
Aston Martin DB4         18522.42
                           ...
Phantom 2013             10351.92
Cadillac Ciel concept    10333.41
Clase GLK                13786.81
Aston Martin DB5         24422.18
Macan                    18076.29
Name: Valor, Length: 258, dtype: float64
```

En el trecho de código de encima fue realizada la selección de la columna con etiqueta "Valor". Pero también puede ser accedido a un índice en una *Series* o una columna en un *DataFrame* utilizando las

etiquetas directamente como atributos del objeto. El código de abajo ejemplifica eso:

```
dataset.Valor
```

**Salida:**

```
Nombre
Jetta Variant            17615.73
Passat                   21232.39
Crossfox                 14566.43
DS5                      24909.81
Aston Martin DB4         18522.42
                           ...
Phantom 2013             10351.92
Cadillac Ciel concept    10333.41
Clase GLK               13786.81
Aston Martin DB5         24422.18
Macan                    18076.29
Name: Valor, Length: 258, dtype: float64
```

Algunos cuidados necesitan ser tomados al realizar selecciones considerando las etiquetas como atributos del objeto:

- Sólo es posible usar este método si las etiquetas de índice y de columna son identificadores Python válidos. El siguiente *link* trae una explicación sobre identificadores válidos: https://docs.python.org/3.6/reference/lexical_analysis.html#identifiers
- El atributo no estará disponible si entra en conflicto con un nombre de método existente, por ejemplo **min** no es permitido, pero ['min'] es posible.

Observe que el *output* de los códigos de arriba están en el formato de *Series*.

```
type(dataset['Valor'])
```

**Salida:**

```
pandas.core.series.Series
```

Caso sea necesario obtener como retorno un *DataFrame* con apenas una columna, basta pasar para el operador de indexación [] una lista con la etiqueta de la columna.

```
dataset[['Valor']]
```

**Salida:**

| Nombre | Valor |
|---|---|
| Jetta Variant | 17615.73 |
| Passat | 21232.39 |
| Crossfox | 14566.43 |
| DS5 | 24909.81 |
| Aston Martin DB4 | 18522.42 |
| ... | ... |
| Phantom 2013 | 10351.92 |
| Cadillac Ciel concept | 10333.41 |
| Classe GLK | 13786.81 |
| Aston Martin DB5 | 24422.18 |
| Macan | 18076.29 |

258 rows × 1 columns

en esta operación tenemos como retorno un *DataFrame*.

```
type(dataset[['Valor']])
```

**Salida:**

```
pandas.core.frame.DataFrame
```

Es posible también pasar una lista de columnas para el operador de indexación [] para seleccionar las columnas en el orden especificada. Caso una de las columnas no sea encontrada en el *DataFrame*, una excepción será generada.

```
dataset[['Año', 'Kilometraje', 'Valor']]
```

**Salida:**

| Nombre | Año | Kilometraje | Valor |
|---|---|---|---|
| **Jetta Variant** | 2003 | 44410.0 | 17615.73 |
| **Passat** | 1991 | 5712.0 | 21232.39 |
| **Crossfox** | 1990 | 37123.0 | 14566.43 |
| **DS5** | 2020 | NaN | 24909.81 |
| **Aston Martin DB4** | 2006 | 25757.0 | 18522.42 |
| **...** | ... | ... | ... |
| **Phantom 2013** | 2014 | 27505.0 | 10351.92 |
| **Cadillac Ciel concept** | 1991 | 29981.0 | 10333.41 |
| **Classe GLK** | 2002 | 52637.0 | 13786.81 |
| **Aston Martin DB5** | 1996 | 7685.0 | 24422.18 |
| **Macan** | 1992 | 50188.0 | 18076.29 |

258 rows × 3 columns

### *Slices* - `[i:j:k]`

Usando el operador de indexación también es posible seleccionar líneas por sus índices numéricos, esto es, por la posición de las líneas. Los ingresos en *DataFrames* y *Series* funciona de la misma forma que con las listas de Python. El operador [i:j:k] permite seleccionar en el *DataFrame* o *Series* del índice i hasta el ínidce j con el paso k. Recordando que la indexación tiene origen en el cero y la línea con índice i es **incluida** y la línea con índice j **no es incluida** en el resultado.

```
dataset[0:3]
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 |

En las próximas secciones aprenderemos una forma más robusta y consistente de realizar *slices* con Pandas utilizando los métodos `loc` y `iloc`.

### Utilizando `.loc` para selecciones

Pandas ofrece un método para indexación puramente basado en etiquetas. Con el método `loc` podemos seleccionar un grupo de líneas y columnas según las etiquetas de estos ejes o utilizando una matriz booleana.

El código a seguir selecciona el registro o registros del *DataFrame* **`dataset`** que presenten la etiqueta de línea "Passat". Note que como fue encontrado apenas un registro, el retorno del código es una *Series* con las informaciones de la línea encontrada.

```
dataset.loc['Passat']
```

**Salida:**

```
Motor                                                Motor Diesel
Año                                                          1991
Kilometraje                                                  5712
Cero_km                                                     False
Accesorios     ['Pantalla multimedia', 'Techo panorámico', 'F...
Valor                                                     21232.4
Name: Passat, dtype: object
```

Para múltiples selecciones basta pasar una lista con las etiquetas de la búsqueda. Observe que con el uso del método `loc` todos las etiquetas solicitadas deben estar en el índice o caso contrario un error (KeyError) será generado.

```
dataset.loc[['Passat', 'DS5']]
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 |
| **DS5** | Motor 2.4 Turbo | 2020 | NaN | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 |

Los *slices* también son posibles con el método `loc` . En este tipo de selección la línea con la etiqueta inicial es **incluida** y la línea con la etiqueta final también es **incluida** en el resultado.

```
dataset.loc['Jetta Variant':'DS5']
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 |
| **DS5** | Motor 2.4 Turbo | 2020 | NaN | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 |

Figura 7.15: 0.5

Como informado al inicio de esta sección, es posible seleccionar un grupo de líneas y columnas de un *DataFrame* a través de las etiquetas de estos ejes. El siguiente código selecciona, del *DataFrame*

`dataset`, las líneas con etiquetas "Passat" y "DS5" y las columnas con etiquetas "Motor" y "Valor".

```
dataset.loc[['Passat', 'DS5'], ['Motor', 'Valor']]
```

**Salida:**

| Nombre | Motor | Valor |
|---|---|---|
| **Passat** | Motor Diesel | 21232.39 |
| **DS5** | Motor 2.4 Turbo | 24909.81 |

Vea que en el método `loc` cuando seleccionamos líneas y columnas de un *DataFrame*, el primer argumento son las etiquetas de las líneas y el segundo las etiquetas de las columnas, o sea, `loc['etiquetas de las líneas', 'etiquetas de las columnas']`. Pudiendo estos argumentos ser apenas una etiqueta o una lista o un *slice*. El código de abajo muestra la selección de todos los registros ( : ) del `dataset` y apenas las columnas "Motor" y "Valor".

```
dataset.loc[:, ['Motor', 'Valor']]
```

**Salida:**

| Nombre | Motor | Valor |
|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 17615.73 |
| **Passat** | Motor Diesel | 21232.39 |
| **Crossfox** | Motor Diesel V8 | 14566.43 |
| **DS5** | Motor 2.4 Turbo | 24909.81 |
| **Aston Martin DB4** | Motor 2.4 Turbo | 18522.42 |
| **...** | ... | ... |
| **Phantom 2013** | Motor V8 | 10351.92 |
| **Cadillac Ciel concept** | Motor V8 | 10333.41 |
| **Classe GLK** | Motor 5.0 V8 Bi-Turbo | 13786.81 |
| **Aston Martin DB5** | Motor Diesel | 24422.18 |
| **Macan** | Motor Diesel V6 | 18076.29 |

258 rows × 2 columns

### Utilizando *.iloc* para selecciones

La diferencia entre los métodos `loc` y `iloc` está apenas en la base para indexación. En el caso del `loc`, como aprendimos en la sección anterior, la indexación está basada en las etiquetas, ya para el método `iloc` la indexación es basada en números enteros, o mejor, en la posición de las informaciones.

```
dataset.iloc[[1]]
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 |

El funcionamiento es básicamente el mismo de listas del Python y *arrays* NumPy. Recordando nuevamente que la indexación tiene origen en el cero y que el índice inicial es incluido en el resultado y el índice final no es incluido en el resultado.

```
dataset.iloc[1:4]
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 |
| **DS5** | Motor 2.4 Turbo | 2020 | NaN | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 |

Abajo tenemos un ejemplo donde la selección de las líneas es hecha con un *slice* y la selección de las columnas a partir de una lista de índices.

```
dataset.iloc[1:4, [0, 5, 2]]
```

**Salida:**

| Nombre | Motor | Valor | Kilometraje |
|---|---|---|---|
| **Passat** | Motor Diesel | 21232.39 | 5712.0 |
| **Crossfox** | Motor Diesel V8 | 14566.43 | 37123.0 |
| **DS5** | Motor 2.4 Turbo | 24909.81 | NaN |

## Queries con *DataFrames*

### Utilizando una *Series* booleana

Otra forma bastante común de filtrar los datos es con el uso de vectores booleanos. Observe la secuencia de códigos de ejemplo abajo. Vamos a utilizar la columna "Motor" de nuestro *DataFrame* de ejemplo.

```
dataset.Motor
```

**Salida:**

```
Nombre
Jetta Variant                  Motor 4.0 Turbo
Passat                            Motor Diesel
Crossfox                       Motor Diesel V8
DS5                            Motor 2.4 Turbo
Aston Martin DB4               Motor 2.4 Turbo
                                  ...
Phantom 2013                        Motor V8
Cadillac Ciel concept               Motor V8
Clase GLK               Motor 5.0 V8 Bi-Turbo
Aston Martin DB5                  Motor Diesel
Macan                          Motor Diesel V6
Name: Motor, Length: 258, dtype: object
```

Una forma simple de crear una *Series* booleana es con el uso de operadores de comparación. El próximo trecho de código crea una variable llamada **`select`** y almacena en ella el resultado de la siguiente prueba: `dataset.Motor == 'Motor Diesel'`.

Este prueba tiene como resultado una *Series* booleana con el mismo número de líneas del *DataFrame* **`dataset`**. Esta *Series* tendrá valores *False* cuando la condición no es satisfecha y *True* cuando es satisfecha.

```
select = (dataset.Motor == 'Motor Diesel')
select
```

**Salida:**

```
Nombre
Jetta Variant            False
Passat                    True
Crossfox                 False
DS5                      False
Aston Martin DB4         False
                         ...
Phantom 2013             False
Cadillac Ciel concept    False
Clase GLK               False
Aston Martin DB5          True
Macan                    False
Name: Motor, Length: 258, dtype: bool
```

```
type(select)
```

**Salida:**

```
pandas.core.series.Series
```

Pasando la *Series* booleana para el operador de indexación tenemos como resultado un *DataFrame* apenas con las líneas que tienen el valor *True*.

```
dataset[select]
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 |
| **Effa Hafei Picape Baú** | Motor Diesel | 1991 | 102959.0 | False | ['Control de estabilidad', 'Tablero digital', ... | 25136.93 |
| **Sorento** | Motor Diesel | 2020 | NaN | True | ['Sensor de lluvia', 'Cámara de estacionamient... | 16279.87 |
| **New Fiesta Hatch** | Motor Diesel | 2017 | 118895.0 | False | ['Sensor de estacionamiento', 'Cerraduras eléc... | 13201.43 |
| **Kangoo Express** | Motor Diesel | 2007 | 29132.0 | False | ['Bancos de cuero', 'Cambio automático', 'Pilo... | -999999.00 |
| **Fit** | Motor Diesel | 2013 | 44329.0 | False | ['Frenos ABS', 'Cámara de estacionamiento', 'C... | 15567.25 |
| **Cielo Hatch** | Motor Diesel | 2019 | 18036.0 | False | ['Tablero digital', 'Pantalla multimedia', 'Cá... | 29039.54 |
| **Symbol** | Motor Diesel | 2016 | 117714.0 | False | ['4 X 4', 'Piloto automático', 'Sensor crepusc... | 26606.12 |
| **A4 Sedan** | Motor Diesel | 2002 | 30511.0 | False | ['Cámara de estacionamiento', '4 X 4', 'Cerrad... | 19273.81 |
| **A4 Avant** | Motor Diesel | 2014 | 17357.0 | False | ['Techo panorámico', '4 X 4', 'Bancos de cuero... | 27789.38 |
| **Silver Shadow** | Motor Diesel | 2015 | 99052.0 | False | ['4 X 4', 'Pantalla multimedia', 'Aire acondic... | 28713.64 |
| **Camry** | Motor Diesel | 2020 | NaN | True | ['Cerraduras eléctricas', 'Llantas de aleación... | 27719.45 |
| **Clio** | Motor Diesel | 1990 | 66437.0 | False | ['4 X 4', 'Cambio automático', 'Frenos ABS', '... | 22226.07 |
| **Serie 3 M3 Coupé** | Motor Diesel | 2013 | 19896.0 | False | ['Bancos de cuero', 'Sensor de lluvia', 'Aire ... | 13843.98 |
| **Gallardo LP 560 – 4** | Motor Diesel | 2000 | 113045.0 | False | ['Frenos ABS', 'Techo panorámico', 'Pantalla m... | 28823.78 |
| **Chana Utility** | Motor Diesel | 2009 | 52034.0 | False | ['Aire acondicionado', 'Sensor de estacionamie... | 26532.46 |
| **TT Roadster** | Motor Diesel | 1998 | 107392.0 | False | ['Aire acondicionado', 'Sensor de estacionamie... | 17062.28 |
| **Aston Martin Virage** | Motor Diesel | 2020 | NaN | True | ['Cerraduras eléctricas', 'Control de tracción... | 19458.04 |
| **Sandero** | Motor Diesel | 2015 | 34783.0 | False | ['Piloto automático', 'Techo panorámico', 'Cam... | 22470.12 |
| **XKR** | Motor Diesel | 1996 | 85127.0 | False | ['Control de tracción', 'Aire acondicionado', ... | 10852.63 |
| **X5** | Motor Diesel | 2002 | 13606.0 | False | ['Control de tracción', 'Piloto automático', '... | 12408.26 |
| **Dodge Dakota** | Motor Diesel | 1993 | 71544.0 | False | ['Control de tracción', 'Sensor de estacionami... | 28216.67 |
| **i30 CW** | Motor Diesel | 2015 | 8497.0 | False | ['Sensor de lluvia', 'Vidrios eléctricos', 'Co... | 14662.35 |
| **Serie 7 Sedan** | Motor Diesel | 2020 | NaN | True | ['Vidrios eléctricos', 'Cerraduras eléctricas'... | 13507.96 |
| **V60** | Motor Diesel | 2004 | 91840.0 | False | ['Llantas de aleación', 'Sensor crepuscular', ... | 22945.75 |
| **Aston Martin DB5** | Motor Diesel | 1996 | 7685.0 | False | ['Aire acondicionado', '4 X 4', 'Cambio automá... | 24422.18 |

Podemos crear condiciones más complejas con el uso de operadores condicionales. Los operadores válidos para este tipo de operación son: **|** para o, **&** para y, y **~** para negación. Las expresiones deben ser agrupadas con el uso de paréntesis, pues, por patrón, Python evaluará expresiones del tipo `df['X'] > 0 & df['Y'] < 5` como `df['X'] > (0 & df['Y']) < 5`, mientras el orden de evaluación deseado es `(df['X'] > 0) & (df['Y'] < 5)`.

Usar una *Series* booleana para indexar una *Series* o un *DataFrame* funciona exactamente como en un *array* NumPy.

```
dataset[(dataset.Motor == 'Motor Diesel') & (dataset.Cero_km == True)]
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|
| **Sorento** | Motor Diesel | 2020 | NaN | True | ['Sensor de lluvia', 'Cámara de estacionamient... | 16279.87 |
| **Camry** | Motor Diesel | 2020 | NaN | True | ['Cerraduras eléctricas', 'Llantas de aleación... | 27719.45 |
| **Aston Martin Virage** | Motor Diesel | 2020 | NaN | True | ['Cerraduras eléctricas', 'Control de tracción... | 19458.04 |
| **Serie 7 Sedan** | Motor Diesel | 2020 | NaN | True | ['Vidrios eléctricos', 'Cerraduras eléctricas'... | 13507.96 |

Con los métodos `loc` y `iloc` también podemos seleccionar a lo largo de más de un eje usando vectores booleanos combinados con otras expresiones de indexación.

```
dataset.loc[(dataset.Motor == 'Motor Diesel') & (dataset.Cero_km == True), 'Valor']
```

**Salida:**

```
Nombre
Sorento                16279.87
Camry                  27719.45
Aston Martin Virage    19458.04
Série 7 Sedan           13507.96
Name: Valor, dtype: float64
```

### Utilizando el método `query()`

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.query.html

El método `query()` permite la realización de consultas por columnas de un *DataFrame* con el uso de expresiones booleanas.

```
dataset.query('Kilometraje < 5000')
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|
| **Série 5 Gran Turismo** | Motor 4.0 Turbo | 2001 | 2395.0 | False | ['Control de tracción', 'Piloto automático', '... | 18156.33 |
| **HB20S** | Motor V8 | 2008 | 610.0 | False | ['Techo panorámico', 'Llantas de aleación', 'C... | 23256.24 |
| **Cayenne** | Motor 1.0 8v | 2003 | 4539.0 | False | ['Control de tracción', 'Piloto automático', '... | 24782.36 |
| **Aston Martin DB9** | Motor 1.8 16v | 1996 | 4821.0 | False | ['Aire acondicionado', 'Vidrios eléctricos', '... | 20985.98 |
| **Grand Cherokee** | Motor 5.0 V8 Bi-Turbo | 1996 | 3173.0 | False | ['Cambio automático', 'Cerraduras eléctricas',... | 13794.85 |
| **RX 350** | Motor V6 | 2013 | 2314.0 | False | ['Piloto automático', 'Sensor crepuscular', 'A... | 27764.76 |
| **Cayman** | Motor 3.0 32v | 2013 | 2627.0 | False | ['Pantalla multimedia', 'Tablero digital', 'Se... | 23592.39 |
| **X1** | Motor Diesel V8 | 1991 | 107.0 | False | ['4 X 4', 'Techo panorámico', 'Aire acondicion... | 20593.75 |
| **Phantom III** | Motor Diesel V6 | 2008 | 4420.0 | False | ['Vidrios eléctricos', 'Bancos de cuero', 'Pil... | 13027.61 |

Con el uso de la notación patrón de Python, el código anterior es equivalente a `dataset[dataset['Kilometraje'] < 5000]`.

```
dataset.query('Kilometraje < 5000 or Año == 2018')
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|
| **Linea** | Motor 1.0 8v | 2018 | 17720.0 | False | ['Piloto automático', 'Tablero digital', 'Vidr... | 11871.74 |
| **Série 5 Gran Turismo** | Motor 4.0 Turbo | 2001 | 2395.0 | False | ['Control de tracción', 'Piloto automático', '... | 18156.33 |
| **HB20S** | Motor V8 | 2008 | 610.0 | False | ['Techo panorámico', 'Llantas de aleación', 'C... | 23256.24 |
| **Cadillac CTS** | Motor 1.8 16v | 2018 | 88661.0 | False | ['Cerraduras eléctricas', 'Sensor de lluvia', ... | 10778.92 |
| **Cayenne** | Motor 1.0 8v | 2003 | 4539.0 | False | ['Control de tracción', 'Piloto automático', '... | 24782.36 |
| **Edge** | Motor V6 | 2018 | 26544.0 | False | ['Frenos ABS', 'Control de tracción', 'Sensor ... | 17683.31 |
| **Aston Martin DB9** | Motor 1.8 16v | 1996 | 4821.0 | False | ['Aire acondicionado', 'Vidrios eléctricos', '... | 20985.98 |
| **Grand Cherokee** | Motor 5.0 V8 Bi-Turbo | 1996 | 3173.0 | False | ['Cambio automático', 'Cerraduras eléctricas',... | 13794.85 |
| **RX 350** | Motor V6 | 2013 | 2314.0 | False | ['Piloto automático', 'Sensor crepuscular', 'A... | 27764.76 |
| **Cayman** | Motor 3.0 32v | 2013 | 2627.0 | False | ['Pantalla multimedia', 'Tablero digital', 'Se... | 23592.39 |
| **Lamborghini Jalpa** | Motor 1.8 16v | 2018 | 9146.0 | False | ['Piloto automático', 'Frenos ABS', 'Sensor de... | 10877.62 |
| **Golf** | Motor 1.0 8v | 2018 | 17924.0 | False | ['Llantas de aleación', 'Control de tracción',... | 29240.67 |
| **X1** | Motor Diesel V8 | 1991 | 107.0 | False | ['4 X 4', 'Techo panorámico', 'Aire acondicion... | 20593.75 |
| **Phantom III** | Motor Diesel V6 | 2008 | 4420.0 | False | ['Vidrios eléctricos', 'Bancos de cuero', 'Pil... | 13027.61 |

En la creación de las expresiones pueden ser utilizados los operadores condicionales **or** y **and**, también no son necesarios el uso de paréntesis para separar las expresiones, pero los paréntesis y los operadores **|** y **&** también pueden ser utilizados. El siguiente código es equivalente al código de más abajo: `dataset.query('(Motor == "Motor Diesel") & (Cero_km == True)')`.

```
dataset.query('Motor == "Motor Diesel" and Cero_km == True')
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor |
|---|---|---|---|---|---|---|
| **Sorento** | Motor Diesel | 2020 | NaN | True | ['Sensor de lluvia', 'Cámara de estacionamient... | 16279.87 |
| **Camry** | Motor Diesel | 2020 | NaN | True | ['Cerraduras eléctricas', 'Llantas de aleación... | 27719.45 |
| **Aston Martin Virage** | Motor Diesel | 2020 | NaN | True | ['Cerraduras eléctricas', 'Control de tracción... | 19458.04 |
| **Serie 7 Sedan** | Motor Diesel | 2020 | NaN | True | ['Vidrios eléctricos', 'Cerraduras eléctricas'... | 13507.96 |

El método *query* de *DataFrames* con su configuración *default*, tiene performance un poco superior para conjuntos de datos grandes (arriba de 200.000 líneas).

## Iterando con *DataFrames*

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iterrows.html

En los capítulos sobre secuencias y diccionarios de Python aprendimos cómo iterar en estos objetos accediendo individualmente sus contenidos. Con las estructuras de datos de Pandas también es posible realizar este tipo de operación, pero el comportamiento de la iteración básica va a depender del tipo de objeto. Al iterar sobre una *Series* la iteración patrón irá retornar los valores de la *Series*. Vea el ejemplo de abajo con la *Series* **s**:

```
s = pd.Series(range(5), index=['A', 'B', 'C', 'D', 'E'])
s
```

**Salida:**

```
A    0
B    1
C    2
D    3
E    4
dtype: int64
```

```
for item in s:
    print(item)
```

**Salida:**

```
0
1
2
3
4
```

En el caso de los *DataFrame*, estos siguen un patrón semejante a los diccionarios y si utilizamos una iteración básica con un *DataFrame* tendremos como respuesta las etiquetas de las columnas.

```
for item in dataset:
  print(item)
```

**Salida:**

```
Motor
Año
Kilometraje
Cero_km
Accesorios
Valor
```

Así como los diccionarios, los *DataFrames* también tienen el método `items()` para iterar sobre los pares (clave, valor), solo que en este caso las claves son las etiquetas de las columnas y el valor son *Series* conteniendo los valores de las columnas.

Para hacer la iteración por las líneas de un *DataFrame* existen dos método. El método `iterrows()` que itera por las líneas de un *DataFrame* como pares (índice, *Serie*). Este proceso convierte las líneas en *Series*, lo que puede implicar en pérdida de performance. Acompañe los ejemplos de abajo para entender mejor el funcionamiento del método `iterrows()`.

```
count = 0
for index, row in dataset.iterrows():
    count += 1
    print(index)
    if count >= 5:
        break
```

**Salida:**

```
Jetta Variant
Passat
Crossfox
DS5
```

```
Aston Martin DB4
```

Note que cuando recorremos el *DataFrame* una tupla es retornada para cada línea. En esta tupla el primer valor corresponde al índice de la línea (código de arriba) y el segundo es una *Series* conteniendo las informaciones de la línea para cada columna (código de abajo).

```
count = 0
for index, row in dataset.iterrows():
    count += 1
    print(row, '\n\n')
    if count >= 3:
        break
```

**Salida:**

```
Motor                                          Motor 4.0 Turbo
Año                                                       2003
Kilometraje                                              44410
Cero_km                                                  False
Accesorios     ['Llantas de aleación', 'Cerraduras eléctricas...
Valor                                                  17615.7
Name: Jetta Variant, dtype: object


Motor                                            Motor Diesel
Año                                                      1991
Kilometraje                                              5712
Cero_km                                                 False
Accesorios     ['Pantalla multimedia', 'Techo panorámico', 'F...
Valor                                                  21232.4
Name: Passat, dtype: object


Motor                                          Motor Diesel V8
Año                                                      1990
Kilometraje                                             37123
Cero_km                                                 False
Accesorios     ['Piloto automático', 'Control de estabilidad'...
Valor                                                  14566.4
Name: Crossfox, dtype: object
```

Con este recurso podemos, por ejemplo, construir una nueva variable que sea el resultado de una operación entre variables existentes en el *DataFrame* y que tenga que atender a determinado tipo de condición. El código de abajo es un buen ejemplo de eso.

Vamos a crear una columna "Km_media" que es la razón entre el kilometraje total del vehículo y su edad en años. La edad del vehículo también es una operación de sustracción entre el año actual, en el caso 2020, y el año de fabricación del vehículo. Como sabemos, no es posible realizar una división cuando su denominador es cero, y eso ocurre cuando el año actual es igual al año de fabricación, por eso necesitamos verificar esa condición y caso ocurra, realizar el tratamiento más adecuado para evitar errores.

```
for index, row in dataset.iterrows():
  if(2020 - row['Año'] != 0):
    dataset.loc[index, 'Km_media'] = row['Kilometraje'] / (2020 - row['Año'])
  else:
    dataset.loc[index, 'Km_media'] = 0
```

```
dataset
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media |
|---|---|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 | 2612.352941 |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 | 196.965517 |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 | 1237.433333 |
| **DS5** | Motor 2.4 Turbo | 2020 | NaN | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 | 0.000000 |
| **Aston Martin DB4** | Motor 2.4 Turbo | 2006 | 25757.0 | False | ['Llantas de aleación', '4 X 4', 'Pantalla mul... | 18522.42 | 1839.785714 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **Phantom 2013** | Motor V8 | 2014 | 27505.0 | False | ['Control de estabilidad', 'Piloto automático'... | 10351.92 | 4584.166667 |
| **Cadillac Ciel concept** | Motor V8 | 1991 | 29981.0 | False | ['Bancos de cuero', 'Tablero digital', 'Sensor... | 10333.41 | 1033.827586 |
| **Classe GLK** | Motor 5.0 V8 Bi-Turbo | 2002 | 52637.0 | False | ['Llantas de aleación', 'Control de tracción',... | 13786.81 | 2924.277778 |
| **Aston Martin DB5** | Motor Diesel | 1996 | 7685.0 | False | ['Aire acondicionado', '4 X 4', 'Cambio automá... | 24422.18 | 320.208333 |
| **Macan** | Motor Diesel V6 | 1992 | 50188.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'V... | 18076.29 | 1792.428571 |

258 rows × 7 columns

El método `itertuples()` también itera por las líneas de un *DataFrame* retornando tuplas nombradas de valores. Ese proceso es más rápido que el realizado por el método `iterrows()` y por eso debe tener preferencia, siempre que sea posible, en este tipo de proceso. Abajo tenemos el código para el cálculo de la columna "Km_media" con el uso del método `itertuples()`.

```
for row in dataset.itertuples():
    if(2020 - row.Año != 0):
        dataset.loc[row.Index, 'Km_media'] = row.Kilometraje / (2020 - row.Año)
    else:
        dataset.loc[row.Index, 'Km_media'] = 0
```

```
dataset
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media |
|---|---|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 | 2612.352941 |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 | 196.965517 |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 | 1237.433333 |
| **DS5** | Motor 2.4 Turbo | 2020 | NaN | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 | 0.000000 |
| **Aston Martin DB4** | Motor 2.4 Turbo | 2006 | 25757.0 | False | ['Llantas de aleación', '4 X 4', 'Pantalla mul... | 18522.42 | 1839.785714 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **Phantom 2013** | Motor V8 | 2014 | 27505.0 | False | ['Control de estabilidad', 'Piloto automático'... | 10351.92 | 4584.166667 |
| **Cadillac Ciel concept** | Motor V8 | 1991 | 29981.0 | False | ['Bancos de cuero', 'Tablero digital', 'Sensor... | 10333.41 | 1033.827586 |
| **Classe GLK** | Motor 5.0 V8 Bi-Turbo | 2002 | 52637.0 | False | ['Llantas de aleación', 'Control de tracción',... | 13786.81 | 2924.277778 |
| **Aston Martin DB5** | Motor Diesel | 1996 | 7685.0 | False | ['Aire acondicionado', '4 X 4', 'Cambio automá... | 24422.18 | 320.208333 |
| **Macan** | Motor Diesel V6 | 1992 | 50188.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'V... | 18076.29 | 1792.428571 |

258 rows × 7 columns

En términos de performance, iteraciones con objetos de Pandas son generalmente lentas, principalmente cuando trabajamos con grandes conjuntos de datos, pero en buena parte de los casos

ellas pueden ser evitadas a través de la substitución por otras alternativas como la utilización del método `apply()` en combinación con funciones `lambda` (veremos en las próximas secciones) o funciones NumPy.

## *Series.str*

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.str.html

Trabajar con *Data Science* significa tener que lidiar con grandes volúmenes de información. En los días actuales las informaciones están disponibles en formatos variados como imágenes, tablas, vídeos, textos, etc. Esa gran variedad hace que un científico de datos tenga que conocer y dominar el mayor número posible de técnicas y herramientas para el manejo y tratamiento de los diversos tipos de datos disponibles.

Un formato bastante utilizado para disponibilidad de datos es el texto. En este formato no estructurado tenemos como ejemplo los *posts* de redes sociales (Twitter, Facebook, etc.), e-mails, mensajes, reportes, etc. Para trabajar con datos de este tipo Pandas ofrece una forma de aplicar funciones de *strings* en sus objetos *Series* e *Index*.

En nuestro *dataset* de clase tenemos algunos ejemplos de datos en el formato *string*, como "Motor" y "Accesorios". La variable "Motor" es una *string* simple y no requiere tratamientos más elaborados para que podamos generar un resumen, pero la variable "Accesorios" presenta un cierto desafío a la hora de acceder su contenido. Acompañe los códigos de abajo.

```
dataset.iloc[0, -3]
```

**Salida:**

```
"['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero', 'Aire acond
icionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia']"
```

Accediendo a la columna "Accesorios" para el primer vehículo del *DataFrame* tenemos como respuesta una lista conteniendo los accesorios del vehículo en formato *string*. Note que aparentemente se trata de una lista patrón del Python, pero cuando verificamos su tipo con la función `type()` vemos que en la verdad se trata de una *string*.

```
type(dataset.iloc[0, -3])
```

**Salida:**

```
str
```

Eso hace el acceso a los accesorios de los vehículos una tarea difícil y por eso necesitamos desarrollar una forma de conversión de estas *strings* para listas de Python y, a partir de allí, poder acceder a sus contenidos con los métodos de listas que aprendimos en el capítulo sobre secuencias.

**Métodos de *strings***

**Documentación**: https://docs.python.org/3.6/library/stdtypes.html#string-methods)

Para entender mejor el paso a paso que necesitamos realizar hasta la conversión de la lista en el formato *string* para una lista de Python vamos a seleccionar apenas uno de los registros de la variable "Accesorios" y almacenar esa *string* en una variable llamada **`lista_str`** .

```
lista_str = dataset.iloc[0, -3]
lista_str
```

**Salida:**

```
"['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero', 'Aire acond
icionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia']"
```

Para este método que vamos a utilizar, el primer paso es excluir los corchetes de la "lista" dentro de nuestra *string* **`lista_str`** .

*Strings* son secuencias inmutables de caracteres y estos caracteres pueden ser accedidos con el uso del operador de indexación [] apenas informando su posición (valor de índice), así como en las listas. Eso nos permite realizar *slices* en la *string* **`lista_str`** eliminando el primer carácter "[" y el último "]" de la siguiente forma:

```
lista_str[1:-1]
```

**Salida:**

```
"'Llantas de aleación', 'Cerraduras eléctricas', 'Piloto automático', 'Bancos de cuero', 'Aire acondi
cionado', 'Sensor de estacionamiento', 'Sensor crepuscular', 'Sensor de lluvia'"
```

El segundo paso en este procedimiento es eliminar las comillas simples que limitan cada accesorio dentro de la *string*. Para eso podemos utilizar el método de *strings* `replace()` . Este método recibe como primer argumento el carácter que deseamos encontrar para sustituir en la *string* y en el segundo argumento pasamos el carácter que queremos usar en la sustitución. En nuestro ejemplo vamos a buscar el carácter comillas simples "'" ( una comilla simple dentro de dos comillas dobles) y sustituir por ningún carácter "" (dos comillas dobles). Note que podemos ir encadenando los métodos en apenas una línea de código.

```
lista_str[1:-1].replace("'", "")
```

**Salida:**

```
'Llantas de aleación, Cerraduras eléctricas, Piloto automático, Bancos de cuero, Aire acondicionado,
Sensor de estacionamiento, Sensor crepuscular, Sensor de lluvia'
```

El último paso en este proceso es colocar cada accesorio como ítem de una lista, y eso puede ser hecho con el método `split()` . Este método retorna una lista con los elementos de una secuencia separados por un delimitador específico.

Como podemos ver en el *output* de arriba, nuestros accesorios están separados por los caracteres coma y espacio ", " y este es el parámetro que debemos pasar para el método `split()` . El siguiente

código muestra el resultado de nuestra transformación.

```
lista = lista_str[1:-1].replace("'", "").split(', ')
lista
```

**Salida:**

```
['Llantas de aleación',
 'Cerraduras eléctricas',
 'Piloto automático',
 'Bancos de cuero',
 'Aire acondicionado',
 'Sensor de estacionamiento',
 'Sensor crepuscular',
 'Sensor de lluvia']
```

Ahora tenemos una lista Python y podemos acceder a su contenido con las técnicas que aprendimos en los capítulos anteriores.

```
type(lista)
```

**Salida:**

```
list
```

Muy bien, aprendimos a realizar esa transformación en apenas un ítem que extrajimos de nuestro *DataFrame*. Ahora vamos a hacer todo este procedimiento con todos los ítems de la columna "Accesorios", y haremos eso usando el atributo `str` de las *Series*. Para facilitar nuestro entendimiento vamos a crear una *Series* apenas con el contenido de la columna "Accesorios", pues `str` es un atributo de objetos *Series* e *Index*. Llamaremos esta *Series* de **`Texto`**.

```
Texto = dataset['Accesorios']
Texto
```

**Salida:**

```
Nombre
Jetta Variant            ['Llantas de aleación', 'Cerraduras eléctricas', 'Piloto ...
Passat                   ['Pantalla multimedia', 'Techo panorámico', 'Fre...
Crossfox                 ['Piloto automático', 'Control de estabilidad...
DS5                      ['Cerraduras eléctricas', '4 X 4', 'Vidrios eléctricos...
Aston Martin DB4         ['Llantas de aleación', '4 X 4', 'Pantalla multimedia...
                                                ...
Phantom 2013             ['Control de estabilidad', 'Piloto automátic...
Cadillac Ciel concept    ['Bancos de cuero', 'Tablero digital', 'Sensor ...
Clase GLK                ['Llantas de aleación', 'Control de tracción', 'Câmbi...
Aston Martin DB5         ['Aire acondicionado', '4 X 4', 'Cambio automátic...
Macan                    ['Pantalla multimedia', 'Techo panorámico', 'Vid...
Name: Accesorios, Length: 258, dtype: object
```

Si llamamos `str` en un objeto *Series* o *Index* que contenga objetos *string*, podremos utilizar métodos de *strings* en todos los elementos de esta *Series* o *Index*. Observe como hacer la primera transformación en la *Series* **`Texto`**.

```
Texto.str[1:-1]
```

**Salida:**

```
Nombre
Jetta Variant            'Llantas de aleación', 'Cerraduras eléctricas', 'Piloto a...
Passat                   'Pantalla multimedia', 'Techo panorámico', 'Frei...
Crossfox                 'Piloto automático', 'Control de estabilidad...
DS5                      'Cerraduras eléctricas', '4 X 4', 'Vidrios eléctricos...
Aston Martin DB4         'Llantas de aleación', '4 X 4', 'Pantalla multimedia'...
                                                ...
Phantom 2013             'Control de estabilidad', 'Piloto automático...
Cadillac Ciel concept    'Bancos de cuero', 'Tablero digital', 'Sensor d...
Clase GLK                'Llantas de aleación', 'Control de tracción', 'Cambio...
Aston Martin DB5         'Aire acondicionado', '4 X 4', 'Cambio automático...
Macan                    'Pantalla multimedia', 'Techo panorámico', 'Vidr...
Name: Accesorios, Length: 258, dtype: object
```

Llamamos el atributo `str` y a partir de este punto podemos utilizar los métodos de *string* como si estuviésemos trabajando con una *string* simple. También es posible ir encadenando los métodos, pero no como presentado en el código de abajo. Observe que cuando utilizamos el método `replace()` ninguna alteración fue realizada en la *Series*.

```
Texto.str[1:-1].replace("'", "")
```

**Salida:**

```
Nombre
Jetta Variant            'Llantas de aleación', 'Cerraduras eléctricas', 'Piloto a...
Passat                   'Pantalla multimedia', 'Techo panorámico', 'Frei...
Crossfox                 'Piloto automático', 'Control de estabilidad...
DS5                      'Cerraduras eléctricas', '4 X 4', 'Vidrios eléctricos...
Aston Martin DB4         'Llantas de aleación', '4 X 4', 'Pantalla multimedia'...
                                                ...
Phantom 2013             'Control de estabilidad', 'Piloto automático...
Cadillac Ciel concept    'Bancos de cuero', 'Tablero digital', 'Sensor d...
Clase GLK               'Llantas de aleación', 'Control de tracción', 'Cambio...
Aston Martin DB5         'Aire acondicionado', '4 X 4', 'Cambio automático...
Macan                    'Pantalla multimedia', 'Techo panorámico', 'Vidr...
Name: Accesorios, Length: 258, dtype: object
```

La forma correcta de encadenamiento es, después de la utilización de determinado método, llamar nuevamente el atributo `str` y luego después el nuevo método, como presentado en el código de abajo.

```
Texto.str[1:-1].str.replace("'", "")
```

**Salida:**

```
Nombre
Jetta Variant            Llantas de aleación, Cerraduras eléctricas, Piloto automá...
Passat                   Pantalla multimedia, Techo panorámico, Frenos AB...
Crossfox                 Piloto automático, Control de estabilidad, S...
DS5                      Cerraduras eléctricas, 4 X 4, Vidrios eléctricos, Sen...
Aston Martin DB4         Llantas de aleación, 4 X 4, Pantalla multimedia, Aire c...
                                                ...
Phantom 2013             Control de estabilidad, Piloto automático, C...
Cadillac Ciel concept    Bancos de cuero, Tablero digital, Sensor de chu...
Clases GLK               Llantas de aleación, Control de tracción, Cambio auto...
Aston Martin DB5         Aire acondicionado, 4 X 4, Cambio automático, Fre...
Macan                    Pantalla multimedia, Techo panorámico, Vidrios el...
```

```
Name: Accesorios, Length: 258, dtype: object
```

Finalizando el procedimiento con el método `split()` tenemos como resultado una *Series* de listas de Python.

```
Texto.str[1:-1].str.replace("'", "").str.split(', ')
```

**Salida:**

```
Nombre
Jetta Variant            [Llantas de aleación, Cerraduras eléctricas, Piloto autom...
Passat                   [Pantalla multimedia, Techo panorámico, Frenos A...
Crossfox                 [Piloto automático, Control de estabilidad, ...
DS5                      [Cerraduras eléctricas, 4 X 4, Vidrios eléctricos, Se...
Aston Martin DB4         [Llantas de aleación, 4 X 4, Pantalla multimedia, Ar ...
                                                 ...
Phantom 2013             [Control de estabilidad, Piloto automático, ...
Cadillac Ciel concept    [Bancos de cuero, Tablero digital, Sensor de ch...
Clase GLK               [Llantas de aleación, Control de tracción, Cambio aut...
Aston Martin DB5         [Aire acondicionado, 4 X 4, Cambio automático, Fr...
Macan                    [Pantalla multimedia, Techo panorámico, Vidros e...
Name: Accesorios, Length: 258, dtype: object
```

Realizando todos estos pasos directamente en la columna "Accesorios" y sustituyendo su contenido por el resultado obtenido.

```
dataset['Accesorios'] = dataset['Accesorios'].str[1:-1].str.replace("'", "").str.split(', ')
```

Tenemos así una columna con las listas de accesorios para cada vehículo en nuestro *DataFrame*. Ahora los accesorios pueden ser accedidos de forma simple con el uso de los métodos de listas que aprendimos en el curso.

```
dataset['Accesorios'][0][1]
```

**Salida:**

```
'Cerraduras eléctricas'
```

Un listado de los métodos de *string* vectorizados que podemos utilizar con objetos de Pandas puede ser visualizado en el siguiente *link*: https://pandas.pydata.org/pandas-docs/stable/user_guide/text.html#method-summary.

## 7.5 PARA SABER MÁS...

### `unique()`

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.unique.html

Retorna los valores exclusivos de un objeto *Series* o *Index*. Los ítems únicos son retornados en un *array* NumPy y en el orden que aparecen.

```
dataset.Motor.unique()
```

**Salida:**

```
array(['Motor 4.0 Turbo', 'Motor Diesel', 'Motor Diesel V8',
       'Motor 2.4 Turbo', 'Motor 1.8 16v', 'Motor 1.0 8v',
       'Motor 3.0 32v', 'Motor 5.0 V8 Bi-Turbo', 'Motor Diesel V6',
       'Motor V6', 'Motor V8', 'Motor 2.0 16v'], dtype=object)
```

Con el uso de este método obtenemos una lista de todos los ítems existentes en una *Series* o columna de un *DataFrame*. Eso es bastante útil en la fase de exploración de los datos.

Como la respuesta es en forma de *array* NumPy, podemos crear una nueva *Series* con este resultado.

```
pd.Series(dataset.Motor.unique())
```

**Salida:**

```
0           Motor 4.0 Turbo
1              Motor Diesel
2           Motor Diesel V8
3           Motor 2.4 Turbo
4             Motor 1.8 16v
5              Motor 1.0 8v
6             Motor 3.0 32v
7     Motor 5.0 V8 Bi-Turbo
8           Motor Diesel V6
9                 Motor V6
10                Motor V8
11            Motor 2.0 16v
dtype: object
```

### `drop_duplicates()`

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop_duplicates.html

Puede ser aplicado en los objetos *Index*, *Series* y *DataFrame* del pandas y retorna el objeto con los valores (*Index* y *Series*) o líneas (*DataFrame*) duplicados excluidos.

Los códigos de abajo muestran el funcionamiento del método en una *DataFrame* simple.

```
df = pd.DataFrame({'A': [1, 2, 3, 4, 5, 1], 'B': [1, 5, 4, 3, 5, 1], 'C': [5, 5, 4, 3, 5, 5]})
df
```

**Salida:**

| | A | B | C |
|---|---|---|---|
| **0** | 1 | 1 | 5 |
| **1** | 2 | 5 | 5 |
| **2** | 3 | 4 | 4 |
| **3** | 4 | 3 | 3 |
| **4** | 5 | 5 | 5 |
| **5** | 1 | 1 | 5 |

Note que las líneas con índices 0 y 5 son idénticas.

```
df.drop_duplicates()
```

**Salida:**

| | A | B | C |
|---|---|---|---|
| **0** | 1 | 1 | 5 |
| **1** | 2 | 5 | 5 |
| **2** | 3 | 4 | 4 |
| **3** | 4 | 3 | 3 |
| **4** | 5 | 5 | 5 |

La aplicación del método elimina la última línea del DataFrame.

El método `drop_duplicates` también puede ser aplicado apenas a algunas columnas del *DataFrame*, bastando para eso utilizar el argumento `subset` e informar una lista con las columnas que deben ser consideradas en la comparación. Observe en el código de abajo que `drop_duplicates` posee el argumento `inplace` y cuando deseamos que el método modifique el proprio objeto sin realizar copias debemos configurarlo con *True*.

```
df.drop_duplicates(subset=['B', 'C'], inplace=True)
```

Ejecutando el código de arriba nos quedamos apenas con cuatro líneas en el *DataFrame* de ejemplo.

```
df
```

**Salida:**

| | A | B | C |
|---|---|---|---|
| **0** | 1 | 1 | 5 |
| **1** | 2 | 5 | 5 |
| **2** | 3 | 4 | 4 |
| **3** | 4 | 3 | 3 |

El método `drop_duplicates` es bastante útil en la fase de limpieza y tratamiento de un conjunto de datos.

## 7.6 EJERCICIOS:

Conociendo y agregando información al dataset.

1. Verifica si existen registros duplicados en el dataset. Caso afirmativo, elimina estos registros.

   **Respuesta:**

   ```
   proyecto.shape
   ```

   **Salida:**

   ```
   (31, 10)
   ```

   ```
   proyecto.drop_duplicates().shape
   ```

   **Salida:**

   ```
   (30, 10)
   ```

   ```
   proyecto.drop_duplicates(inplace=True)
   proyecto.shape
   ```

   **Salida:**

   ```
   (30, 10)
   ```

2. Crea una variable para representar el valor del alquiler más el valor del mantenimiento. Llama esa nueva variable de **"ValorTotal"**.

   **Respuesta:**

   ```
   proyecto['ValorTotal'] = proyecto['Valor'] + proyecto['ValorMantenimiento']
   proyecto.head()
   ```

   **Salida:**

| | Tipo | Valor | Barrio | Alcaldia | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | Caracteristicas | ValorTotal |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Departamento | 681.50 | Barrio del Niño Jesús | Tlalpan | 85 | 3 | 2 | 1.0 | 40.7 | [Planta baja, Buena iluminación, Cocina integr... | 722.20 |
| 1 | Departamento | 681.55 | Carola | Alvaro Obregón | 60 | 2 | 2 | 2.0 | 0.0 | [Buena iluminación,Sala Comedor,Cocina Equipad... | 681.55 |
| 2 | Departamento | 864.80 | Condesa | Cuauhtémoc | 70 | 2 | 2 | 1.0 | 84.6 | [Vigilancia 24 horas, Gimnasio, Roof Garden co... | 949.40 |
| 3 | Casa | 1363.00 | Contadero | Cuajimalpa de Morelos | 131 | 3 | 2 | 2.0 | 0.0 | [Sala, Comedor, Cocina integral,Área de lavado... | 1363.00 |
| 4 | Departamento | 705.00 | Del Valle | Benito Juárez | 79 | 2 | 2 | 1.0 | 61.1 | [Recámara con vestidor y baño, Closet,Comedor ... | 766.10 |

3. Crea una variable que representa el valor total del contrato de locación (alquiler + condominio) por metro cuadrado. Llame esa nueva variable de **"ValorM2"**.

   **Respuesta:**

   ```
   proyecto['ValorM2'] = proyecto['ValorTotal'] / proyecto['Area']
   proyecto.head()
   ```

**Salida:**

| | Tipo | Valor | Barrio | Alcaldia | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | Caracteristicas | ValorTotal | ValorM2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Departamento | 681.50 | Barrio del Niño Jesús | Tlalpan | 85 | 3 | 2 | 1.0 | 40.7 | [Planta baja, Buena iluminación, Cocina integr... | 722.20 | 8.496471 |
| 1 | Departamento | 681.55 | Carola | Alvaro Obregón | 60 | 2 | 2 | 2.0 | 0.0 | [Buena iluminación,Sala Comedor,Cocina Equipad... | 681.55 | 11.359167 |
| 2 | Departamento | 864.80 | Condesa | Cuauhtémoc | 70 | 2 | 2 | 1.0 | 84.6 | [Vigilancia 24 horas, Gimnasio, Roof Garden co... | 949.40 | 13.562857 |
| 3 | Casa | 1363.00 | Contadero | Cuajimalpa de Morelos | 131 | 3 | 2 | 2.0 | 0.0 | [Sala, Comedor, Cocina integral,Área de lavado... | 1363.00 | 10.404580 |
| 4 | Departamento | 705.00 | Del Valle | Benito Juárez | 79 | 2 | 2 | 1.0 | 61.1 | [Recámara con vestidor y baño, Closet,Comedor ... | 766.10 | 9.697468 |

4. Transforma el contenido de la variable **"Caracteristicas"** (*strings*) en listas de Python.

**Respuesta:**

```
proyecto['Caracteristicas'] = proyecto['Caracteristicas'].str[1:-1].str.replace("'", "").str.spli
t(', ')
proyecto.head()
```

**Salida:**

| | Tipo | Valor | Barrio | Alcaldia | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | Caracteristicas | ValorTotal | ValorM2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Departamento | 681.50 | Barrio del Niño Jesús | Tlalpan | 85 | 3 | 2 | 1.0 | 40.7 | [Planta baja, Buena iluminación, Cocina integr... | 722.20 | 8.496471 |
| 1 | Departamento | 681.55 | Carola | Alvaro Obregón | 60 | 2 | 2 | 2.0 | 0.0 | [Buena iluminación,Sala Comedor,Cocina Equipad... | 681.55 | 11.359167 |
| 2 | Departamento | 864.80 | Condesa | Cuauhtémoc | 70 | 2 | 2 | 1.0 | 84.6 | [Vigilancia 24 horas, Gimnasio, Roof Garden co... | 949.40 | 13.562857 |
| 3 | Casa | 1363.00 | Contadero | Cuajimalpa de Morelos | 131 | 3 | 2 | 2.0 | 0.0 | [Sala, Comedor, Cocina integral,Área de lavado... | 1363.00 | 10.404580 |
| 4 | Departamento | 705.00 | Del Valle | Benito Juárez | 79 | 2 | 2 | 1.0 | 61.1 | [Recámara con vestidor y baño, Closet,Comedor ... | 766.10 | 9.697468 |

5. Crea una variable booleana que indique si el inmueble tiene como característica **"Vigilancia 24 horas"**. Llama esa nueva variable de **"Vigilancia"**.

**Respuesta:**

```
for index, row in proyecto.iterrows():
    proyecto.loc[index, 'Vigilancia'] = 'Vigilancia 24 horas' in row['Características']

proyecto.head()
```

**Salida:**

| | Tipo | Valor | Barrio | Alcaldia | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | Caracteristicas | ValorTotal | ValorM2 | Vigilancia |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Departamento | 681.50 | Barrio del Niño Jesús | Tlalpan | 85 | 3 | 2 | 1.0 | 40.7 | [Planta baja, Buena iluminación, Cocina integr... | 722.20 | 8.496471 | False |
| 1 | Departamento | 681.55 | Carola | Alvaro Obregón | 60 | 2 | 2 | 2.0 | 0.0 | [Buena iluminación,Sala Comedor,Cocina Equipad... | 681.55 | 11.359167 | True |
| 2 | Departamento | 864.80 | Condesa | Cuauhtémoc | 70 | 2 | 2 | 1.0 | 84.6 | [Vigilancia 24 horas, Gimnasio, Roof Garden co... | 949.40 | 13.562857 | True |
| 3 | Casa | 1363.00 | Contadero | Cuajimalpa de Morelos | 131 | 3 | 2 | 2.0 | 0.0 | [Sala, Comedor, Cocina integral,Área de lavado... | 1363.00 | 10.404580 | True |
| 4 | Departamento | 705.00 | Del Valle | Benito Juárez | 79 | 2 | 2 | 1.0 | 61.1 | [Recámara con vestidor y baño, Closet,Comedor ... | 766.10 | 9.697468 | False |

6. Verifica cuántas categorías de inmuebles tenemos en la variable **"Tipo"**.

**Respuesta:**

```
proyecto.Tipo.unique()
```

**Salida:**

```
array(['Departamento', 'Casa'], dtype=object)
```

7. Haga una consulta para verificar cuántos inmuebles del tipo **"Casa"** existen.

**Respuesta:**

```
len(proyecto[proyecto.Tipo == 'Casa'])
```

**Salida:**

```
9
```

```
proyecto[proyecto.Tipo == 'Casa'].shape
```

**Salida:**

```
(9, 13)
```

```
proyecto.query('Tipo == "Casa"').shape[0]
```

**Salida:**

```
9
```

# 7.7 OBTENCIÓN DE ESTADÍSTICAS DESCRIPTIVAS

**Lista de funciones**: https://pandas.pydata.org/pandas-docs/stable/user_guide/computation.html#method-summary

En esta sección vamos a conocer algunos métodos para el resumen (Summary) estadístico que pueden ser utilizados con las estructuras de datos de Pandas. Son métodos que generalmente retornan un único valor de una *Series* o una *Series* con los resultados para las líneas o columnas de un *DataFrame*.

## Media aritmética

### `mean()`

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.mean.html

El método `mean()` retorna el valor de la media aritmética de un conjunto de datos. Puede ser aplicado en objetos *Series* y *DataFrames*.

Abajo tenemos la media aritmética de la columna "Valor" de nuestro *dataset* de la clase.

```
dataset.Valor.mean()
```

**Salida:**

```
-27737.896007751937
```

Buena parte de los métodos para los principales estadísticos presentan argumentos en común. Considere el *DataFrame* abajo para entender mejor con algunos ejemplos.

```
df = pd.DataFrame([[44410.0, 5712.0, 37123.0, np.nan, 25757.0],
                   [17615.73, 21232.39, 14566.43, 24909.81, 18522.42]],
                  index=['Kilometraje', 'Valor'],
                  columns=['Jetta Variant', 'Passat', 'Crossfox', 'DS5', 'Aston Martin DB4'])
df
```

**Salida:**

| | Jetta Variant | Passat | Crossfox | DS5 | Aston Martin DB4 |
|---|---|---|---|---|---|
| **Kilometraje** | 44410.00 | 5712.00 | 37123.00 | NaN | 25757.00 |
| **Valor** | 17615.73 | 21232.39 | 14566.43 | 24909.81 | 18522.42 |

Tenga en cuenta que el *DataFrame* `df` presenta los vehículos en columnas y sus atributos (kilometraje y valor) en las líneas. Cuando aplicamos el método `mean()`, en su configuración *default*, obtenemos como resultado una *Series* con los valores medios por vehículo, o sea, la función calcula la media aritmética entre las variables kilometraje y valor para cada vehículo y eso no tiene significado práctico pues se tratan de variables distintas y con unidades de medida totalmente diferentes.

```
df.mean()
```

**Salida:**

```
Jetta Variant       31012.865
Passat              13472.195
Crossfox            25844.715
DS5                 24909.810
Aston Martin DB4    22139.710
dtype: float64
```

Lo que necesitamos en este caso es obtener el kilometraje y el valor medio, o mejor dicho, necesitamos realizar el cálculo por el eje de las columnas y no por el eje de las líneas, como fue hecho en el ejemplo anterior. Para eso basta configurar el argumento `axis` para determinar sobre cuál eje se aplicará la operación. El ejemplo de abajo muestra la configuración *default* del método. Podemos

también sustituir el valor 0 por la *string* 'index'.

```
df.mean(axis=0)
```

**Salida:**

```
Jetta Variant       31012.865
Passat              13472.195
Crossfox            25844.715
DS5                 24909.810
Aston Martin DB4    22139.710
dtype: float64
```

Para realizar la operación sobre el eje de las columnas se debe configurar el argumento `axis` con el valor 1 o con la *string* 'columns'. Abajo tenemos el resultado correcto para nuestro análisis.

```
df.mean(axis=1)
```

**Salida:**

```
Kilometraje    28250.500
Valor          19369.356
dtype: float64
```

Otro punto interesante sobre los métodos de "sumarización" estadística y matemática de Pandas es que estos vienen con tratamiento integrado para valores ausentes (NaN). Por *default* los valores NaN son excluidos de las operaciones, pero podemos borrar este tratamiento configurando el argumento `skipna` como *False*. Observe el resultado abajo.

```
df.mean(axis='columns', skipna=False)
```

**Salida:**

```
Kilometraje          NaN
Valor          19369.356
dtype: float64
```

Tenga en cuenta que por presentar un valor NaN en la línea "Kilometraje" el resultado de la aplicación del método `mean()` será también un valor NaN.

## Mediana

### *median()*

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.median.html

El método `median()` retorna el valor de la mediana de un conjunto de datos, o sea, el valor que separa los datos ordenados en dos partes iguales, con 50% de los registros abajo de este valor y 50% de los registros por encima de este valor. Así como la media, la mediana es una medida de tendencia central que nos auxilia en el entendimiento de la distribución de un conjunto de datos. Puede ser aplicado en objetos *Series* y *DataFrames*.

```
dataset['Valor'].median()
```

**Salida:**

```
18964.255
```

Las observaciones sobre los argumentos `axis` y `skipna` hechas en la sección anterior se aplican también al método `median()`.

## Desviación Estándar

### *std()*

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.std.html

El método `std()` retorna el valor de la desviación estándar de un conjunto de datos. La desviación estándar es un importante estadístico que mide el grado de dispersión de un conjunto de datos en relación a su media. Puede ser aplicado en objetos *Series* y *DataFrames*.

```
dataset['Valor'].std()
```

**Salida:**

```
215232.09154321515
```

Abajo tenemos el cálculo de la desviación estándar para una selección de apenas dos variables del *DataFrame* **dataset**.

```
dataset[['Valor', 'Km_media']].std()
```

**Salida:**

```
Valor       215232.091543
Km_media      6945.967606
dtype: float64
```

Los argumentos `axis` y `skipna` funcionan de la misma forma para el método `std()`, pero este método tiene un argumento adicional ( `ddof` ) que indica el número grados de libertad utilizados en la operación. Este asunto sale un poco del alcance de este curso pero lo que debe quedar claro aquí es que cuando utilizamos la configuración default del método ( `ddof=1` ) estamos obteniendo la desviación estándar de los valores de una muestra.

## Cuantiles

Los cuantiles dividen un conjunto de datos ordenados en n subconjuntos de datos de igual tamaño. En realidad, los cuantiles son los valores límites que determinan la división entre los subconjuntos consecutivos de los datos. Un ejemplo es la mediana que divide un conjunto de datos ordenados en dos partes iguales.

Hay una serie de medidas de posición semejantes a la mediana en su composición y algunos de estos cuantiles tienen nombres específicos como los **cuartiles** que dividen un conjunto de datos ordenados en

cuatro partes iguales; los **deciles** dividen los datos en diez partes y los **centiles** en cien partes iguales.

### `quantile()`

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.quantile.html

Para obtener los cuantiles de un conjunto de datos, Pandas tiene a disposición el método `quantile()` que, si es aplicado en su configuración *default*, nos da el valor de la mediana de un conjunto de datos, así como el método `median()` visto en las secciones anteriores.

```
dataset['Valor'].quantile()
```

**Salida:**

```
18964.255
```

El método `quantile()` posee un argumento que permite definir cual o cuales medidas de separación queremos calcular. El parámetro `q` viene por *default* con el valor 0.5, pero puede recibir cualquier valor en el intervalo entre 0 y 1 en formato *float* o en el formato de lista para valores múltiplos, como en el ejemplo de abajo donde calculamos los cuartiles, que dividen un conjunto de datos en cuatro partes iguales.

```
dataset['Valor'].quantile(q = [0.25, 0.5, 0.75])
```

**Salida:**

```
0.25    13646.370
0.50    18964.255
0.75    24478.400
Name: Valor, dtype: float64
```

## Estadísticas descriptivas

### `describe()`

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.describe.html

Para facilitar el procedimiento de análisis el paquete pandas también pone a disposición un método para generar un conjunto de estadísticas descriptivas de una sola vez. Este método es el `describe()` y puede ser aplicado a datos numéricos y no numéricos.

En el ejemplo de abajo son seleccionadas apenas las columnas "Valor" y "Km*media" de nuestro _dataset* y aplicado el método. Note que encadenamos el método `round()` que redondea los resultados para un determinado número de casas decimales, en el caso de abajo, ´para dos casas decimales.

```
dataset[['Valor', 'Km_media']].describe().round(2)
```

**Salida:**

| | Valor | Km_media |
|---|---|---|
| count | 258.00 | 258.00 |
| mean | -27737.90 | 4650.52 |
| std | 215232.09 | 6945.97 |
| min | -999999.00 | 0.00 |
| 25% | 13646.37 | 322.80 |
| 50% | 18964.26 | 2867.56 |
| 75% | 24478.40 | 5306.37 |
| max | 29897.98 | 44330.50 |

En el resultado del método tenemos medidas de tendencia central, dispersión y de separación del conjunto de datos, no considerando los valores de NaN.

Para datos numéricos el resultado presenta un conteo de valores no nulos ( `count` ), media aritmética ( `mean` ), desviación estándar ( `std` ), valor mínimo ( `min` ), valor máximo ( `max` ) y los cuartiles (25%, 50% y 75%).

```
dataset['Valor'].describe().loc[['min', '25%', '75%', 'max']]
```

**Salida:**

```
min   -999999.00
25%     13646.37
75%     24478.40
max     29897.98
Name: Valor, dtype: float64
```

Utilizando el método de selección `loc` podemos escoger apenas un conjunto específico de estadísticas para visualización (código de arriba).

Otra herramienta interesante de Pandas es el método `tolist()` . Con este método es posible transformar una *Series* en una lista común de Python. Eso puede ser útil a la hora de transportar los resultados de un método para otros métodos de pandas.

```
dataset['Valor'].describe().loc[['min', '25%', '75%', 'max']].tolist()
```

**Salida:**

```
[-999999.0, 13646.37, 24478.4, 29897.98]
```

Para datos no numéricos el método presenta *output* diferente, teniendo un contador ( `count` ), un contador de valores exclusivos ( `unique` ), un indicador del valor más frecuente ( `top` ) y la frecuencia de este valor ( `freq` ). En el ejemplo de abajo aplicamos el método `describe()` en la columna "Motor" de nuestro *dataset*.

```
dataset['Motor'].describe()
```

**Salida:**

```
count              258
```

```
unique               12
top       Motor 3.0 32v
freq                 27
Name: Motor, dtype: object
```

## 7.8 EJERCICIOS:

1. Utilizando el método `describe()` calcula las estadísticas descriptivas de las variables numéricas del *DataFrame* **`proyecto`** . Analiza los resultados para decidir cuales variables necesitan de tratamiento.

   **Respuesta:**

   ```
   proyecto.describe()
   ```

   **Salida:**

| | Valor | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | ValorTotal | ValorM2 |
|---|---|---|---|---|---|---|---|---|
| count | 29.000000 | 30.00000 | 30.000000 | 30.000000 | 28.000000 | 24.000000 | 23.000000 | 23.000000 |
| mean | 892.029310 | 123.00000 | 2.466667 | 1.900000 | 1.821429 | 28.108333 | 952.210870 | 8.596309 |
| std | 406.108097 | 80.51301 | 1.041661 | 0.994814 | 0.818923 | 44.867786 | 420.798436 | 3.107233 |
| min | 211.500000 | 30.00000 | 1.000000 | 1.000000 | 1.000000 | -1.000000 | 459.600000 | 3.607692 |
| 25% | 681.500000 | 72.25000 | 2.000000 | 1.000000 | 1.000000 | -1.000000 | 692.775000 | 6.291518 |
| 50% | 752.000000 | 105.50000 | 2.500000 | 2.000000 | 2.000000 | 0.000000 | 766.100000 | 8.496471 |
| 75% | 1081.000000 | 145.25000 | 3.000000 | 2.000000 | 2.000000 | 50.525000 | 1151.500000 | 10.232052 |
| max | 2115.000000 | 385.00000 | 5.000000 | 5.000000 | 4.000000 | 164.500000 | 2114.000000 | 16.507317 |

## 7.9 AGREGACIONES

Las agregaciones son extremadamente importantes en la fase de conocimiento de un conjunto de datos. Con estas herramientas podemos crear estadísticas descriptivas adicionales no ofrecidas por métodos padrones, podemos generar resúmenes teniendo como base cruzamientos de variables, generar tablas de frecuencias personalizadas, entre muchas otras.

Con eso aumentamos nuestra capacidad de identificación de posibles problemas que necesitan ser tratados en los datos.

### El método `aggregate()`

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.aggregate.html)

Este método genera agregaciones de los datos utilizando una o más operaciones en un eje especificado. El código siguiente selecciona las columnas "Valor" y "Km_media" y resume estos datos utilizando las funciones `sum` (suma) y `mean` (media aritmética).

```
dataset[['Valor', 'Km_media']].aggregate(func=['sum', 'mean'])
```

**Salida:**

| | Valor | Km_media |
|---|---|---|
| **sum** | -7.156377e+06 | 1.199834e+06 |
| **mean** | -2.773790e+04 | 4.650519e+03 |

El argumento `func` indica la función que será utilizada en la agregación de los datos y puede ser configurado de una de las siguientes formas:

- La función (ejemplo: `np.sum` );
- Una *string* como el nombre de la función;
- Una lista de funciones o de los nombres de las funciones (ejemplo: `[np.mean, 'std']` );
- Un diccionario con las etiquetas de los ejes como claves y una de las tres opciones arriba como valores (ejemplo: `{'Valor': 'mean', 'Km_media': [np.sum, 'std']}` ).

```
dataset[['Valor', 'Km_media']].agg(func=['sum', 'mean', 'std'])
```

**Salida:**

| | Valor | Km_media |
|---|---|---|
| **sum** | -7.156377e+06 | 1.199834e+06 |
| **mean** | -2.773790e+04 | 4.650519e+03 |
| **std** | 2.152321e+05 | 6.945968e+03 |

Utilizando un diccionario con el argumento `func` no es necesario utilizar el operador de indexación [] o cualquier otro de selección para escoger las variables, basta informar en el diccionario los nombres de los ejes (columnas o líneas) como claves y como valores las funciones para agregación. Note en el resultado que como declaramos apenas la media para la variable "Valor" las líneas de la desviación estándar y sumatoria quedarán con valores NaN. Lo mismo pasa para la variable "Km_media" que se quedó con la línea de la media aritmética con valor NaN.

```
dataset.agg(func={'Valor': 'mean', 'Km_media': [np.sum, 'std']})
```

**Salida:**

| | Valor | Km_media |
|---|---|---|
| **mean** | -27737.896008 | NaN |
| **std** | NaN | 6.945968e+03 |
| **sum** | NaN | 1.199834e+06 |

El argumento `axis` funciona como en los otros métodos de pandas que aprendimos hasta ahora, siendo el *default* la opción 0 o 'index'.

## El método `groupby()`

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.groupby.html

Cuando hablamos en `groupby`, nos estamos refiriendo a un proceso que envuelve una o más de las siguientes etapas:

- Separar (*Split*): En esta etapa, los datos de uno de los objetos de Pandas son separados en grupos con base en una o más claves determinadas;
- Aplicar (*Apply*): Aquí una función es aplicada a cada grupo de forma independiente;
- Combinar (*Combine*): En esa fase los resultados obtenidos de todas las aplicaciones de función son combinados en una estructura de datos.

Imagina que sea necesario obtener la media aritmética de la columna "Valor" según los tipos de motor de los vehículos de nuestro *dataset*. Podemos realizar esa operación de varias formas, y una de ellas es llamando el método `groupby()` y configurando su argumento `by` con la variable que deseamos utilizar como divisor de los grupos ("Motor").

```
grupo = dataset.groupby(by = 'Motor')
grupo
```

**Salida:**

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001EE035E7748>
```

Observe que la variable **`grupo`** que creamos en el código de arriba es un objeto `DataFrameGroupBy` y a partir de él podemos obtener agregaciones apenas especificando cuál o cuáles variables queremos resumir y definiendo el tipo de función que debe ser aplicada. El código de abajo obtiene la media aritmética de la columna "Valor" por cada tipo de motor.

```
grupo[['Valor']].mean().round(2)
```

**Salida:**

| Motor | Valor |
|---|---|
| Motor 1.0 8v | -21078.65 |
| Motor 1.8 16v | 16774.49 |
| Motor 2.0 16v | 18808.26 |
| Motor 2.4 Turbo | 19821.00 |
| Motor 3.0 32v | 18754.56 |
| Motor 4.0 Turbo | 20460.79 |
| Motor 5.0 V8 Bi-Turbo | 18669.24 |
| Motor Diesel | -18154.10 |
| Motor Diesel V6 | -124450.17 |
| Motor Diesel V8 | -77008.64 |
| Motor V6 | -80671.17 |
| Motor V8 | -98538.15 |

El corchete doble del código anterior hace con que el resultado obtenido sea un *DataFrame* y no una *Series*, también fue utilizado el método `round(2)` para redondear los resultados y mejorar la visualización de las informaciones.

Otra característica interesante de los objetos `DataFrameGroupBy` es que podemos tener índices con más de un nivel, o sea, podemos configurar el argumento `by` del método `groupby()` con más de una información. En el ejemplo de abajo pasamos para el argumento `by` una lista con las columnas "Motor" y "Cero*km" de nuestro dataset de aula. En este ejemplo ya creamos una _DataFrame* y almacenamos en la variable **`tabla`** .

```
tabla = dataset.groupby(by = ['Motor', 'Cero_km'])[['Valor']].mean().round(2)
```

Accediendo el índice del *DataFrame* **`tabla`** obtenemos el siguiente resultado:

```
tabla.index
```

**Salida:**

```
MultiIndex([(         'Motor 1.0 8v', False),
            (         'Motor 1.0 8v',  True),
            (        'Motor 1.8 16v', False),
            (        'Motor 1.8 16v',  True),
            (        'Motor 2.0 16v', False),
            (        'Motor 2.0 16v',  True),
            (      'Motor 2.4 Turbo', False),
            (      'Motor 2.4 Turbo',  True),
            (        'Motor 3.0 32v', False),
            (        'Motor 3.0 32v',  True),
            (      'Motor 4.0 Turbo', False),
            (      'Motor 4.0 Turbo',  True),
            ('Motor 5.0 V8 Bi-Turbo', False),
            ('Motor 5.0 V8 Bi-Turbo',  True),
            (         'Motor Diesel', False),
            (         'Motor Diesel',  True),
            (      'Motor Diesel V6', False),
            (      'Motor Diesel V6',  True),
            (      'Motor Diesel V8', False),
            (      'Motor Diesel V8',  True),
            (              'Motor V6', False),
            (              'Motor V6',  True),
            (              'Motor V8', False),
            (              'Motor V8',  True)],
           names=['Motor', 'Cero_km'])
```

Note que ahora tenemos un objeto *Index* un poco diferente de los que vimos en el comienzo de este capítulo. En este caso tenemos un *MultiIndex* que es un índice con más de un nivel. Observe el formato de *DataFarme* del resultado.

```
tabla
```

**Salida:**

| | | Valor |
|---|---|---|
| Motor | Cero_km | |
| Motor 1.0 8v | False | -27039.46 |
| | True | 22633.89 |
| Motor 1.8 16v | False | 17011.88 |
| | True | 15587.56 |
| Motor 2.0 16v | False | 18999.81 |
| | True | 17275.85 |
| Motor 2.4 Turbo | False | 19925.10 |
| | True | 19664.85 |
| Motor 3.0 32v | False | 18011.86 |
| | True | 20518.48 |
| Motor 4.0 Turbo | False | 21286.85 |
| | True | 17844.91 |
| Motor 5.0 V8 Bi-Turbo | False | 17381.16 |
| | True | 24143.56 |
| Motor Diesel | False | -24953.27 |
| | True | 19241.33 |
| Motor Diesel V6 | False | -35663.01 |
| | True | -657173.15 |
| Motor Diesel V8 | False | -48065.87 |
| | True | -149365.58 |
| Motor V6 | False | -38594.65 |
| | True | -319104.78 |
| Motor V8 | False | -160977.15 |
| | True | 19402.17 |

## El método `value_counts()`

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.value_counts.html

Otra forma de obtener los agrupamientos con Pandas es a través del método `value_counts()`. Este método retorna un objeto que contiene las frecuencias de los valores exclusivos de determinada variable. El resultado de este método es lo que conocemos en estadística como distribución de frecuencias de una variable.

```
dataset['Motor'].value_counts()
```

**Salida:**

```
Motor 3.0 32v            27
Motor Diesel             26
Motor V8                 26
Motor 4.0 Turbo          25
Motor 1.0 8v             25
Motor Diesel V8          21
Motor Diesel V6          21
Motor 5.0 V8 Bi-Turbo    21
Motor V6                 20
Motor 2.0 16v            18
Motor 1.8 16v            18
Motor 2.4 Turbo          10
Name: Motor, dtype: int64
```

El objeto resultante estará en orden decreciente de frecuencias para que el primer elemento sea el elemento de mayor frecuencia, pero este comportamiento puede ser modificado con el argumento `sort` que por *default* viene configurado con *True*.

Otro argumento de configuración bastante útil del método es el `normalize` (*default False*). Con este argumento indicamos si la distribución de frecuencias debe tener el formato porcentual ( `normalize=True` ) o no (*default*).

```
dataset['Motor'].value_counts(normalize = True)
```

**Salida:**

```
Motor 3.0 32v            0.104651
Motor Diesel             0.100775
Motor V8                 0.100775
Motor 4.0 Turbo          0.096899
Motor 1.0 8v             0.096899
Motor Diesel V8          0.081395
Motor Diesel V6          0.081395
Motor 5.0 V8 Bi-Turbo    0.081395
Motor V6                 0.077519
Motor 2.0 16v            0.069767
Motor 1.8 16v            0.069767
Motor 2.4 Turbo          0.038760
Name: Motor, dtype: float64
```

El método `value_counts()` también tiene el argumento `dropna` que por *default* es *True*, o sea, valores NaN son excluidos del conteo por patrón.

## 7.10 PARA SABER MÁS...

**`Series.to_frame()`**

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.to_frame.html

Convierte una *Series* en un *DataFrame*.

**`Series.rename_axis()`**

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.rename_axis.html

Define un nombre para el índice.

Observa un ejemplo simple de utilización de estos dos métodos junto con el método `value_counts()` para mejorar la visualización de nuestros resultados.

```
dataset['Motor'].value_counts(normalize = True)
```

**Salida:**

```
Motor 3.0 32v            0.104651
Motor Diesel             0.100775
Motor V8                 0.100775
Motor 4.0 Turbo          0.096899
Motor 1.0 8v             0.096899
Motor Diesel V8          0.081395
Motor Diesel V6          0.081395
Motor 5.0 V8 Bi-Turbo    0.081395
Motor V6                 0.077519
Motor 2.0 16v            0.069767
Motor 1.8 16v            0.069767
Motor 2.4 Turbo          0.038760
Name: Motor, dtype: float64
```

Aplicando el método `rename_axis()` podemos asignar un nombre para la columna que representa el índice. Eso ayuda al usuario de la información a entender mejor los resultados.

```
dataset['Motor'].value_counts(normalize = True).rename_axis('Tipos de Motores')
```

**Salida:**

```
Tipos de Motores
Motor 3.0 32v            0.104651
Motor Diesel             0.100775
Motor V8                 0.100775
Motor 4.0 Turbo          0.096899
Motor 1.0 8v             0.096899
Motor Diesel V8          0.081395
Motor Diesel V6          0.081395
Motor 5.0 V8 Bi-Turbo    0.081395
Motor V6                 0.077519
Motor 2.0 16v            0.069767
Motor 1.8 16v            0.069767
Motor 2.4 Turbo          0.038760
Name: Motor, dtype: float64
```

Para mejorar un poco más el resultado anterior podemos transformar la *Series* de retorno en un *DataFrame* y al mismo tiempo asignar una etiqueta para la columna de frecuencias.

```
dataset['Motor'].value_counts(normalize=True).rename_axis('Tipos de Motores').to_frame('Percentual (x
/100)')
```

**Salida:**

| Tipos de Motores | Percentual (x/100) |
|---|---|
| Motor 3.0 32v | 0.104651 |
| Motor Diesel | 0.100775 |
| Motor V8 | 0.100775 |
| Motor 4.0 Turbo | 0.096899 |
| Motor 1.0 8v | 0.096899 |
| Motor Diesel V6 | 0.081395 |
| Motor 5.0 V8 Bi-Turbo | 0.081395 |
| Motor Diesel V8 | 0.081395 |
| Motor V6 | 0.077519 |
| Motor 1.8 16v | 0.069767 |
| Motor 2.0 16v | 0.069767 |
| Motor 2.4 Turbo | 0.038760 |

El resultado de arriba también puede ser presentado de la siguiente forma:

```
dataset['Motor'].value_counts(normalize=True).rename_axis('Tipos de Motores').to_frame('Percentual (%)') * 100
```

**Salida:**

| Tipos de Motores | Percentual (%) |
|---|---|
| Motor 3.0 32v | 10.465116 |
| Motor Diesel | 10.077519 |
| Motor V8 | 10.077519 |
| Motor 4.0 Turbo | 9.689922 |
| Motor 1.0 8v | 9.689922 |
| Motor Diesel V6 | 8.139535 |
| Motor 5.0 V8 Bi-Turbo | 8.139535 |
| Motor Diesel V8 | 8.139535 |
| Motor V6 | 7.751938 |
| Motor 1.8 16v | 6.976744 |
| Motor 2.0 16v | 6.976744 |
| Motor 2.4 Turbo | 3.875969 |

## 7.11 EJERCICIOS:

1. Obtenga una tabla que informe los valores medios de las variables "ValorTotal" y "Area" según el tipo de inmueble (apartamento o casa).

**Respuesta:**

```
proyecto.groupby(by = ['Tipo'])[['ValorTotal', 'Area']].mean().round(2)
```

**Salida:**

| Tipo | ValorTotal | Area |
|---|---|---|
| Casa | 944.80 | 168.56 |
| Departamento | 955.45 | 103.48 |

2. Haga el mismo procedimiento del ítem anterior sustituyendo la variable "Tipo" por la variable "Barrio". Guarde el resultado en una variable llamada **`valor_medio_barrios`** .

**Respuesta:**

```
valor_medio_barrios = proyecto.groupby(by = ['Barrio'])[['ValorTotal', 'Area']].mean().round(2)
valor_medio_barrios
```

**Salida:**

| Barrio | ValorTotal | Area |
|---|---|---|
| Ampliación Paraje San Juan | NaN | 129.0 |
| Anzures | NaN | 349.0 |
| Barrio del Niño Jesús | 722.20 | 85.0 |
| Carola | 681.55 | 60.0 |
| Condesa | 949.40 | 70.0 |
| ... | ... | ... |
| Reforma Social | NaN | 117.0 |
| Roma Sur | 752.00 | 85.0 |
| San Pedro de los Pinos | 845.00 | 84.0 |
| Santa Fe Cuajimalpa | 1089.90 | 87.0 |
| Vertiz Narvarte | 1315.50 | 197.5 |

24 rows × 2 columns

3. Crea una visualización de la tabla creada en el ítem anterior **`valor_medio_barrios`** solamente con los siguientes barrios:

```
['Carola', 'Del Valle', 'Condesa', 'Moctezuma', 'Santa Fe Cuajimalpa']
```

**Respuesta:**

```
barrios = ['Carola', 'Del Valle', 'Condesa', 'Moctezuma', 'Santa Fe Cuajimalpa']
valor_medio_barrios.loc[barrios]
```

**Salida:**

| Barrio | ValorTotal | Area |
|---|---|---|
| Carola | 681.55 | 60.0 |
| Del Valle | 828.77 | 90.0 |
| Condesa | 949.40 | 70.0 |
| Moctezuma | NaN | 30.0 |
| Santa Fe Cuajimalpa | 1089.90 | 87.0 |

4. Crea una tabla de frecuencias para la variable "Tipo".

   **Sugerencias**: utilize las sugerencias de la sección "para saber más..." para mejorar la visualización de los resultados.

   **Respuesta:**

```
proyecto['Tipo'].value_counts(sort=False)
```

   **Salida:**

```
Casa             9
Departamento    21
Name: Tipo, dtype: int64
```

```
proyecto['Tipo'].value_counts(sort=False).rename_axis('Tipos').to_frame('Frecuencias')
```

   **Salida:**

| Tipos | Frecuencias |
|---|---|
| Casa | 9 |
| Departamento | 21 |

5. Crea 3 tablas de frecuencias con valores porcentuales para las variables "Recámaras", "Baños" y "Estacionamientos". Multiplique los resultados por 100.

   **Respuesta:**

```
proyecto['Recámaras'].value_counts(normalize=True, sort=False).rename_axis('Número de recámaras')
.to_frame('Porcentaje (%)') * 100
```

   **Salida:**

| Número de recamaras | Porcentaje (%) |
|---|---|
| 1 | 20.000000 |
| 2 | 30.000000 |
| 3 | 36.666667 |
| 4 | 10.000000 |
| 5 | 3.333333 |

```
proyecto['Baños'].value_counts(normalize=True, sort=False).rename_axis('Número de baños').to_fram
e('Porcentaje (%)') * 100
```

   **Salida:**

| Numero de baños | Porcentaje (%) |
|---|---|
| 1 | 40.000000 |
| 2 | 40.000000 |
| 3 | 13.333333 |
| 4 | 3.333333 |
| 5 | 3.333333 |

```
proyecto['Estacionamientos'].value_counts(normalize=True, sort=False).rename_axis('Número de estacionamientos').to_frame('Porcentaje (%)') * 100
```

**Salida:**

| Numero de estacionamientos | Porcentaje (%) |
|---|---|
| 1.0 | 39.285714 |
| 2.0 | 42.857143 |
| 3.0 | 14.285714 |
| 4.0 | 3.571429 |

6. Crea una tabla con el metro cuadrado medio según el tipo de inmueble y la cantidad de recámaras.

**Respuesta:**

```
proyecto.groupby(by = ['Tipo', 'Recamaras'])[['Area']].mean()
```

**Salida:**

| Tipo | Recamaras | Area |
|---|---|---|
| Casa | 3 | 174.000000 |
| | 4 | 137.333333 |
| | 5 | 235.000000 |
| Departamento | 1 | 58.666667 |
| | 2 | 88.777778 |
| | 3 | 170.333333 |

## 7.12 TRATAMIENTO BÁSICO DE LOS DATOS

La fase de tratamiento de los datos es una de las más importantes de un proyecto en *Data Science*. En esta etapa el científico de datos ya realizó una exploración básica y tiene un cierto conocimiento de las informaciones que está trabajando y de los problemas que debe tener que tratar.

Esa etapa también debe llevar en consideración el tipo de proyecto que será desarrollado y con eso los tipos de tratamientos que deben ser aplicados.

Los primeros pasos comienzan con herramientas simples, como la propia visualización de partes del *dataset* para identificación de los tipos de datos que están disponibles. Eso puede ser hecho con las herramientas de selección que aprendimos o con los métodos `head()` y `tail()` que nos muestran, respectivamente, las partes superior y la inferior de un *DataFrame*.

```
dataset.head()
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media |
|---|---|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 | 2612.352941 |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 | 196.965517 |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 | 1237.433333 |
| **DS5** | Motor 2.4 Turbo | 2020 | NaN | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 | 0.000000 |
| **Aston Martin DB4** | Motor 2.4 Turbo | 2006 | 25757.0 | False | ['Llantas de aleación', '4 X 4', 'Pantalla mul... | 18522.42 | 1839.785714 |

Otra herramienta simple y bastante útil en la fase inicial de tratamiento es el *output* del método `info()` . Este método tiene como respuesta, informaciones sobre el *DataFrame* como los tipos de variables, cuales columnas presentan valores nulos y uso de memoria.

```
dataset.info()
```

**Salida:**

```
<class 'pandas.core.frame.DataFrame'>
Index: 258 entries, Jetta Variant to Macan
Data columns (total 7 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Motor        258 non-null    object
 1   Año          258 non-null    int64
 2   Kilometraje  203 non-null    float64
 3   Cero_km      258 non-null    bool
 4   Accesorios   258 non-null    object
 5   Valor        258 non-null    float64
 6   Km_media     258 non-null    float64
dtypes: bool(1), float64(3), int64(1), object(2)
memory usage: 24.4+ KB
```

## Tratamiento para datos ausentes

### Identificando los registros

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.isna.html

Uno de los problemas más comunes que un científico de datos tiene que resolver en el tratamiento de las informaciones, son los datos ausentes. Eso ocurre por una serie de razones que pueden ir desde errores en la colecta de los datos hasta una simple falta de aplicación de algún requisito específico del *dataset*. Felizmente Pandas nos ofrece un conjunto de métodos sencillos para la identificación y tratamiento de este tipo de problema.

En la parte de identificación de valores *missing* tenemos el método `isna()`. Este método retorna un objeto de Pandas booleano, del mismo tamaño del objeto original, indicando cuáles valores son NA. los valores NA, como *None* o `numpy.nan`, son mapeados para valores *True*. Todo el resto es mapeado para valores *False*. Caracteres como *strings* vacías "" no son considerados valores NA.

Veamos un ejemplo:

```
dataset.Kilometraje.isna()
```

**Salida:**

```
Nombre
Jetta Variant            False
Passat                   False
Crossfox                 False
DS5                       True
Aston Martin DB4         False
                         ...
Phantom 2013             False
Cadillac Ciel concept    False
Clase GLK               False
Aston Martin DB5         False
Macan                    False
Name: Kilometraje, Length: 258, dtype: bool
```

Volviendo al *output* del método `info()` podemos ver que nuestro *DataFrame* tiene 258 registros (líneas) y nuestra variable "Kilometraje" tiene apenas 197 registros no nulos (197 *non-null*). Eso nos muestra que debemos investigar esta variable y decidir qué tipo de tratamiento utilizar para estos valores ausentes. El código de arriba muestra el resultado de la aplicación del método `isna()` en una *Series* con apenas los valores de la columna "Kilometraje".

Note que tenemos como respuesta una *Series* booleana que marca como *True* las líneas que presentan valores ausentes y, como ya aprendimos, podemos utilizar esta *Series* para tener acceso apenas a los registros que presentan este tipo de problema (código de abajo).

Observe en el *output* de abajo, apenas los vehículos con variable "Cero*km" igual a _True* presentan valores NaN en la variable "Kilometraje". Volveremos a este problema cuando hablemos sobre el tratamiento de valores *missing*.

```
dataset[dataset.Kilometraje.isna()]
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media |
|---|---|---|---|---|---|---|---|
| **DS5** | Motor 2.4 Turbo | 2020 | NaN | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 | 0.0 |
| **A5** | Motor 4.0 Turbo | 2020 | NaN | True | ['Cambio automático', 'Cámara de estacionamien... | 11289.04 | 0.0 |
| **A3** | Motor 1.0 8v | 2020 | NaN | True | ['4 X 4', 'Piloto automático', 'Pantalla multi... | 17710.48 | 0.0 |
| **Serie 1 M** | Motor V8 | 2020 | NaN | True | ['Control de estabilidad', 'Pantalla multimedi... | 18912.88 | 0.0 |
| **Lamborghini Murciélago** | Motor 5.0 V8 Bi-Turbo | 2020 | NaN | True | ['Frenos ABS', 'Cambio automático', 'Aire acon... | 24319.24 | 0.0 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **Lamborghini Reventón** | Motor 4.0 Turbo | 2020 | NaN | True | ['Control de tracción', 'Aire acondicionado', ... | 13532.97 | 0.0 |
| **Benni Mini** | Motor V8 | 2020 | NaN | True | ['Sensor crepuscular', 'Cambio automático', 'C... | 25249.57 | 0.0 |
| **Uno** | Motor Diesel V6 | 2020 | NaN | True | ['Pantalla multimedia', 'Sensor crepuscular', ... | -999999.00 | 0.0 |
| **Santa Fe** | Motor 3.0 32v | 2020 | NaN | True | ['Cerraduras eléctricas', 'Aire acondicionado'... | 25883.07 | 0.0 |
| **XC60** | Motor 4.0 Turbo | 2020 | NaN | True | ['Tablero digital', 'Piloto automático', 'Pant... | 15535.16 | 0.0 |

55 rows × 7 columns

Como vimos, el método `isna()` es una herramienta bastante útil en la tarea de identificación de valores ausentes, pero veremos a partir de ahora que este método no funciona para todos los casos.

Volviendo al *output* del método `info()` percibimos, por ejemplo, que la columna "Valor" tiene 258 registros no nulos y este es exactamente el tamaño total de registros de nuestro *dataset*.

```
dataset.info()
```

**Salida:**

```
<class 'pandas.core.frame.DataFrame'>
Index: 258 entries, Jetta Variant to Macan
Data columns (total 7 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   Motor        258 non-null    object 
 1   Año          258 non-null    int64  
 2   Kilometraje  203 non-null    float64
 3   Cero_km      258 non-null    bool   
 4   Accesorios   258 non-null    object 
 5   Valor        258 non-null    float64
 6   Km_media     258 non-null    float64
dtypes: bool(1), float64(3), int64(1), object(2)
memory usage: 24.4+ KB
```

Cuando buscamos registros con datos ausentes en esta variable la respuesta es la misma, o sea, un *DataFrame* vacío, y eso nos pasa la impresión de que todo está correcto con esta información.

```
dataset[dataset.Valor.isna()]
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media |
|---|---|---|---|---|---|---|---|

Pero cuando realizamos una investigación más detallada en la variable, verificando sus estadísticas

descriptivas básicas con el método `describe()`, notamos una característica sospechosa en la información de valor mínimo. Vea que se trata de un valor negativo y compuesto de seis números nueve (-999999).

Este tipo de característica puede ser encontrada en diversos conjuntos de datos públicos como una forma alternativa de representar valores nulos. Grande parte de estos datos poseen un diccionario elaborado por el responsable por la información para indicar todas las características del *dataset* y de sus variables, pero eso no siempre es una regla y el investigador debe mantener atención a los detalles, principalmente en datos obtenidos a partir de procedimientos automáticos de colecta (*web scraping* por ejemplo).

```
dataset['Valor'].describe().round()
```

**Salida:**

```
count       258.0
mean     -27738.0
std      215232.0
min     -999999.0
25%       13646.0
50%       18964.0
75%       24478.0
max       29898.0
Name: Valor, dtype: float64
```

Haciendo una *query* básica en el *DataFrame* y buscando apenas los registros con el valor -999999 en la columna "Valor" tenemos como respuesta un *DataFrame* pequeño, pero que debe recibir el tratamiento adecuado para no generar problemas futuros.

```
dataset.query('Valor == -999999')
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media |
|---|---|---|---|---|---|---|---|
| **Up!** | Motor Diesel V6 | 2020 | NaN | True | ['Sensor de estacionamiento', 'Vidrios eléctri... | -999999.0 | 0.000000 |
| **Kangoo Express** | Motor Diesel | 2007 | 29132.0 | False | ['Bancos de cuero', 'Cambio automático', 'Pilo... | -999999.0 | 2240.923077 |
| **Celer Hatch** | Motor V8 | 2008 | 113808.0 | False | ['Control de estabilidad', 'Cambio automático'... | -999999.0 | 9484.000000 |
| **Cadillac XTS sedan** | Motor Diesel V6 | 1993 | 90924.0 | False | ['Aire acondicionado', 'Cambio automático', 'S... | -999999.0 | 3367.555556 |
| **Cadillac BLS** | Motor V8 | 1993 | 88634.0 | False | ['Techo panorámico', 'Control de estabilidad',... | -999999.0 | 3282.740741 |
| **Aston Martin DB7** | Motor Diesel V8 | 2005 | 52189.0 | False | ['Cerraduras eléctricas', 'Tablero digital', '... | -999999.0 | 3479.266667 |
| **Vantage Volante** | Motor Diesel V8 | 2020 | NaN | True | ['Frenos ABS', 'Control de tracción', 'Pantall... | -999999.0 | 0.000000 |
| **Serie 1 Coupé** | Motor 1.0 8v | 1993 | 87161.0 | False | ['Control de estabilidad', 'Techo panorámico',... | -999999.0 | 3228.185185 |
| **J3 Turin** | Motor V6 | 2020 | NaN | True | ['Sensor crepuscular', 'Cámara de estacionamie... | -999999.0 | 0.000000 |
| **Cadillac CTS coupe; sedan; and wagon** | Motor V8 | 2006 | 67475.0 | False | ['Sensor de lluvia', 'Sensor de estacionament... | -999999.0 | 4819.642857 |
| **Uno** | Motor Diesel V6 | 2020 | NaN | True | ['Pantalla multimedia', 'Sensor crepuscular', ... | -999999.0 | 0.000000 |
| **RS5** | Motor V6 | 1996 | 55083.0 | False | ['Tablero digital', 'Cambio automático', 'Vidr... | -999999.0 | 2295.125000 |

Una solución simple para este problema es sustituir los valores -999999 por una representación válida de punto flotante para *Not a Number* (NaN) que Pandas pueda identificar con su método

`isna()`.

El código de abajo utiliza dos herramientas sencillas para esta tarea. La primera es el `np.nan` que utiliza el paquete NumPy para generar un valor NaN válido y la segunda es el método `replace()` que puede ser aplicado en objetos de Pandas para sustituir valores.

```
import numpy as np

dataset['Valor'].replace(-999999, np.nan, inplace = True)
```

Después de la aplicación del código de arriba podemos realizar la consulta nuevamente y verificar la presencia de valores NaN en la columna "Valor".

```
dataset[dataset.Valor.isna()]
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media |
|---|---|---|---|---|---|---|---|
| **Up!** | Motor Diesel V6 | 2020 | NaN | True | ['Sensor de estacionamiento', 'Vidrios eléctri... | NaN | 0.000000 |
| **Kangoo Express** | Motor Diesel | 2007 | 29132.0 | False | ['Bancos de cuero', 'Cambio automático', 'Pilo... | NaN | 2240.923077 |
| **Celer Hatch** | Motor V8 | 2008 | 113808.0 | False | ['Control de estabilidad', 'Cambio automático'... | NaN | 9484.000000 |
| **Cadillac XTS sedan** | Motor Diesel V6 | 1993 | 90924.0 | False | ['Aire acondicionado', 'Cambio automático', 'S... | NaN | 3367.555556 |
| **Cadillac BLS** | Motor V8 | 1993 | 88634.0 | False | ['Techo panorámico', 'Control de estabilidad',... | NaN | 3282.740741 |
| **Aston Martin DB7** | Motor Diesel V8 | 2005 | 52189.0 | False | ['Cerraduras eléctricas', 'Tablero digital', '... | NaN | 3479.266667 |
| **Vantage Volante** | Motor Diesel V8 | 2020 | NaN | True | ['Frenos ABS', 'Control de tracción', 'Pantall... | NaN | 0.000000 |
| **Serie 1 Coupé** | Motor 1.0 8v | 1993 | 87161.0 | False | ['Control de estabilidad', 'Techo panorámico',... | NaN | 3228.185185 |
| **J3 Turin** | Motor V6 | 2020 | NaN | True | ['Sensor crepuscular', 'Cámara de estacionamie... | NaN | 0.000000 |
| **Cadillac CTS coupe; sedan; and wagon** | Motor V8 | 2006 | 67475.0 | False | ['Sensor de lluvia', 'Sensor de estacionament... | NaN | 4819.642857 |
| **Uno** | Motor Diesel V6 | 2020 | NaN | True | ['Pantalla multimedia', 'Sensor crepuscular', ... | NaN | 0.000000 |
| **RS5** | Motor V6 | 1996 | 55083.0 | False | ['Tablero digital', 'Cambio automático', 'Vidr... | NaN | 2295.125000 |

### Sustituyendo valores ausentes

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html

En la sección anterior identificamos los valores *missing* en un objeto de Pandas, ahora necesitamos de herramientas para tratar, cuando sea necesario, estos valores. Vamos a comenzar por la sustitución de valores NaN por valores válidos con el uso del método `fillna()`.

Para nuestro ejemplo, volvamos al problema de la variable "Kilometraje" que presenta valores NaN cuando el vehículo es cero kilometro. En este tipo de situación el profesional de *Data Science* tiene que decidir, basado en los objetivos del proyecto, qué tipo de tratamiento aplicar a los datos. En este caso específico vamos a optar por mantener las informaciones en el *dataset* para sustituir los valores nulos de esta variable por valores válidos. Como se trata de vehículos cero kilómetro, vamos a asignar el valor cero cuando el vehículo tenga "Kilometraje" NaN.

```
dataset.fillna(value = {'Kilometraje': 0}, inplace = True)
```

Creando una *query* para mostrar todos los vehículos cero kilómetro del *DataFrame*, tenemos el resultado de la utilización del método `fillna()`.

```
dataset.query("Cero_km == True")
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media |
|---|---|---|---|---|---|---|---|
| **DS5** | Motor 2.4 Turbo | 2020 | 0.0 | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 | 0.0 |
| **A5** | Motor 4.0 Turbo | 2020 | 0.0 | True | ['Cambio automático', 'Cámara de estacionamien... | 11289.04 | 0.0 |
| **A3** | Motor 1.0 8v | 2020 | 0.0 | True | ['4 X 4', 'Piloto automático', 'Pantalla multi... | 17710.48 | 0.0 |
| **Serie 1 M** | Motor V8 | 2020 | 0.0 | True | ['Control de estabilidad', 'Pantalla multimedi... | 18912.88 | 0.0 |
| **Lamborghini Murciélago** | Motor 5.0 V8 Bi-Turbo | 2020 | 0.0 | True | ['Frenos ABS', 'Cambio automático', 'Aire acon... | 24319.24 | 0.0 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **Lamborghini Reventón** | Motor 4.0 Turbo | 2020 | 0.0 | True | ['Control de tracción', 'Aire acondicionado', ... | 13532.97 | 0.0 |
| **Benni Mini** | Motor V8 | 2020 | 0.0 | True | ['Sensor crepuscular', 'Cambio automático', 'C... | 25249.57 | 0.0 |
| **Uno** | Motor Diesel V6 | 2020 | 0.0 | True | ['Pantalla multimedia', 'Sensor crepuscular', ... | NaN | 0.0 |
| **Santa Fe** | Motor 3.0 32v | 2020 | 0.0 | True | ['Cerraduras eléctricas', 'Aire acondicionado'... | 25883.07 | 0.0 |
| **XC60** | Motor 4.0 Turbo | 2020 | 0.0 | True | ['Tablero digital', 'Piloto automático', 'Pant... | 15535.16 | 0.0 |

55 rows × 7 columns

### Eliminando registros con valores ausentes

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html

Algunas variables presentan ciertas características que hacen imposible, o bastante complicada, la sustitución de sus valores NaN. En estos casos el investigador puede optar por mantener las informaciones nulas en el *dataset*, visto que buena parte de los métodos de tratamiento y modelado estadístico de Python ya poseen tratamiento integrado para este tipo de dato, o eliminar los registros con esas características. Pandas ofrece el método `dropna()` que es específico para esta tarea.

Suponiendo que sea necesario, en nuestro proyecto de clase, eliminar los registros que presenten valores nulos para la columna "Valor". El método `dropna()` posee el argumento `axis` que por *default* viene configurado para actuar en las líneas (axis: 0 o 'index'), pero puede ser modificado para actuar en las columnas (axis: 1 o 'columns'). Para eliminar los registros que presenten valores nulos en la columna "Valor" con el método `dropna()` basta informar para el argumento `subset` una lista con el nombre de la columna ( `subset=['Valor']` ).

```
dataset.dropna(subset = ['Valor'], inplace = True)
```

Un cuidado que debe ser tomado en la utilización de este método es el siguiente. Caso sea aplicado en un *DataFrame* en su configuración *default* ( `DataFrame.dropna()` ), todos los registros que presenten valores NaN, en cualquier columna, serán eliminados del *DataFrame*.

## Modificaciones en los datos

### El método `apply()`

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.apply.html

Cuando hablamos sobre iteraciones en objetos de Pandas citamos el método `apply()` como una herramienta alternativa al uso de bucles *for*, principalmente por cuestiones de performance.

Este método simplemente nos permite aplicar una función a lo largo de un eje específico del *DataFrame*. Puede ser utilizado apenas como un agregador, teniendo como argumento una de las funciones básicas de Pandas, de NumPy o funciones personalizadas, como las funciones *lambda* que aprendimos en el capítulo sobre funciones de Python.

```
dataset[['Valor', 'Kilometraje']].apply('mean')
```

**Salida:**

```
Valor          19689.474919
Kilometraje    44694.369919
dtype: float64
```

Este método también puede ser utilizado como constructor de nuevas variables. En este contexto el método `apply()` funciona como una herramienta de iteración a través de los ejes de los objetos Pandas con una performance superior a los bucles *for* que ya estudiamos.

Como ejemplo de utilización del método `apply()` juntamente con funciones *lambda* vamos a reconstruir la variable "Km_media" que, después de los tratamientos realizados en la variable "Kilometraje", necesita ser recalculada para corregir los registros con informaciones nulas.

```
dataset
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media |
|---|---|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 | 2612.352941 |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 | 196.965517 |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 | 1237.433333 |
| **DS5** | Motor 2.4 Turbo | 2020 | 0.0 | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 | 0.000000 |
| **Aston Martin DB4** | Motor 2.4 Turbo | 2006 | 25757.0 | False | ['Llantas de aleación', '4 X 4', 'Pantalla mul... | 18522.42 | 1839.785714 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **Phantom 2013** | Motor V8 | 2014 | 27505.0 | False | ['Control de estabilidad', 'Piloto automático'... | 10351.92 | 4584.166667 |
| **Cadillac Ciel concept** | Motor V8 | 1991 | 29981.0 | False | ['Bancos de cuero', 'Tablero digital', 'Sensor... | 10333.41 | 1033.827586 |
| **Clase GLK** | Motor 5.0 V8 Bi-Turbo | 2002 | 52637.0 | False | ['Llantas de aleación', 'Control de tracción',... | 13786.81 | 2924.277778 |
| **Aston Martin DB5** | Motor Diesel | 1996 | 7685.0 | False | ['Aire acondicionado', '4 X 4', 'Cambio automá... | 24422.18 | 320.208333 |
| **Macan** | Motor Diesel V6 | 1992 | 50188.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'V... | 18076.29 | 1792.428571 |

246 rows × 7 columns

El primer paso debe ser la construcción de la función *lambda* para el cálculo del kilometraje medio. Note que el argumento `data` (puede ser cualquier nombre válido de Python) representa el *DataFrame*

y con él podemos acceder a las informaciones de cada columna.

```
# Función para obtener el kilometraje medio de los vehículos.
f = lambda data: data['Kilometraje'] / (2020 - data['Año']) if 2020 - data['Año'] != 0 else 0
```

Como tenemos una función que necesita ser aplicada línea por línea, así como un bucle *for* patrón, necesitamos configurar el argumento `axis` del método `apply()` con el valor 1 o 'columns' para garantizar este comportamiento. En el método `apply()` el argumento `axis` funciona de la siguiente forma:

**axis** (*default* 0): Eje a lo largo del cual la función es aplicada:

- **0 o 'index'**: aplica la función a cada columna.
- **1 o 'columns'**: aplica la función a cada línea.

Sigue la aplicación del método para la construcción de la variable "Km_media".

```
dataset['Km_media'] = dataset.apply(f, axis = 1)
dataset
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media |
|---|---|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 | 2612.352941 |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 | 196.965517 |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 | 1237.433333 |
| **DS5** | Motor 2.4 Turbo | 2020 | 0.0 | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 | 0.000000 |
| **Aston Martin DB4** | Motor 2.4 Turbo | 2006 | 25757.0 | False | ['Llantas de aleación', '4 X 4', 'Pantalla mul... | 18522.42 | 1839.785714 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **Phantom 2013** | Motor V8 | 2014 | 27505.0 | False | ['Control de estabilidad', 'Piloto automático'... | 10351.92 | 4584.166667 |
| **Cadillac Ciel concept** | Motor V8 | 1991 | 29981.0 | False | ['Bancos de cuero', 'Tablero digital', 'Sensor... | 10333.41 | 1033.827586 |
| **Clase GLK** | Motor 5.0 V8 Bi-Turbo | 2002 | 52637.0 | False | ['Llantas de aleación', 'Control de tracción',... | 13786.81 | 2924.277778 |
| **Aston Martin DB5** | Motor Diesel | 1996 | 7685.0 | False | ['Aire acondicionado', '4 X 4', 'Cambio automá... | 24422.18 | 320.208333 |
| **Macan** | Motor Diesel V6 | 1992 | 50188.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'V... | 18076.29 | 1792.428571 |

246 rows × 7 columns

### El método `map`

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.map.html

El método `map()` es una herramienta interesante de Pandas que puede ser utilizada para facilitar la creación de variables en un *dataset*. Funciona como un "mapeador" de valores y puede ser usado para sustituir cada valor en una *Series* por otro valor, que puede ser derivado de un diccionario. Considere el diccionario **combustible** de abajo. Note que se trata de una correspondencia entre tipo de motor (clave) y tipo de combustible (valor).

```
combustible = {
    'Motor 1.0 8v': 'Flex',
    'Motor 1.8 16v': 'Flex',
```

```
    'Motor 2.0 16v': 'Flex',
    'Motor 2.4 Turbo': 'Flex',
    'Motor 3.0 32v': 'Gasolina',
    'Motor 4.0 Turbo': 'Gasolina',
    'Motor 5.0 V8 Bi-Turbo': 'Diesel',
    'Motor Diesel': 'Diesel',
    'Motor Diesel V6': 'Diesel',
    'Motor Diesel V8': 'Diesel',
    'Motor V6': 'Gasolina',
    'Motor V8': 'Gasolina'
}
```

Utilizando el método `map()` en una *Series* con los valores de la columna "Motor" y pasando como argumento el diccionario **combustible** tenemos como resultado una *Series* que hace la correspondencia entre el tipo de motor y el tipo de combustible.

```
dataset['Motor'].map(combustible)
```

**Salida:**

```
Nombre
Jetta Variant            Gasolina
Passat                     Diesel
Crossfox                   Diesel
DS5                          Flex
Aston Martin DB4             Flex
                            ...
Phantom 2013             Gasolina
Cadillac Ciel concept    Gasolina
Clase GLK                  Diesel
Aston Martin DB5           Diesel
Macan                      Diesel
Name: Motor, Length: 246, dtype: object
```

Esto puede ser utilizado para agregar información a nuestro *DataFrame* de vehículos con la creación de una nueva variable para indicar el tipo de combustible de cada vehículo.

```
dataset['Combustible'] = dataset['Motor'].map(combustible)
dataset
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media | Combustible |
|---|---|---|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 | 2612.352941 | Gasolina |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 | 196.965517 | Diesel |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 | 1237.433333 | Diesel |
| **DS5** | Motor 2.4 Turbo | 2020 | 0.0 | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 | 0.000000 | Flex |
| **Aston Martin DB4** | Motor 2.4 Turbo | 2006 | 25757.0 | False | ['Llantas de aleación', '4 X 4', 'Pantalla mul... | 18522.42 | 1839.785714 | Flex |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **Phantom 2013** | Motor V8 | 2014 | 27505.0 | False | ['Control de estabilidad', 'Piloto automático'... | 10351.92 | 4584.166667 | Gasolina |
| **Cadillac Ciel concept** | Motor V8 | 1991 | 29981.0 | False | ['Bancos de cuero', 'Tablero digital', 'Sensor... | 10333.41 | 1033.827586 | Gasolina |
| **Clase GLK** | Motor 5.0 V8 Bi-Turbo | 2002 | 52637.0 | False | ['Llantas de aleación', 'Control de tracción',... | 13786.81 | 2924.277778 | Diesel |
| **Aston Martin DB5** | Motor Diesel | 1996 | 7685.0 | False | ['Aire acondicionado', '4 X 4', 'Cambio automá... | 24422.18 | 320.208333 | Diesel |
| **Macan** | Motor Diesel V6 | 1992 | 50188.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'V... | 18076.29 | 1792.428571 | Diesel |

246 rows × 8 columns

### Eliminando líneas y columnas de un *DataFrame*

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html

También puede ser necesario, durante el tratamiento de los datos, la eliminación de determinadas líneas o columnas por otros motivos que no sean solamente la presencia de valores ausentes. Veamos algunas formas de realizar estas tareas.

#### Eliminando líneas - filtros

Un método que ya conocemos para la remoción de líneas es a través de la utilización de filtros simples en los datos.

```
filtro = dataset['Valor'] > 5000000
```

Observe que el filtro de arriba selecciona apenas los vehículos con valores arriba de 5.000.000 y el código de abajo utiliza este filtro con el operador de negación (~) para seleccionar todos los vehículos con valores menores o iguales a 5.000.000. Observe que el operador de negación (~) invierte los valores de una Series booleana, o sea, quien es *True* pasa a ser *False* y quien es *False* pasa a ser *True*.

```
dataset = dataset[~filtro]
```

#### Eliminando líneas o columnas - método `drop()`

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html

Otra técnica de remoción es con el uso del método `drop()` que remueve líneas o columnas apenas con la especificación de los nombres de *labels* o ejes correspondientes.

Suponga que sea necesario eliminar los vehículos con valores por encima de 1.000.000. Para eso podemos identificar esas informaciones y obtener las etiquetas de índices de ellas, de la siguiente forma:

```
filtro = dataset[dataset.Valor > 1000000].index
filtro
```

**Salida:**

```
Index(['Fox'], dtype='object', name='Nombre')
```

Para la consulta de arriba obtenemos apenas un registro con el *label* del índice igual a "Fox". Pasando esa información para el argumento `index` del método `drop()` tenemos como resultado un *DataFrame* sin la información para el vehículo de label "Fox".

```
dataset.drop(index=filtro, inplace=True)
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media | Combustible |
|---|---|---|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 | 2612.352941 | Gasolina |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 | 196.965517 | Diesel |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 | 1237.433333 | Diesel |
| **DS5** | Motor 2.4 Turbo | 2020 | 0.0 | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 | 0.000000 | Flex |
| **Aston Martin DB4** | Motor 2.4 Turbo | 2006 | 25757.0 | False | ['Llantas de aleación', '4 X 4', 'Pantalla mul... | 18522.42 | 1839.785714 | Flex |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **Phantom 2013** | Motor V8 | 2014 | 27505.0 | False | ['Control de estabilidad', 'Piloto automático'... | 10351.92 | 4584.166667 | Gasolina |
| **Cadillac Ciel concept** | Motor V8 | 1991 | 29981.0 | False | ['Bancos de cuero', 'Tablero digital', 'Sensor... | 10333.41 | 1033.827586 | Gasolina |
| **Clase GLK** | Motor 5.0 V8 Bi-Turbo | 2002 | 52637.0 | False | ['Llantas de aleación', 'Control de tracción',... | 13786.81 | 2924.277778 | Diesel |
| **Aston Martin DB5** | Motor Diesel | 1996 | 7685.0 | False | ['Aire acondicionado', '4 X 4', 'Cambio automá... | 24422.18 | 320.208333 | Diesel |
| **Macan** | Motor Diesel V6 | 1992 | 50188.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'V... | 18076.29 | 1792.428571 | Diesel |

245 rows × 8 columns

Para remover columnas con el método `drop()` basta pasar una lista con el nombre de las columnas y configurar el argumento `axis` con valor 1 o 'columns'.

```
dataset.drop(['Combustible'], axis = 'columns', inplace=True)
dataset.head()
```

**Salida:**

| Nombre | Motor | Año | Kilometraje | Cero_km | Accesorios | Valor | Km_media |
|---|---|---|---|---|---|---|---|
| **Jetta Variant** | Motor 4.0 Turbo | 2003 | 44410.0 | False | ['Llantas de aleación', 'Cerraduras eléctricas... | 17615.73 | 2612.352941 |
| **Passat** | Motor Diesel | 1991 | 5712.0 | False | ['Pantalla multimedia', 'Techo panorámico', 'F... | 21232.39 | 196.965517 |
| **Crossfox** | Motor Diesel V8 | 1990 | 37123.0 | False | ['Piloto automático', 'Control de estabilidad'... | 14566.43 | 1237.433333 |
| **DS5** | Motor 2.4 Turbo | 2020 | 0.0 | True | ['Cerraduras eléctricas', '4 X 4', 'Vidrios el... | 24909.81 | 0.000000 |
| **Aston Martin DB4** | Motor 2.4 Turbo | 2006 | 25757.0 | False | ['Llantas de aleación', '4 X 4', 'Pantalla mul... | 18522.42 | 1839.785714 |

El siguiente código es equivalente al código de arriba: `dataset.drop(columns=['Combustible'], inplace=True)`

## 7.13 EJERCICIOS:

1. Haga una evaluación inicial de los datos utilizando los métodos `info()` y `describe()`.

**Respuesta:**

```
proyecto.head()
```

**Salida:**

| | Tipo | Valor | Barrio | Alcaldia | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | Caracteristicas | ValorTotal | ValorM2 | Vigilancia |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Departamento | 681.50 | Barrio del Niño Jesús | Tlalpan | 85 | 3 | 2 | 1.0 | 40.7 | [Planta baja, Buena iluminación, Cocina integr... | 722.20 | 8.496471 | False |
| 1 | Departamento | 681.55 | Carola | Alvaro Obregón | 60 | 2 | 2 | 2.0 | 0.0 | [Buena iluminación,Sala Comedor,Cocina Equipad... | 681.55 | 11.359167 | True |
| 2 | Departamento | 864.80 | Condesa | Cuauhtémoc | 70 | 2 | 2 | 1.0 | 84.6 | [Vigilancia 24 horas, Gimnasio, Roof Garden co... | 949.40 | 13.562857 | True |
| 3 | Casa | 1363.00 | Contadero | Cuajimalpa de Morelos | 131 | 3 | 2 | 2.0 | 0.0 | [Sala, Comedor, Cocina integral,Área de lavado... | 1363.00 | 10.404580 | True |
| 4 | Departamento | 705.00 | Del Valle | Benito Juárez | 79 | 2 | 2 | 1.0 | 61.1 | [Recámara con vestidor y baño, Closet,Comedor ... | 766.10 | 9.697468 | False |

```
proyecto.info()
```

**Salida:**

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 30 entries, 0 to 29
Data columns (total 13 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   Tipo                30 non-null     object 
 1   Valor               29 non-null     float64
 2   Barrio              30 non-null     object 
 3   Alcaldia            30 non-null     object 
 4   Area                30 non-null     int64  
 5   Recamaras           30 non-null     int64  
 6   Baños               30 non-null     int64  
 7   Estacionamientos    28 non-null     float64
 8   ValorMantenimiento  24 non-null     float64
 9   Caracteristicas     30 non-null     object 
 10  ValorTotal          23 non-null     float64
 11  ValorM2             23 non-null     float64
 12  Vigilancia          30 non-null     object 
dtypes: float64(5), int64(3), object(5)
memory usage: 4.5+ KB
```

```
proyecto.describe()
```

**Salida:**

| | Valor | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | ValorTotal | ValorM2 |
|---|---|---|---|---|---|---|---|---|
| **count** | 29.000000 | 30.00000 | 30.000000 | 30.000000 | 28.000000 | 24.000000 | 23.000000 | 23.000000 |
| **mean** | 892.029310 | 123.00000 | 2.466667 | 1.900000 | 1.821429 | 28.108333 | 952.210870 | 8.596309 |
| **std** | 406.108097 | 80.51301 | 1.041661 | 0.994814 | 0.818923 | 44.867786 | 420.798436 | 3.107233 |
| **min** | 211.500000 | 30.00000 | 1.000000 | 1.000000 | 1.000000 | -1.000000 | 459.600000 | 3.607692 |
| **25%** | 681.500000 | 72.25000 | 2.000000 | 1.000000 | 1.000000 | -1.000000 | 692.775000 | 6.291518 |
| **50%** | 752.000000 | 105.50000 | 2.500000 | 2.000000 | 2.000000 | 0.000000 | 766.100000 | 8.496471 |
| **75%** | 1081.000000 | 145.25000 | 3.000000 | 2.000000 | 2.000000 | 50.525000 | 1151.500000 | 10.232052 |
| **max** | 2115.000000 | 385.00000 | 5.000000 | 5.000000 | 4.000000 | 164.500000 | 2114.000000 | 16.507317 |

2. Evalúa los problemas encontrados en la variable "Valor" (valor del alquiler). Decide y ejecuta el mejor tratamiento. Al final del tratamiento ejecuta nuevamente el método `describe()` para verificar el resultado obtenido para la variable "Valor" y para las demás variables.

**Respuesta:**

```
proyecto[proyecto.Valor.isna()].shape
```

**Salida:**

```
(1, 13)
```

```
proyecto.dropna(subset = ['Valor'], inplace = True)
```

```
proyecto[proyecto.Valor.isna()].shape
```

**Salida:**

```
(0, 13)
```

```
proyecto.describe()
```

**Salida:**

| | Valor | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | ValorTotal | ValorM2 |
|---|---|---|---|---|---|---|---|---|
| count | 29.000000 | 29.000000 | 29.000000 | 29.000000 | 27.000000 | 23.000000 | 23.000000 | 23.000000 |
| mean | 892.029310 | 115.206897 | 2.448276 | 1.896552 | 1.814815 | 29.373913 | 952.210870 | 8.596309 |
| std | 406.108097 | 69.475268 | 1.055131 | 1.012240 | 0.833761 | 45.436061 | 420.798436 | 3.107233 |
| min | 211.500000 | 30.000000 | 1.000000 | 1.000000 | 1.000000 | -1.000000 | 459.600000 | 3.607692 |
| 25% | 681.500000 | 70.000000 | 2.000000 | 1.000000 | 1.000000 | -1.000000 | 692.775000 | 6.291518 |
| 50% | 752.000000 | 94.000000 | 2.000000 | 2.000000 | 2.000000 | 0.000000 | 766.100000 | 8.496471 |
| 75% | 1081.000000 | 131.000000 | 3.000000 | 2.000000 | 2.000000 | 54.050000 | 1151.500000 | 10.232052 |
| max | 2115.000000 | 385.000000 | 5.000000 | 5.000000 | 4.000000 | 164.500000 | 2114.000000 | 16.507317 |

3. Repite el procedimiento del ítem anterior para la variable "ValorMantenimiento". Note que el valor **-1** indica que el inmueble no tiene valor de mantenimiento, o sea, el valor es **0**.

**Respuesta:**

```
proyecto[proyecto.ValorMantenimiento.isna()].shape
```

**Salida:**

```
(6, 13)
```

```
proyecto.dropna(subset = ['ValorMantenimiento'], inplace = True)
```

```
proyecto[proyecto.ValorMantenimiento.isna()].shape
```

**Salida:**

```
(0, 13)
```

```
proyecto[proyecto.ValorMantenimiento == -1].shape
```

**Salida:**

```
(9, 13)
```

```
proyecto['ValorMantenimiento'].replace(-1, 0, inplace = True)
```

```
proyecto[proyecto.ValorMantenimiento == -1].shape
```

**Salida:**

```
(0, 13)
```

```
proyecto.describe()
```

**Salida:**

| | Valor | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | ValorTotal | ValorM2 |
|---|---|---|---|---|---|---|---|---|
| count | 23.000000 | 23.000000 | 23.000000 | 23.000000 | 22.000000 | 23.000000 | 23.000000 | 23.000000 |
| mean | 922.836957 | 123.739130 | 2.652174 | 2.043478 | 1.818182 | 29.765217 | 952.210870 | 8.596309 |
| std | 415.972479 | 73.547892 | 0.934622 | 1.065076 | 0.852803 | 45.164514 | 420.798436 | 3.107233 |
| min | 460.600000 | 41.000000 | 1.000000 | 1.000000 | 1.000000 | 0.000000 | 459.600000 | 3.607692 |
| 25% | 681.525000 | 74.500000 | 2.000000 | 1.000000 | 1.000000 | 0.000000 | 692.775000 | 6.291518 |
| 50% | 752.000000 | 120.000000 | 3.000000 | 2.000000 | 2.000000 | 0.000000 | 766.100000 | 8.496471 |
| 75% | 1104.500000 | 151.500000 | 3.000000 | 2.500000 | 2.000000 | 54.050000 | 1151.500000 | 10.232052 |
| max | 2115.000000 | 385.000000 | 5.000000 | 5.000000 | 4.000000 | 164.500000 | 2114.000000 | 16.507317 |

4. Realice el tratamiento para las variable "Estacionamiento" y "Baños". Observa que cuando el inmueble no posee una de estas características el valor de la variable viene como nulo. Sustituya los valores nulos por ceros.

**Respuesta:**

```
proyecto.fillna(value = {'Estacionamiento': 0,'Baños':0}, inplace = True)
```

```
proyecto.describe()
```

**Salida:**

| | Valor | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | ValorTotal | ValorM2 |
|---|---|---|---|---|---|---|---|---|
| count | 23.000000 | 23.000000 | 23.000000 | 23.000000 | 22.000000 | 23.000000 | 23.000000 | 23.000000 |
| mean | 922.836957 | 123.739130 | 2.652174 | 2.043478 | 1.818182 | 29.765217 | 952.210870 | 8.596309 |
| std | 415.972479 | 73.547892 | 0.934622 | 1.065076 | 0.852803 | 45.164514 | 420.798436 | 3.107233 |
| min | 460.600000 | 41.000000 | 1.000000 | 1.000000 | 1.000000 | 0.000000 | 459.600000 | 3.607692 |
| 25% | 681.525000 | 74.500000 | 2.000000 | 1.000000 | 1.000000 | 0.000000 | 692.775000 | 6.291518 |
| 50% | 752.000000 | 120.000000 | 3.000000 | 2.000000 | 2.000000 | 0.000000 | 766.100000 | 8.496471 |
| 75% | 1104.500000 | 151.500000 | 3.000000 | 2.500000 | 2.000000 | 54.050000 | 1151.500000 | 10.232052 |
| max | 2115.000000 | 385.000000 | 5.000000 | 5.000000 | 4.000000 | 164.500000 | 2114.000000 | 16.507317 |

5. Recalcule la variable "ValorTotal" y "ValorM2".

**Respuesta:**

```
proyecto['ValorTotal'] = proyecto['Valor'] + proyecto['ValorMantenimiento']
proyecto['ValorM2'] = proyecto['ValorTotal'] / proyecto['Area']
```

```
proyecto.describe()
```

**Salida:**

| | Valor | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | ValorTotal | ValorM2 |
|---|---|---|---|---|---|---|---|---|
| count | 23.000000 | 23.000000 | 23.000000 | 23.000000 | 22.000000 | 23.000000 | 23.000000 | 23.000000 |
| mean | 922.836957 | 123.739130 | 2.652174 | 2.043478 | 1.818182 | 29.765217 | 952.602174 | 8.599673 |
| std | 415.972479 | 73.547892 | 0.934622 | 1.065076 | 0.852803 | 45.164514 | 420.792489 | 3.106404 |
| min | 460.600000 | 41.000000 | 1.000000 | 1.000000 | 1.000000 | 0.000000 | 460.600000 | 3.615385 |
| 25% | 681.525000 | 74.500000 | 2.000000 | 1.000000 | 1.000000 | 0.000000 | 693.275000 | 6.294643 |
| 50% | 752.000000 | 120.000000 | 3.000000 | 2.000000 | 2.000000 | 0.000000 | 766.100000 | 8.496471 |
| 75% | 1104.500000 | 151.500000 | 3.000000 | 2.500000 | 2.000000 | 54.050000 | 1151.500000 | 10.238004 |
| max | 2115.000000 | 385.000000 | 5.000000 | 5.000000 | 4.000000 | 164.500000 | 2115.000000 | 16.507317 |

6. Utilizando el método `apply()` , crea una variable que indique si el inmueble tiene Gimnasio. Para crear esta columna utiliza la variable "Caracteristicas" y procure por el ítem "Gimnasio". La nueva variable debe ser del tipo booleana y tener como etiqueta de columna "Gimnasio".

**Respuesta:**

```
proyecto['Gimnasio'] = proyecto['Caracteristicas'].apply(lambda x: 'Gimnasio' in x)
proyecto.head()
```

**Salida:**

| | Tipo | Valor | Barrio | Alcaldia | Area | Recamaras | Baños | Estacionamientos | ValorMantenimiento | Caracteristicas | ValorTotal | ValorM2 | Vigilancia | Gimnasio |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Departamento | 681.50 | Barrio del Niño Jesús | Tlalpan | 85 | 3 | 2 | 1.0 | 40.7 | [Planta baja, Buena iluminación, Cocina integr... | 722.20 | 8.496471 | False | False |
| 1 | Departamento | 681.55 | Carola | Alvaro Obregón | 60 | 2 | 2 | 2.0 | 0.0 | [Buena iluminación,Sala Comedor,Cocina Equipad... | 681.55 | 11.359167 | True | False |
| 2 | Departamento | 864.80 | Condesa | Cuauhtémoc | 70 | 2 | 2 | 1.0 | 84.6 | [Vigilancia 24 horas, Gimnasio, Roof Garden co... | 949.40 | 13.562857 | True | True |
| 3 | Casa | 1363.00 | Contadero | Cuajimalpa de Morelos | 131 | 3 | 2 | 2.0 | 0.0 | [Sala, Comedor, Cocina integral,Área de lavado... | 1363.00 | 10.404580 | True | True |
| 4 | Departamento | 705.00 | Del Valle | Benito Juárez | 79 | 2 | 2 | 1.0 | 61.1 | [Recámara con vestidor y baño, Closet,Comedor ... | 766.10 | 9.697468 | False | False |

7. Verifique el porcentaje de inmuebles con y sin gimnasio en el *dataset*.

**Respuesta:**

```
proyecto['Gimnasio'].value_counts(normalize = True)
```

**Salida:**

```
False    0.869565
True     0.130435
Name: Gimnasio, dtype: float64
```

# 7.14 GENERANDO VISUALIZACIONES CON EL PANDAS

**Documentación**: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html **Documentación**: https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html

El uso de gráficos es una de las herramientas más importantes en un proceso de análisis de datos. Las visualizaciones tienen un papel importante tanto en la fase exploratoria de los datos como en la

divulgación de los resultados de un proyecto. En la fase exploratoria el uso de gráficos nos ayuda en la evaluación del comportamiento de una variable, en la identificación de datos discrepantes y de posibles transformaciones necesarias en los datos. Ya en la fase de presentación de los resultados permite resumir el contenido de forma sencilla, objetiva e informativa.

La biblioteca Pandas, utilizando otra biblitoeca llamada Matplotlib (https://matplotlib.org/) como base, tiene métodos integrados que permiten la creación de gráficos a partir de *DataFrames* y *Series*. Matplotlib es una biblioteca especializada en la creación de visualizaciones estáticas, animadas e interactivas en Python.

Como se trata de un asunto amplio, nuestro objetivo en este curso es apenas de presentación básica del recurso de visualización que podemos utilizar con Pandas. Y para eso, utilizando nuestro *dataset* de clase, vamos a construir algunos análisis gráficos para ejemplificar el uso de esta herramienta.

Los objetos *Series* y *DataFrame* de pandas presentan un atributo `plot` que permite la creación de algunos tipos básicos de gráficos. Este atributo `plot` es apenas un método simple que pertenece al paquete Matplotlib. Eso nos permite acceder a los recursos de la biblioteca Matplotlib y utilizar el contenido de los objetos Pandas para construcción de visualizaciones.

## Gráfico de líneas

Los gráficos de líneas son extensamente utilizados en análisis que envuelven datos temporales. Entonces, para ejemplificar el uso de este tipo de gráfico vamos a crear una serie temporal con la ayuda de nuestro *DataFrame* **dataset**.

El código a seguir crea una *Series* con el resultado de una agregación utilizando el método `groupby()` . Aquí estamos calculando el valor medio de los vehículos según los respectivos años de fabricación.

```
valor_medio = dataset.groupby('Año')['Valor'].mean()
valor_medio
```

**Salida:**

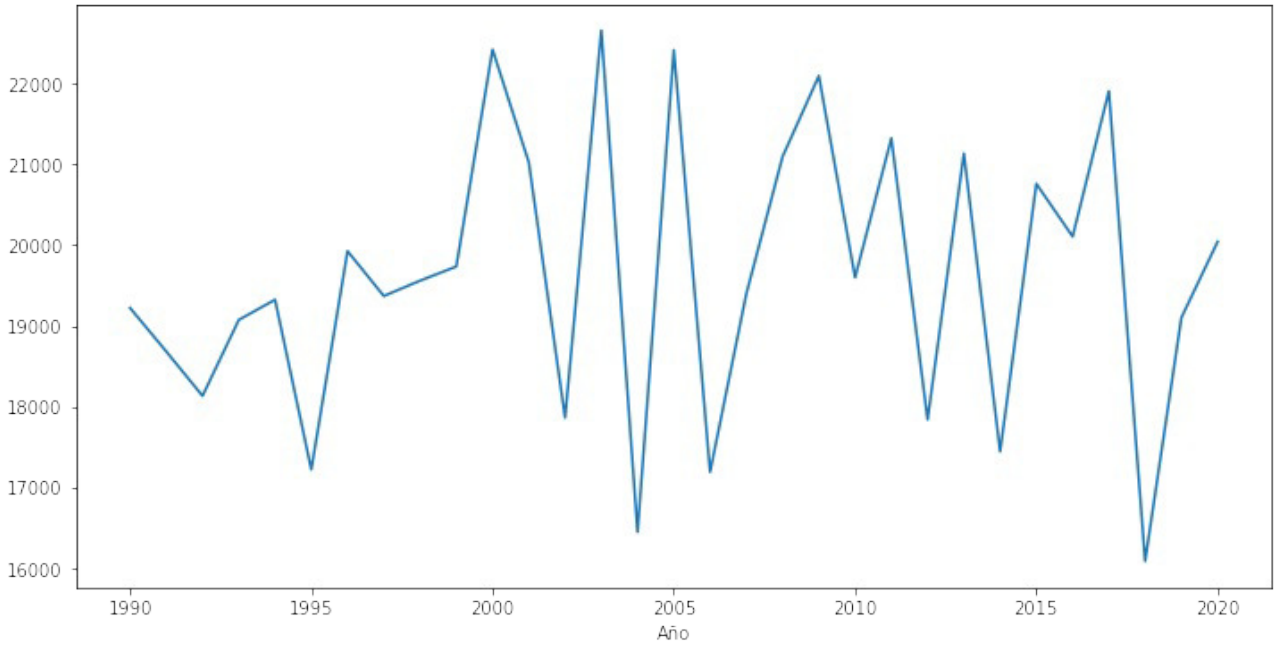
```
Año
1990    19223.241667
1991    18684.131111
1992    18134.132500
1993    19075.042000
1994    19324.665556
1995    17223.540000
1996    19925.744444
1997    19370.576667
1998    19560.383333
1999    19735.045000
2000    22420.383333
2001    21024.150000
2002    17865.502727
2003    22657.083333
```

```
2004    16450.664000
2005    22413.784000
2006    17194.130000
2007    19398.276364
2008    21095.458000
2009    22094.122857
2010    19600.206667
2011    21322.648000
2012    17842.440000
2013    21134.401111
2014    17445.723333
2015    20759.021250
2016    20106.235000
2017    21906.714286
2018    16090.452000
2019    19102.343333
2020    20043.542157
Name: Valor, dtype: float64
```

Por default el método `plot()` genera gráficos de líneas como el del ejemplo de abajo. En este código fue utilizado apenas el argumento `figsize` que es un parámetro de configuración para definir el tamaño de la imagen de salida ( `figsize=(largura, altura)` ).

```
valor_medio.plot(figsize=(12, 6))
```

**Salida:**




Note que la construcción del gráfico es muy simple, basta llamar el método `plot()` de una *Series* o *DataFrame*. Recordando que si el *DataFrame* tuviera más de una columna debemos especificar cual de ellas debe ser utilizada en la construcción del gráfico (ejemplo: `DataFrame['Rótulo de la Columna'].plot()` ).

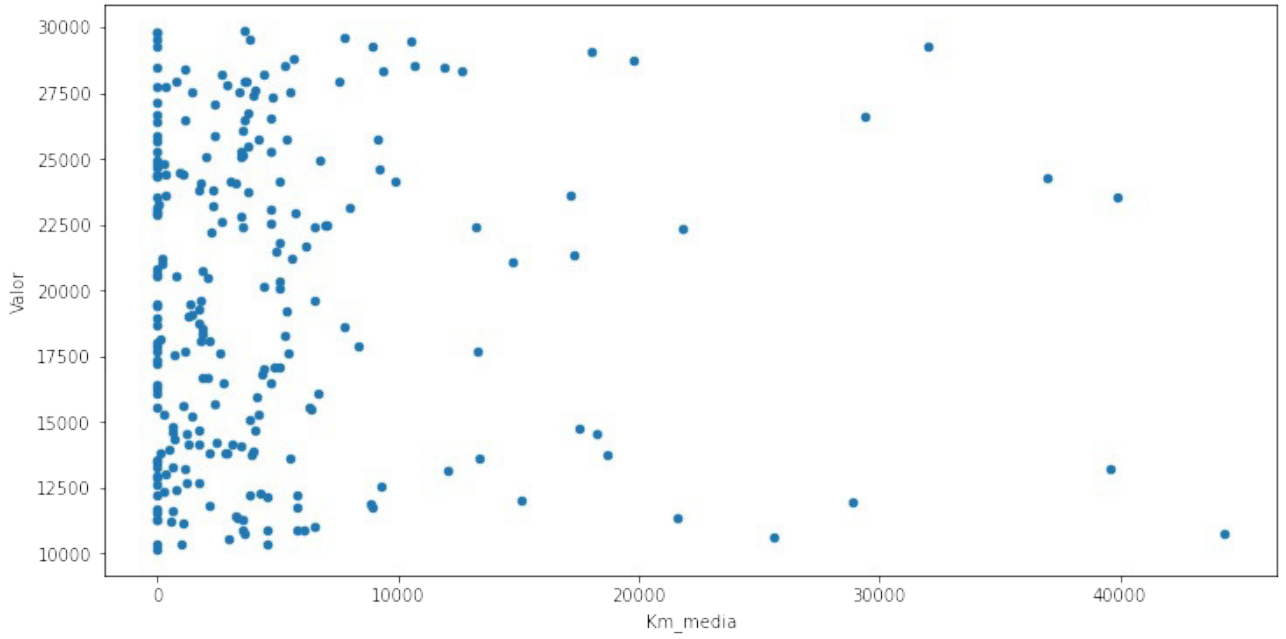
## Gráfico de dispersión

Un diagrama de dispersión es una representación gráfica de la relación existente entre dos variables numéricas. La evaluación de gráficos de dispersión es extremadamente importante para la aplicación de algunas técnicas estadísticas, como en análisis de regresión y correlaciones.

El método `scatter()` del atributo `plot` es una de las alternativas para construir gráficos de dispersión con Pandas. Las coordenadas de cada punto son definidas por dos columnas de un *DataFrame* y círculos llenos son utilizados para representar cada punto. En el ejemplo de abajo creamos un gráfico de dispersión entre las variables "Km_media" (eje X) y "Valor" (eje Y).

```
dataset.plot.scatter(x='Km_media', y='Valor', figsize = (12, 6))
```

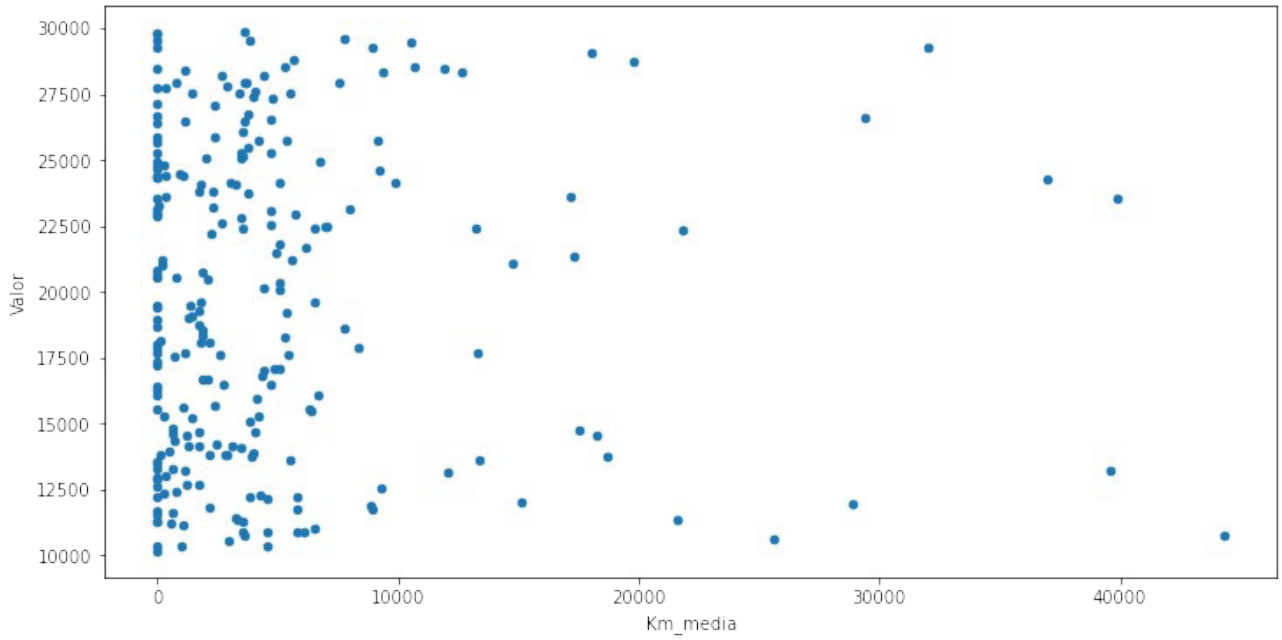
**Salida:**




Podemos obtener este mismo resultado utilizando `plot` como un método y pasando los mismos parámetros del ejemplo arriba y el parámetro adicional `kind` que determina el tipo de gráfico que queremos plotear.

```
dataset.plot(kind='scatter', x='Km_media', y='Valor', figsize = (12, 6))
```

**Salida:**




Las opciones posibles del parámetro `kind` son presentadas en la tabla abajo.

| kind | Tipo de gráfico |
|---|---|
| 'line' | Gráfico de líneas (*default*) |
| 'bar' | Gráfico de barras verticales |
| 'barh' | Gráfico de barras horizontales |
| 'hist' | Histograma |
| 'box' | Boxplot |
| 'kde' | Gráfico *Kernel Density Estimation* (Estimativa de Densidad por Kernel) |
| 'density' | El mismo que 'kde' |
| 'area' | Gráfico de áreas |
| 'pie' | Gráfico de sectores (pizza) |
| 'scatter' | Gráfico de dispersión |
| 'hexbin' | Gráfico bin hexagonal |

## Gráfico de sectores (pizza)

Los gráficos de sectores o, como son popularmente conocidos, gráficos de pizza son un recurso gráfico para representar las distribuciones de frecuencia relativa y las distribuciones de frecuencia porcentual de variables cualitativas sobre la forma de sectores de un círculo.

Para ejemplificar su uso vamos a crear una *Series* con las frecuencias de la variable "Cero*km" de nuestro _dataset*. Esa variable es del tipo booleana y para crear una distribución de frecuencias del

número de verdaderos y falsos basta utilizar el método `value_counts()`.

```
tabla = dataset['Cero_km'].value_counts(sort = False)
tabla
```

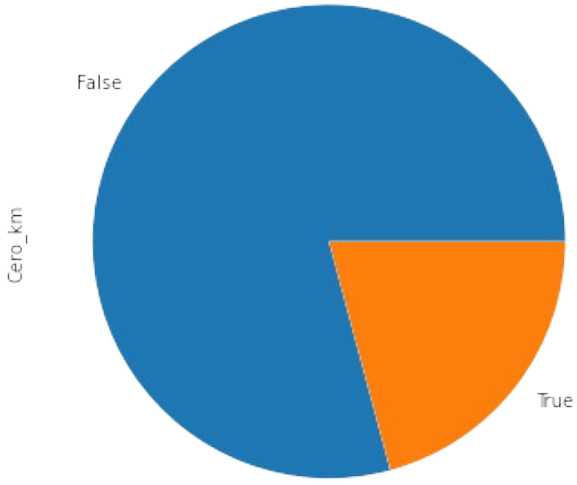
**Salida:**

```
False    194
True      51
Name: Cero_km, dtype: int64
```

Como la variable **`tabla`** es una *Series*, para crear un gráfico de sectores a partir de su contenido basta apenas ejecutar el código de abajo. Como se puede notar el único argumento pasado para el método `pie()` fue el `figsize` que es un parámetro de configuración para el tamaño de la figura de *output*. Si estuviéramos trabajando con un *DataFrame* tendríamos que informar el parámetro `y` con la columna que sería utilizada en la construcción del gráfico, veremos un ejemplo más adelante.

```
tabla.plot.pie(figsize = (12, 6))
```

**Salida:**

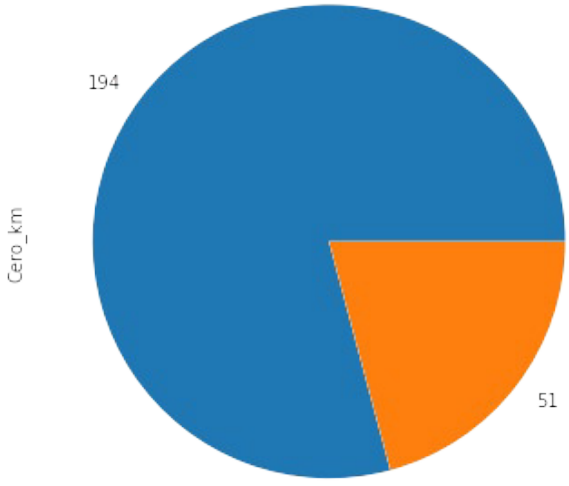



Observa que el gráfico de arriba es poco informativo, visto que apenas muestra los sectores del círculo y los *labels* (texto) de la variable. Eso nos indica apenas que tenemos más carros usados en nuestra base que carros nuevos, pero no nos informa la verdadera dimensión de esta diferencia. Para eso podemos proceder de dos formas.

La primera es cambiando los *labels* por los valores absolutos de las frecuencias. Para eso solo es necesario configurar el parámetro `labels` del método `pie()` con una lista conteniendo los valores o, en el caso de una *Series*, con la propia *Series*. Ver código de abajo.

```
tabla.plot.pie(labels=tabla, figsize = (12, 6))
```

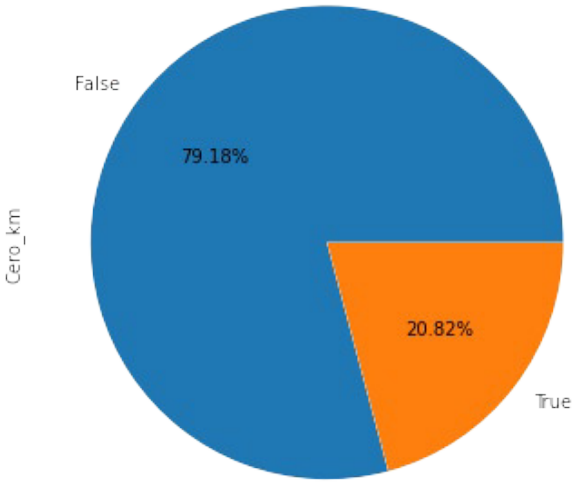
**Salida:**




La segunda forma es configurando el argumento `autopct` para mostrar los valores porcentuales de cada pedazo de la pizza. Puede ser dado como un especificador de formato para definir la forma como el valor será representado. En nuestro ejemplo configuramos con '%.2f%%' que puede ser dividido en dos parte para entender mejor ese tipo de notación. La primera es '%.2f' donde '%' es un carácter especial que informa que la string se trata de una *format string* y '.2f' que indica el tipo de dato y cómo será representado, donde el 'f' informa que se trata de un valor del tipo float y el '.2' indica que será representado con apenas dos posiciones decimales. Los dos últimos caracteres de la *string* ('%%') son para informar que el carácter '%' debe ser impreso en el final de la *string*.

```
tabla.plot.pie(autopct='%.2f%%', figsize = (12, 6))
```

**Salida:**



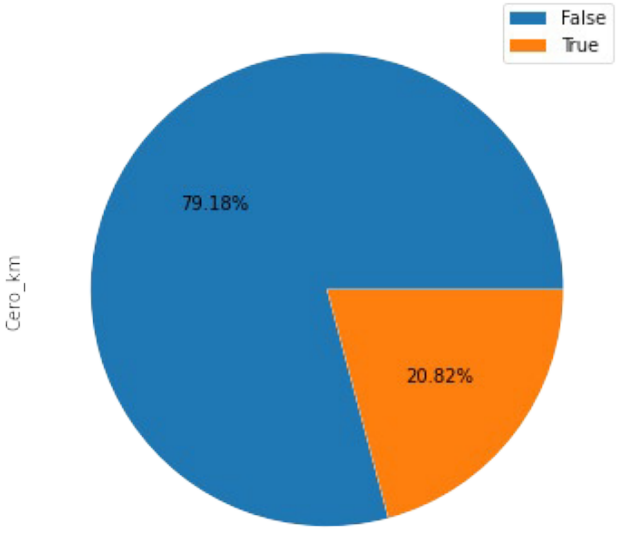

Como mencionado en el inicio de esta sección, cuando trabajamos con el método *pie()* en *DataFrames* necesitamos realizar una configuración extra. Para ejemplificar vamos a transformar nuestra variable **tabla** en un *DataFrame*.

```
tabla = tabla.to_frame()
```

Ahora para construir un gráfico de pizza necesitamos informar el parámetro y con el nombre de la columna que deseamos utilizar en el gráfico. Note que en este tipo de gráfico tenemos acceso al recurso de leyendas de los gráficos de sectores.

```
tabla.plot.pie(y='Cero_km', autopct='%.2f%%', labels=None, figsize = (12, 6))
```

**Salida:**



## Gráfico de barras

Los gráficos de barras representan las frecuencias sobre la forma de barras verticales o horizontales.

Considere la distribución de frecuencias de los tipos de motor de nuestro *DataFrame* de aula para ejemplificar la utilización de los gráficos de barras.

```
motores = dataset['Motor'].value_counts(sort = False)
motores
```

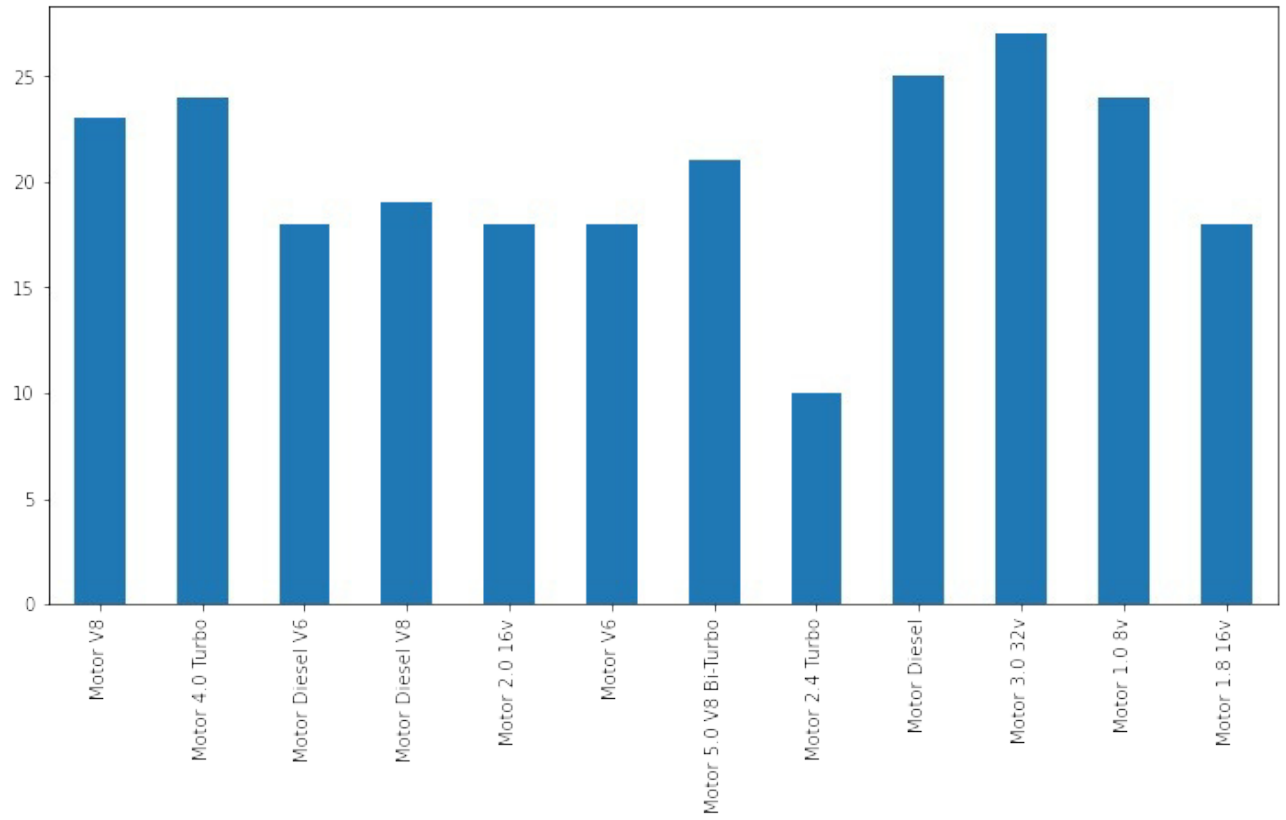
**Salida:**

```
Motor V8                 23
Motor 4.0 Turbo          24
Motor Diesel V6          18
Motor Diesel V8          19
Motor 2.0 16v            18
Motor V6                 18
Motor 5.0 V8 Bi-Turbo    21
Motor 2.4 Turbo          10
Motor Diesel             25
Motor 3.0 32v            27
Motor 1.0 8v             24
Motor 1.8 16v            18
Name: Motor, dtype: int64
```

Para crear un gráfico de barras verticales utilizamos el método `bar()` del atributo `plot`.

```
motores.plot.bar(figsize = (12, 6))
```

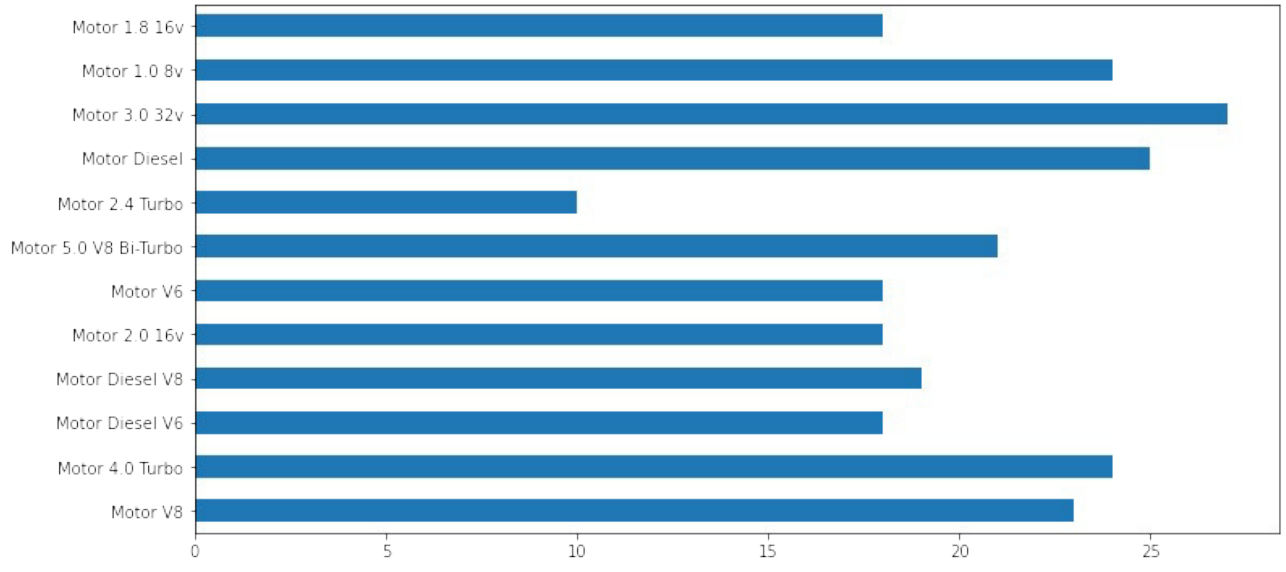
**Salida:**




Para un gráfico de barras horizontales utilizamos el método `barh()`.

```
motores.plot.barh(figsize = (12, 6))
```

**Salida:**



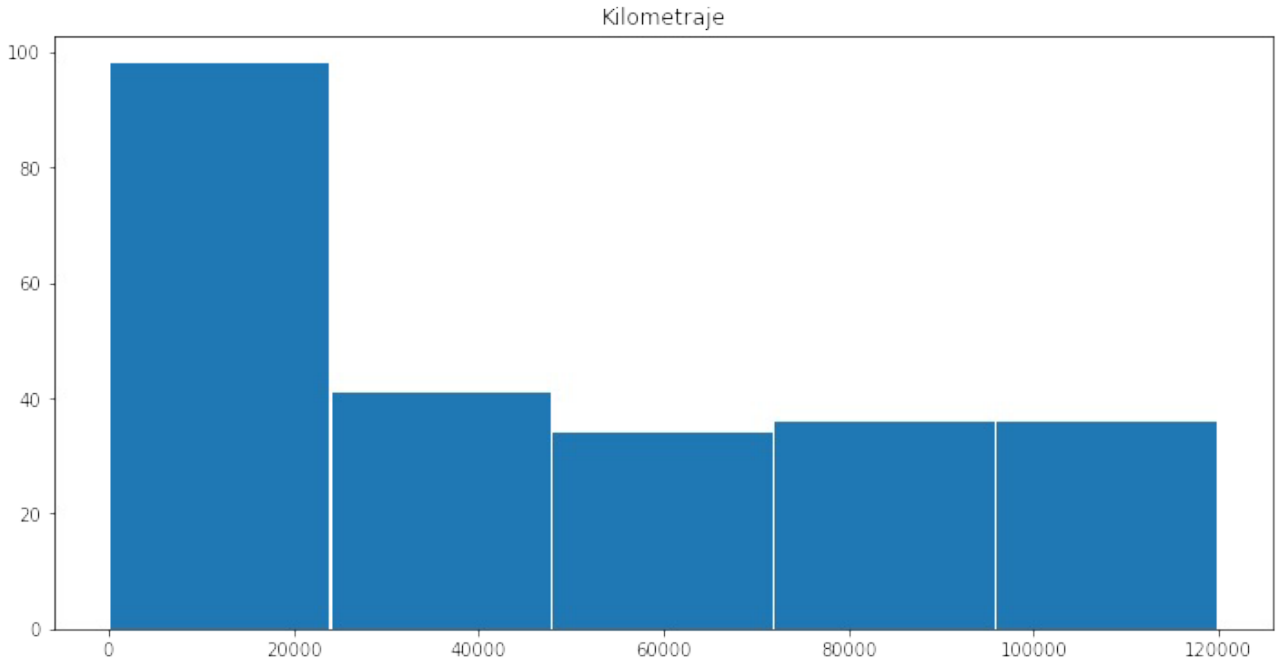

## Histograma

El histograma es una representación gráfica para datos cuantitativos bastante utilizada en estadística. Los histogramas son criados colocando la variable de interés en el eje x, y las frecuencias, simples o relativas, en el eje y. Las frecuencias de cada clase son representadas por rectángulos cuya base está determinada por los límites de la clase en el eje x y la altura por las frecuencias correspondientes.

Para construir histogramas con Pandas basta utilizar el método `hist()` de un *DataFrame*.

```
dataset.hist(column='Kilometraje', bins=5, grid=False, rwidth=0.99, figsize=(12, 6))
```

**Salida:**

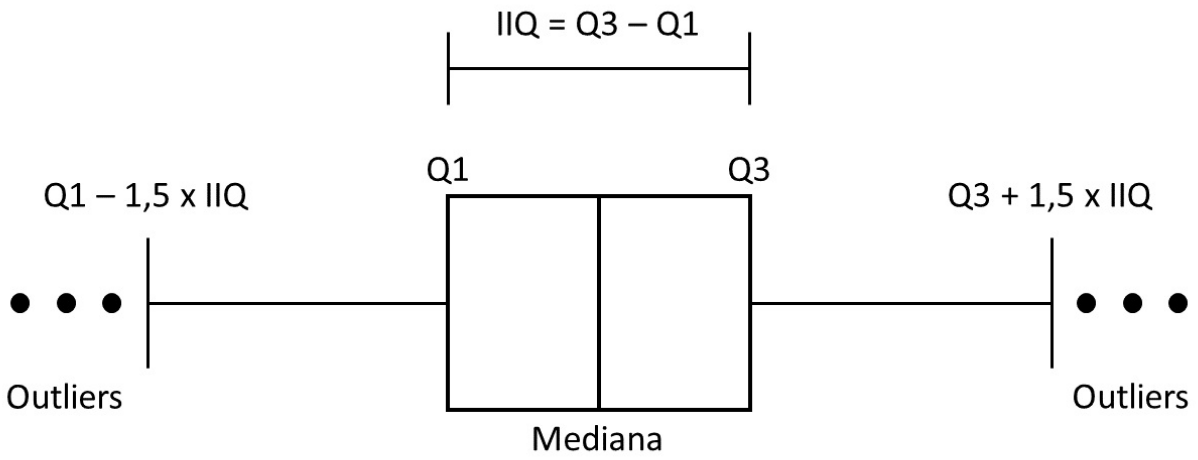



Los argumentos del método `hist()` utilizados fueron:

- `column` - Determina la columna del *DataFrame* que será utilizada;
- `bins` - Determina el número de classes (barras) que serán utilizadas;
- `grid` - Configuración para mostrar o no las líneas de grade del gráfico;
- `rwidth` - Configura la largura de las barras (utilizada en este ejemplo apenas como efecto visual).

## *Box plot*

El *box plot* es una herramienta gráfica bastante útil en estadística, a través de él es posible entender de forma rápida y sencilla como un conjunto de datos se distribuye. La figura de abajo muestra cómo leer un *box plot*.



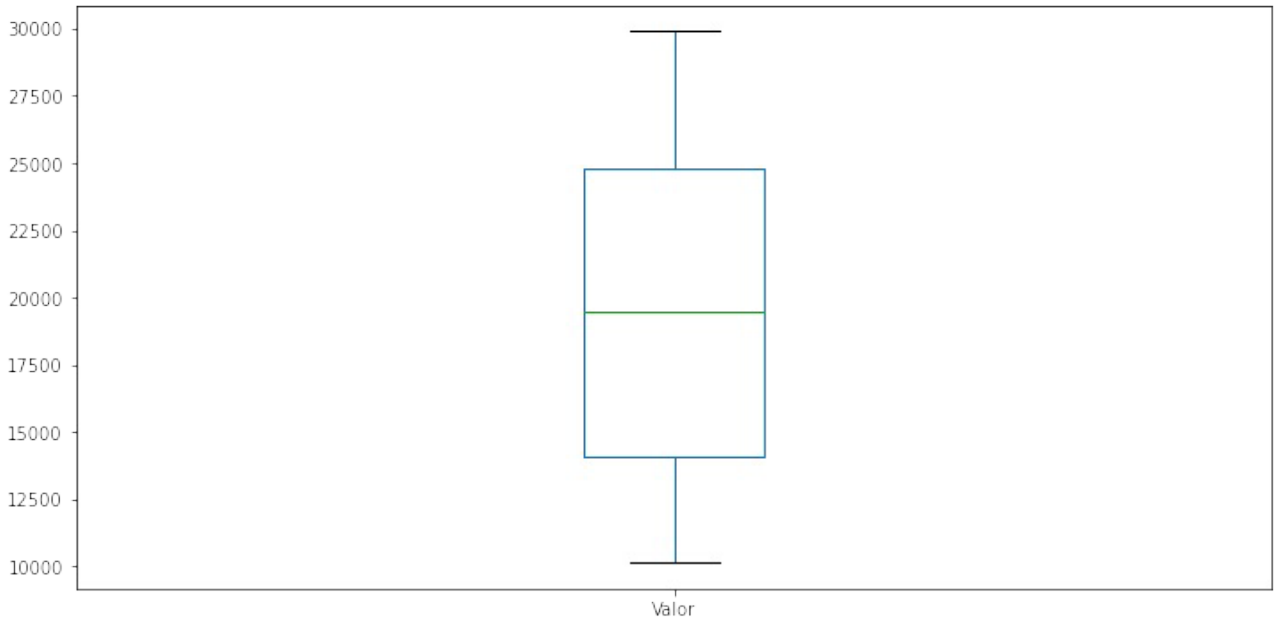

**Box-plot**

El *box plot* nos pasa informaciones sobre posición, dispersión, asimetría y datos discrepantes (*outliers*). La posición de la caja es definida por la mediana y la dispersión por $IIQ$ (intervalo intercuartílico). Las posiciones relativas de $Q1$ (1º cuartil), $Mediana$ y $Q3$ (3º cuartil) dan una noción de la simetría de la distribución. Las longitudes de las colas son datos que van del rectángulo a los valores remotos ($Q1 - 1,5 \times IIQ$ y $Q3 + 1,5 \times IIQ$) y por los valores atípicos.

Para crear *box plots* con Pandas basta acceder el método `boxplot()` del *DataFrame*. En el ejemplo de abajo creamos un *box plot* con la variable "Valor".

```
dataset.boxplot(column='Valor', grid=False, figsize = (12, 6))
```

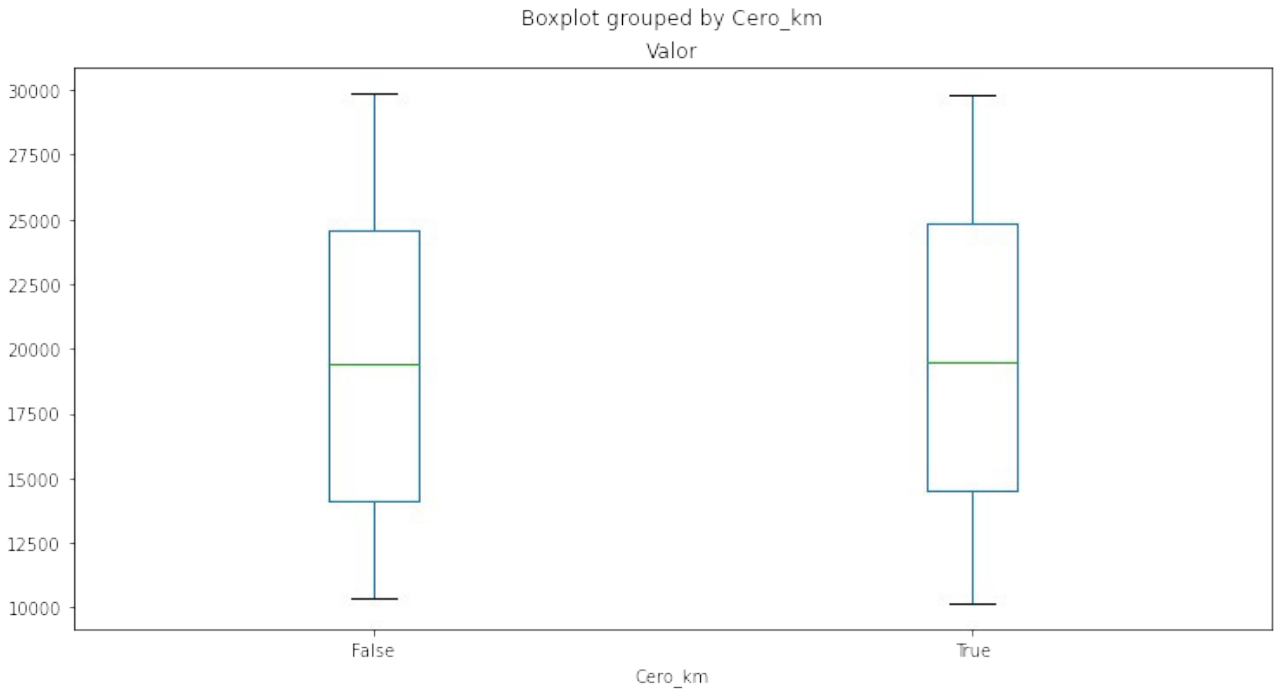
**Salida:**




También es posible crear un *box plot* para determinada variable según los valores de otra variable. Para eso utilizamos el argumento `by` e informamos la variable de separación.

```
dataset.boxplot(column='Valor', by='Cero_km', grid=False, figsize = (12, 6))
```

**Salida:**

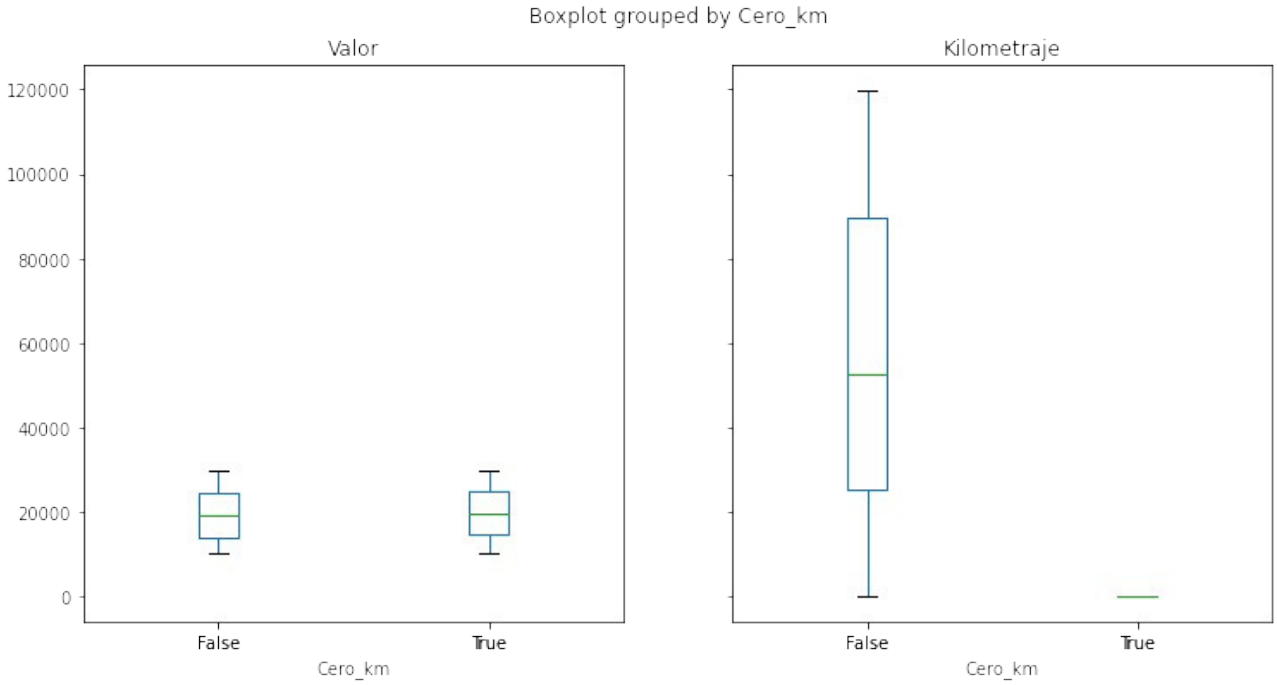



También podemos construir *box plots* con más de una variable en una misma figura, pasando una lista con los nombres de las columnas del *DataFrame* para el parámetro `column` . Eso también funciona

para el parámetro `by`.

```
dataset.boxplot(column=['Valor', 'Kilometraje'], by='Cero_km', grid=False, figsize = (12, 6))
```

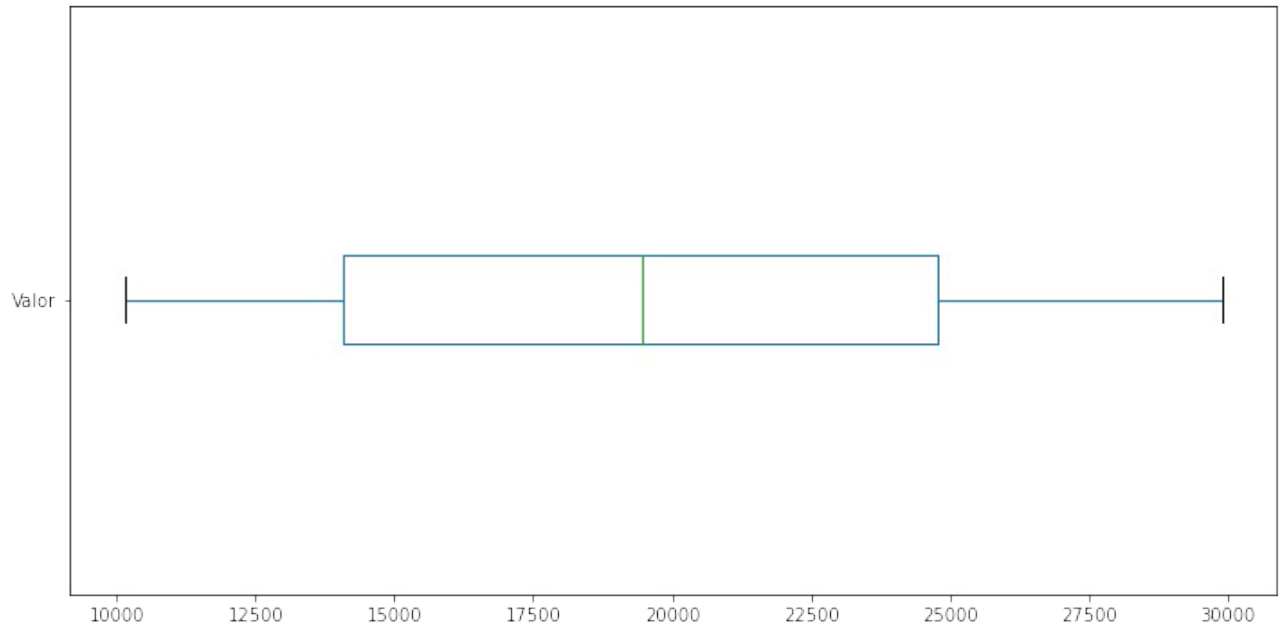
**Salida:**




Para generar un *box plot* como orientación horizontal, así como la primera figura de esta sección, configure el parámetro `vert` como *False*.

```
dataset.boxplot(column='Valor', grid=False, vert=False, figsize = (12, 6))
```

**Salida:**




Observe que los parámetros `grid` y `figsize` tienen la misma función de los método aprendidos anteriormente.

## 7.15 EJERCICIOS:

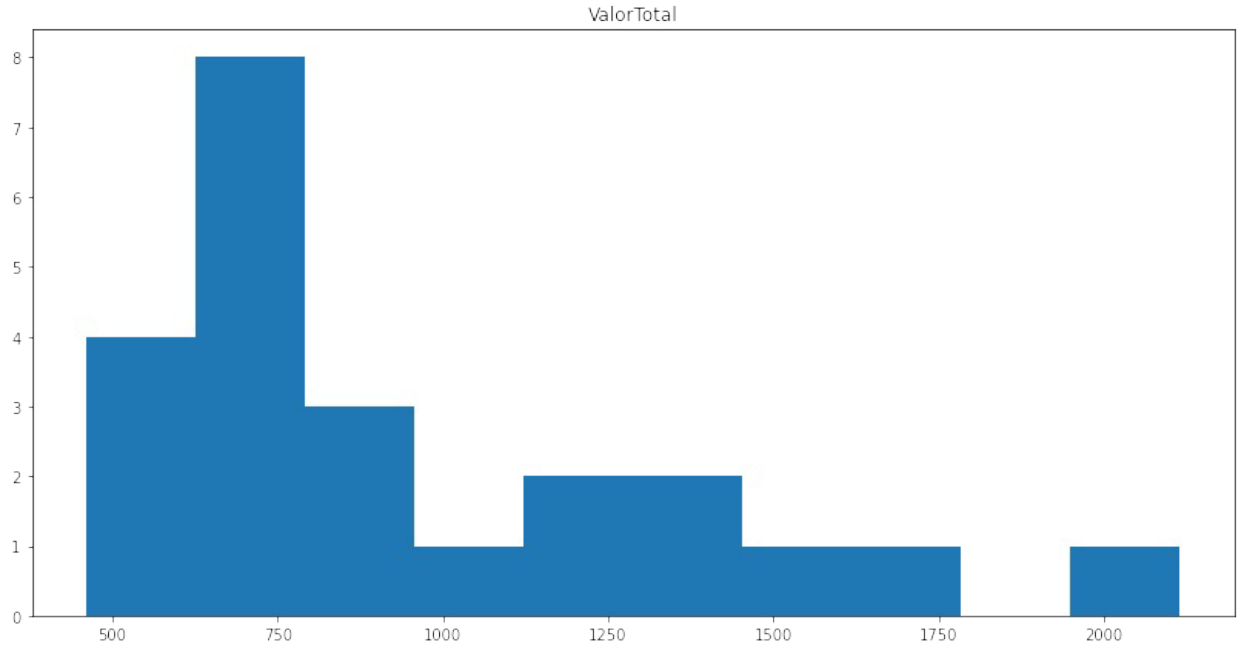
Evaluar gráficamente los datos puede nos ayudar en varias etapas de un proyecto de *Data Science*. En análisis de este tipo podemos identificar tendencias, tipos de relaciones entre variables, informaciones discrepantes (*outliers*), etc. Los gráficos también nos permiten presentar resultados de forma simple y resumida.

1. Algunas variables presentan características que necesitan ser identificadas para que el científico de datos sepa qué técnicas aplicar en su tratamiento. El histograma nos muestra la distribución de frecuencias de una variable en la forma gráfica. Eso nos facilita la identificación de asimetrías, posibles problemas con valores discrepantes, etc. Construye un histograma para la variable "ValorTotal".

   **Respuesta:**

   ```
   proyecto.hist(column='ValorTotal', grid=False, figsize=(14, 7))
   ```

   **Salida:**

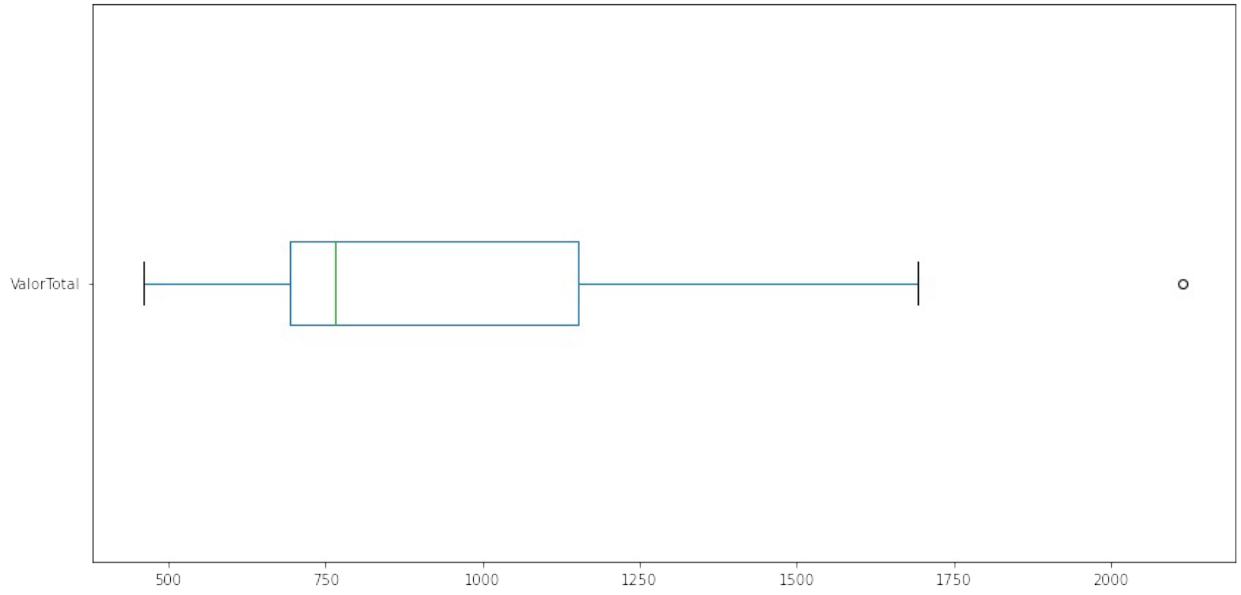



2. El gráfico de *box plot* nos da una idea de la posición, dispersión, asimetría, colas y datos discrepantes (*outliers*). La posición de la caja es dada por la mediana y la dispersión por la diferencia entre el 3º y el 1º cuartil que son los límites de la caja. Las posiciones relativas de $Q1$, $Mediana$ y $Q3$ dan una noción de la simetría de la distribución. Construye un *box plot* para la variable "ValorTotal".

   **Respuesta:**

```
proyecto.boxplot(column='ValorTotal', grid=False, vert=False, figsize = (14, 7))
```

   **Salida:**

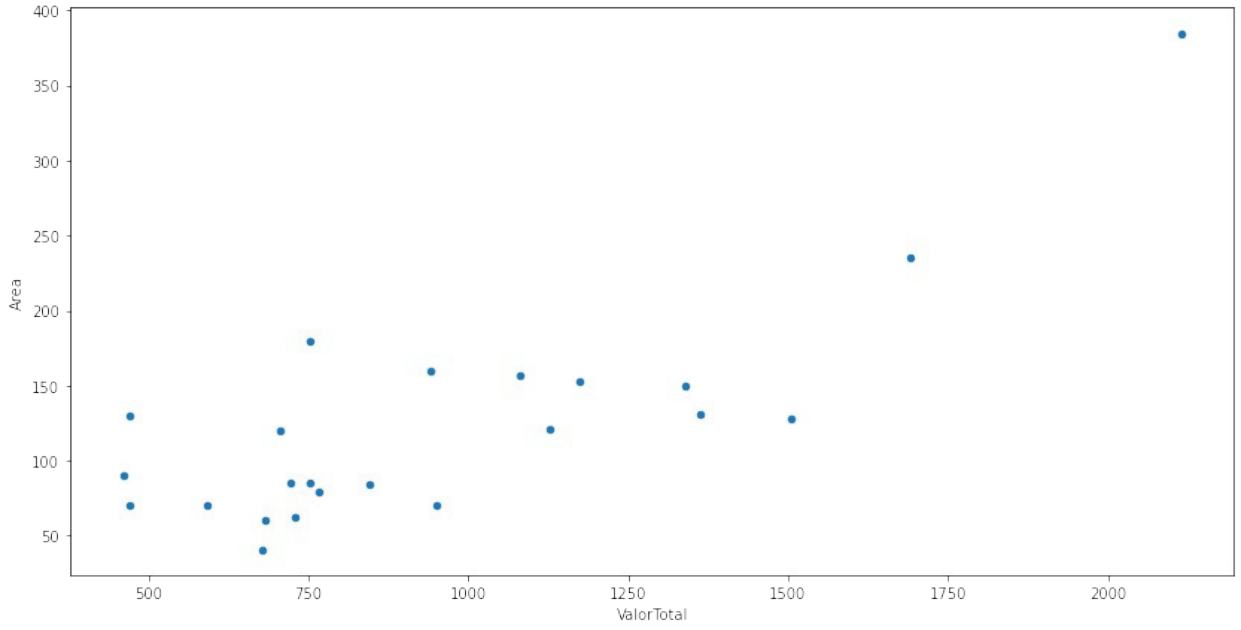


3. Suponga que necesitamos construir un modelo de regresión lineal entre las variables "ValorTotal" y "Area". Para eso necesitamos verificar si existe una relación lineal entre estas variables. El gráfico de dispersión nos ayuda en esta tarea. Construye un gráfico de dispersión para las variables "ValorTotal" (eje x) y "Area" (eje y).

   **Respuesta:**

```
proyecto.plot.scatter(x='ValorTotal', y='Area', figsize = (14, 7))
```

   **Salida:**

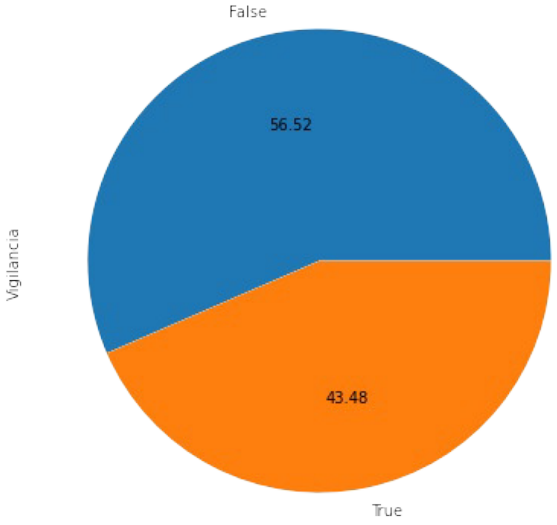



4. Un gráfico que auxilia en la presentación de resultados que envuelven variables cualitativas y sus proporciones es el gráfico de sectores, o como acostumbra ser llamado, gráfico de pizza. Construye dos gráficos de pizza, uno para la variable "Accesibilidad" y otro para la variable "Piscina". Observe que primero deben ser construidas las tablas de distribución de frecuencias para cada variable y a partir de estos resultados construir los gráficos.

   **Respuesta:**

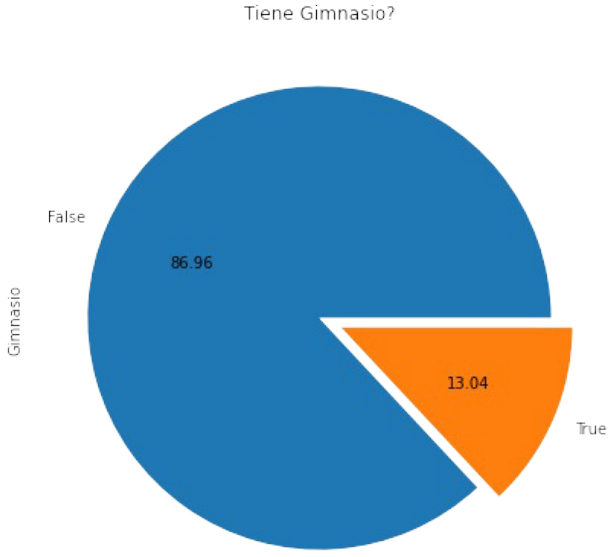
```
pizza_accesibilidad = proyecto['Vigilancia'].value_counts()
pizza_accesibilidad.plot.pie(autopct='%.2f', figsize = (14, 7))
```

   **Salida:**




```
pizza_piscina = proyecto['Gimnasio'].value_counts()
pizza_piscina.plot.pie(
    autopct='%.2f',
    figsize = (14, 7),
    explode=(0, 0.1),
    title='Tiene Gimnasio?'
)
```

**Salida:**



Capítulo 8

# ATTRIBUTION-NONCOMMERCIAL-NODERIVATIVES 4.0 INTERNATIONAL

=======================================================================

Creative Commons Corporation ("Creative Commons") is not a law firm and does not provide legal services or legal advice. Distribution of Creative Commons public licenses does not create a lawyer-client or other relationship. Creative Commons makes its licenses and related information available on an "as-is" basis. Creative Commons gives no warranties regarding its licenses, any material licensed under their terms and conditions, or any related information. Creative Commons disclaims all liability for damages resulting from their use to the fullest extent possible.

Using Creative Commons Public Licenses

Creative Commons public licenses provide a standard set of terms and conditions that creators and other rights holders may use to share original works of authorship and other material subject to copyright and certain other rights specified in the public license below. The following considerations are for informational purposes only, are not exhaustive, and do not form part of our licenses.

```
 Considerations for licensors: Our public licenses are
 intended for use by those authorized to give the public
 permission to use material in ways otherwise restricted by
 copyright and certain other rights. Our licenses are
 irrevocable. Licensors should read and understand the terms
 and conditions of the license they choose before applying it.
 Licensors should also secure all rights necessary before
 applying our licenses so that the public can reuse the
 material as expected. Licensors should clearly mark any
 material not subject to the license. This includes other CC-
 licensed material, or material used under an exception or
 limitation to copyright. More considerations for licensors:
wiki.creativecommons.org/Considerations_for_licensors

 Considerations for the public: By using one of our public
 licenses, a licensor grants the public permission to use the
 licensed material under specified terms and conditions. If
 the licensor's permission is not necessary for any reason--for
 example, because of any applicable exception or limitation to
 copyright--then that use is not regulated by the license. Our
 licenses grant only permissions under copyright and certain
 other rights that a licensor has authority to grant. Use of
 the licensed material may still be restricted for other
 reasons, including because others have copyright or other
 rights in the material. A licensor may make special requests,
```

```
 such as asking that all changes be marked or described.
 Although not required by our licenses, you are encouraged to
 respect those requests where reasonable. More_considerations
 for the public:
wiki.creativecommons.org/Considerations_for_licensees

    =======================================================================
```

Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 -- Definitions.

a. Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

c. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

d. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

e. Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

f. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

g. Licensor means the individual(s) or entity(ies) granting rights under this Public License.

h. NonCommercial means not primarily intended for or directed towards commercial advantage or monetary compensation. For purposes of this Public License, the exchange of the Licensed Material for other material subject to Copyright and Similar Rights by digital file-sharing or similar means is NonCommercial provided there is no payment of monetary compensation in connection with the exchange.

i. Share means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

Section 2 -- Scope.

a. License grant.

```
1. Subject to the terms and conditions of this Public License,
   the Licensor hereby grants You a worldwide, royalty-free,
   non-sublicensable, non-exclusive, irrevocable license to
   exercise the Licensed Rights in the Licensed Material to:

     a. reproduce and Share the Licensed Material, in whole or
        in part, for NonCommercial purposes only; and

     b. produce and reproduce, but not Share, Adapted Material
        for NonCommercial purposes only.

2. Exceptions and Limitations. For the avoidance of doubt, where
   Exceptions and Limitations apply to Your use, this Public
   License does not apply, and You do not need to comply with
   its terms and conditions.

3. Term. The term of this Public License is specified in Section
   6(a).

4. Media and formats; technical modifications allowed. The
   Licensor authorizes You to exercise the Licensed Rights in
   all media and formats whether now known or hereafter created,
   and to make technical modifications necessary to do so. The
   Licensor waives and/or agrees not to assert any right or
   authority to forbid You from making technical modifications
   necessary to exercise the Licensed Rights, including
   technical modifications necessary to circumvent Effective
   Technological Measures. For purposes of this Public License,
   simply making modifications authorized by this Section 2(a)
   (4) never produces Adapted Material.
```

5. Downstream recipients.

     a. Offer from the Licensor -- Licensed Material. Every
        recipient of the Licensed Material automatically
        receives an offer from the Licensor to exercise the
        Licensed Rights under the terms and conditions of this
        Public License.

     b. No downstream restrictions. You may not offer or impose
        any additional or different terms or conditions on, or
        apply any Effective Technological Measures to, the
        Licensed Material if doing so restricts exercise of the
        Licensed Rights by any recipient of the Licensed
        Material.

6. No endorsement. Nothing in this Public License constitutes or
   may be construed as permission to assert or imply that You
   are, or that Your use of the Licensed Material is, connected
   with, or sponsored, endorsed, or granted official status by,
   the Licensor or others designated to receive attribution as
   provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not
   licensed under this Public License, nor are publicity,
   privacy, and/or other similar personality rights; however, to
   the extent possible, the Licensor waives and/or agrees not to
   assert any such rights held by the Licensor to the limited
   extent necessary to allow You to exercise the Licensed
   Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this
   Public License.

3. To the extent possible, the Licensor waives any right to
   collect royalties from You for the exercise of the Licensed
   Rights, whether directly or through a collecting society
   under any voluntary or waivable statutory or compulsory
   licensing scheme. In all other cases the Licensor expressly
   reserves any right to collect such royalties, including when
   the Licensed Material is used other than for NonCommercial
   purposes.

Section 3 -- License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material, You must:

     a. retain the following if it is supplied by the Licensor
        with the Licensed Material:

          i. identification of the creator(s) of the Licensed
             Material and any others designated to receive
             attribution, in any reasonable manner requested by
             the Licensor (including by pseudonym if
             designated);

```
          ii. a copyright notice;

         iii. a notice that refers to this Public License;

          iv. a notice that refers to the disclaimer of
              warranties;

           v. a URI or hyperlink to the Licensed Material to the
              extent reasonably practicable;

      b. indicate if You modified the Licensed Material and
         retain an indication of any previous modifications; and

      c. indicate the Licensed Material is licensed under this
         Public License, and include the text of, or the URI or
         hyperlink to, this Public License.

   For the avoidance of doubt, You do not have permission under
   this Public License to Share Adapted Material.

2. You may satisfy the conditions in Section 3(a)(1) in any
   reasonable manner based on the medium, means, and context in
   which You Share the Licensed Material. For example, it may be
   reasonable to satisfy the conditions by providing a URI or
   hyperlink to a resource that includes the required
   information.

3. If requested by the Licensor, You must remove any of the
   information required by Section 3(a)(1)(A) to the extent
   reasonably practicable.
```

Section 4 -- Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database for NonCommercial purposes only and provided You do not Share Adapted Material;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 -- Disclaimer of Warranties and Limitation of Liability.

a. UNLESS OTHERWISE SEPARATELY UNDERTAKEN BY THE LICENSOR, TO THE

EXTENT POSSIBLE, THE LICENSOR OFFERS THE LICENSED MATERIAL AS-IS AND AS-AVAILABLE, AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE LICENSED MATERIAL, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHER. THIS INCLUDES, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR NOT KNOWN OR DISCOVERABLE. WHERE DISCLAIMERS OF WARRANTIES ARE NOT ALLOWED IN FULL OR IN PART, THIS DISCLAIMER MAY NOT APPLY TO YOU.

b. TO THE EXTENT POSSIBLE, IN NO EVENT WILL THE LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE) OR OTHERWISE FOR ANY DIRECT, SPECIAL, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, EXEMPLARY, OR OTHER LOSSES, COSTS, EXPENSES, OR DAMAGES ARISING OUT OF THIS PUBLIC LICENSE OR USE OF THE LICENSED MATERIAL, EVEN IF THE LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES, COSTS, EXPENSES, OR DAMAGES. WHERE A LIMITATION OF LIABILITY IS NOT ALLOWED IN FULL OR IN PART, THIS LIMITATION MAY NOT APPLY TO YOU.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 -- Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

```
1. automatically as of the date the violation is cured, provided
   it is cured within 30 days of Your discovery of the
   violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any
right the Licensor may have to seek remedies for Your violations
of this Public License.
```

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 -- Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 -- Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

==========

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at creativecommons.org/policies, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at creativecommons.org.