

Taller: Python (Pandas) para análisis de datos

Néstor Montaña

Sociedad Ecuatoriana de Estadística

Octubre 2023

**Nota:**

Con *Alt + F* o *Option + F* puede hacer que estas diapositivas ocupen todo el navegador (es decir que se ignore el aspecto de diapositiva que tiene por default la presentación)

Diapositivas y Set de Datos

https://github.com/nestormontano/2023_DS_Month__Taller_Pandas

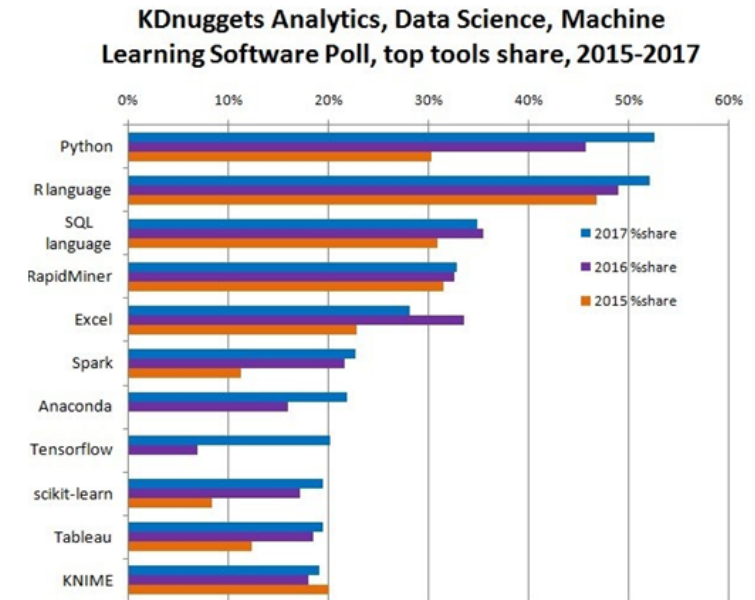
Introducción a Python

Taller: Python (Pandas) para análisis de datos

Néstor Montaña

¿Por qué Python?

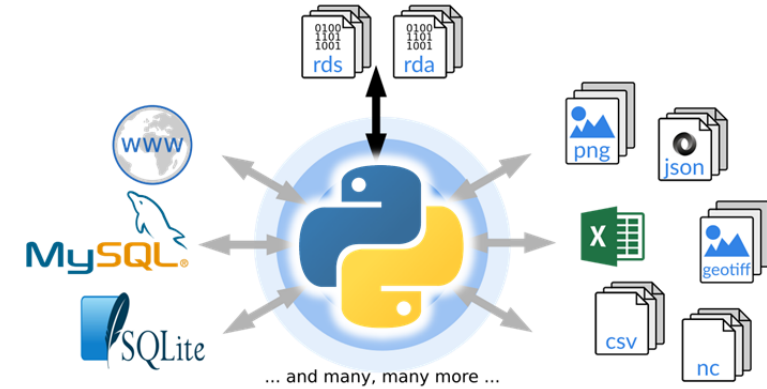
- Software Libre (Open Source), gratuito y de desarrollo independiente,
- Es un lenguaje de objetivo general, es decir que sirve para hacer sitios web, aplicaciones móviles, sistemas de escritorio y para hacer ciencia de datos
- Hoy es el lenguaje más usado para Ciencia de datos y uno de los más usados en general,
- Enorme comunidad de usuarios,
- La mayoría de Universidades enseñan Python para carreras de computación o sistemas.



Popularidad Python

¿Por qué Python?

- Rico ecosistema de librerías, integraciones, frameworks, etc,
- Las integraciones de un modelo a producción suelen ser sencillo con Python,
- Hay distribuciones de Python enfocadas a Ciencia de Datos como Anaconda,
- Para programar en Python se pueden usar algunas IDEs (Interfaz de desarrollo) como **Jupyter**, **PyCharm**, **Spyder**, **Rstudio**, algunas de ellas integradas con Anaconda,
- Los Frameworks más usados para DeepLearning usan Python como base.



Algunas de las integraciones de Python



Instalar Python

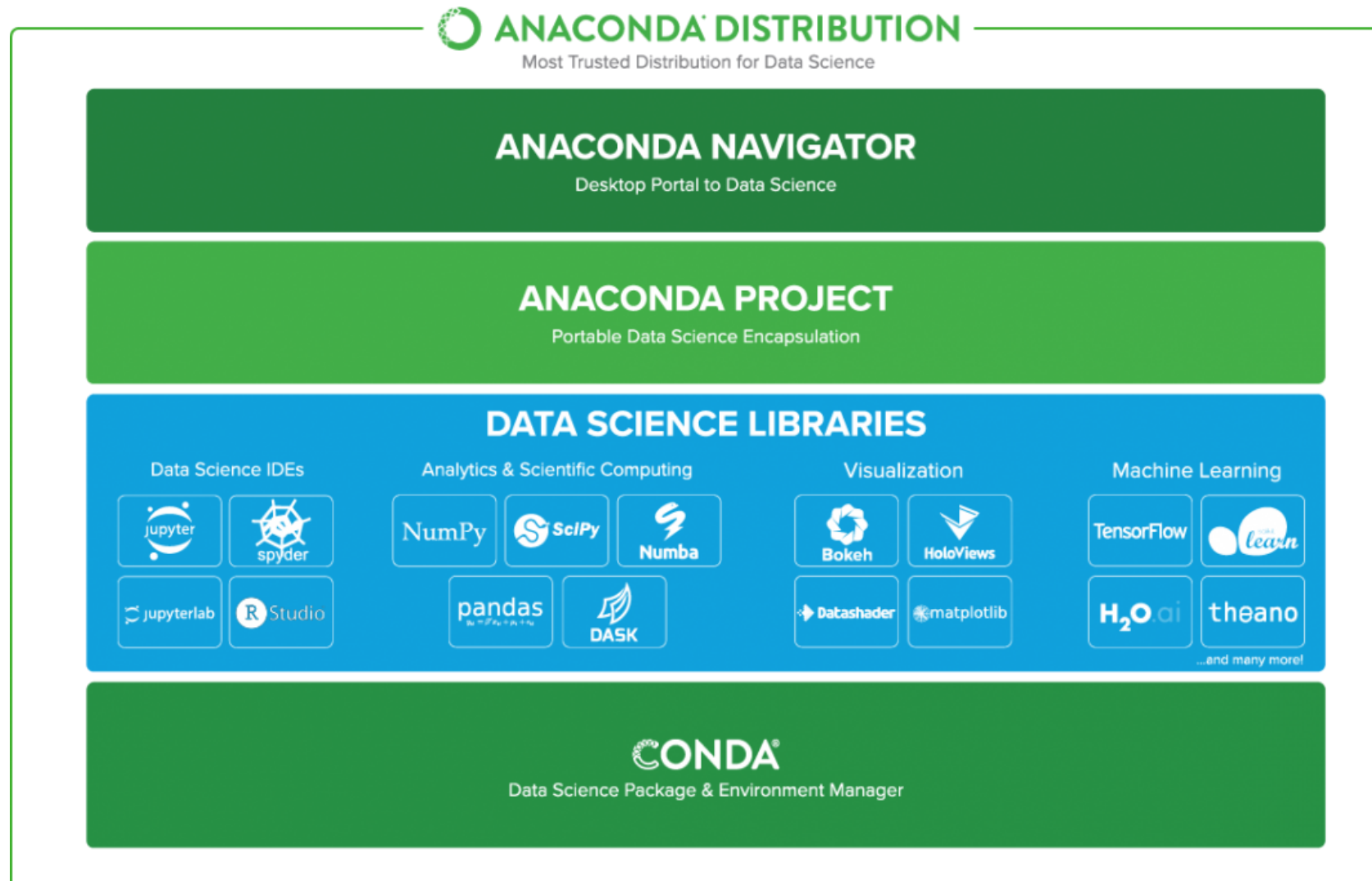
En windows y Mac

- Visitar sitio web de Python
- Elegir la versión que se desea instalar
- Descargar y ejecutar el instalador marcando la casilla "Añadir Python `##` al PATH"
- Ojo: Mac viene con python 2.7, pero se aconseja actualizar a nuevas versiones

En Linux (Distribuciones)

- Python también viene instalado en linux
- Se lo instala usando la consola, ejemplo:
 - `sudo yum install python` (en Fedora, Red Hat o derivadas)
 - `sudo apt-get install Python` (en Debian, Ubuntu y derivadas)

Distribuciones: Anaconda



Python desde Google Colab

Google permite usar Python desde la nube con su herramienta **Google Colab**, la cual no es más que un Jupyter modificado por Google, con la ventaja de que no usaremos los recursos de nuestra computadora sino los que Google nos "preste"

Link a [Google Colab](#),



Google Colab



Generalidades

Python, aparte de objetos, tiene:

- Expresión.- Se evalúa, se imprime y el valor se pierde (iPython)

```
5+5 # Expresión con output
```

```
## 10
```

```
5+5; # Expresión sin salida
```

```
## 10
```

- Asignación.- Evalúa la expresión y guarda el resultado en una variable (no lo imprime)

```
a = 5+5 # Asigna el valor a la variable "a"
```

Asignaciones

- El resultado de una función de un objeto X puede ser asignada al mismo objeto X en la misma sentencia, es decir

```
a = 5 # Asignación  
a
```

```
## 5
```

```
a = 2*a # Asignación a mismo objeto  
a
```

```
## 10
```



Generalidades

- Comandos se separan por ; o enter
- Para comentar se usa #
- Case sensitivity (Abc es diferente de abc)

```
a= 2; b= 1; a + b
```

```
## 3
```

Python como calculadora

```
2 + 3*5 # operaciones básicas
```

```
## 17
```

```
7 // 3 # division entera
```

```
## 2
```

```
7 % 3 # Modular
```

```
## 1
```

```
2 ** 3 # 2 elevado al cubo
```

```
## 8
```

```
pow(2, 3) # 2 elevado al cubo
```

```
## 8
```



Python como calculadora

Para usar operaciones más "complicadas" debemos ya importar una biblioteca/librería, la cual es una colección de módulos, funciones y objetos que aumentan las capacidades del lenguaje, en este caso `math` permite cargar funciones enfocadas en cálculos matemáticos básicos.

Una biblioteca se instala (que es descargar los archivos ordenados a nuestro disco duro) y luego se activa (que es cargarla a RAM para poder usar sus funciones módulos), esto último se hace con `import`

```
import math
math.floor(2.3) # Funcion piso parte del paquete math
```

```
## 2
```

```
math.fabs(-5) # valor absoluto parte del paquete math
```

```
## 5.0
```

```
math.factorial(3) # Factorial parte del paquete math
```

```
## 6
```



Manejo de paquetes

Como se dijo:

- Una biblioteca/librería es una colección de funciones y objetos que aumentan las capacidades del lenguaje,
- Es simplemente un directorio que contiene otros paquetes, módulos o scripts,
- Instalación: `conda install`, `pip install`, `pip3 install`.



Bibliotecas a usar

Los paquetes o bibliotecas más usadas para análisis de datos son:

```
import os                # Funciones relacionadas con el sistema operativo
import math              # Operaciones matemáticas avanzadas
import numpy as np       # Operaciones vectoriales (np.array) eficientes
import pandas as pd     # Manejo de datos tabulares (dataframe) y descriptiva
import scipy             # Herramientas y algoritmos matemáticos
import matplotlib.pyplot as plt # Visualizaciones
import seaborn as sns   # Visualizaciones mas sencillas
```

Existen otras para modelamiento como scikit learn, keras, etc.



Pandas

Pandas es la gran navaja suiza de Python para Data Science

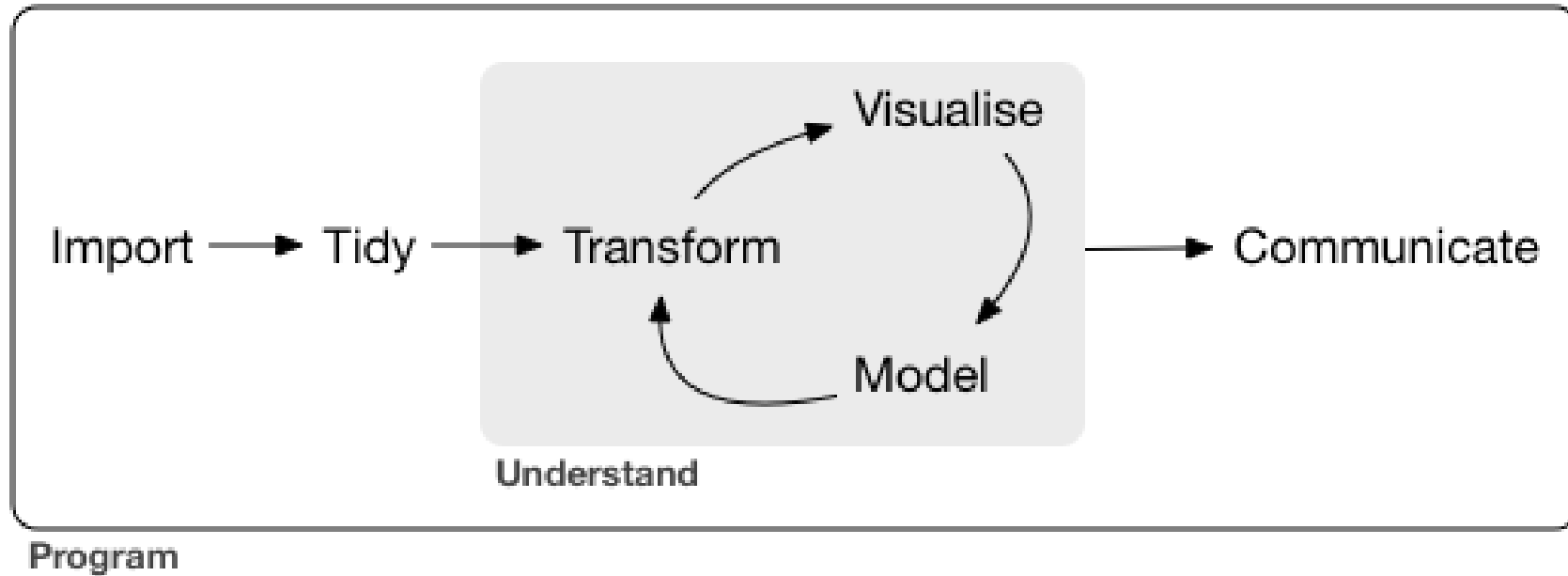
- Soporta lectura desde una variedad de datos, integrarlos y transformarlos
- Permite realizar estadística descriptiva
- Tiene opciones de gráficos integrados
- Soporta series de tiempo
- Métodos integrados para manejar valores perdidos
- Soporte para procesamiento de imágenes
- Internamente maneja dos tipos de objetos, pandas Series, panda DataFrame

Desarrollo de un caso de análisis de datos

Taller: Python (Pandas) para análisis de datos

Néstor Montaña

Workflow de un análisis de datos



- Import: Obtener y entender los datos
- Tidy: Ordenar los datos de tal manera que sea sencillo transformarlos, resumizarlos, visualizarlos o realizar un modelo con ellos
- Transform: Manipular los datos hasta obtener el input que el análisis o técnica estadística necesita
- Visualise: Realizar el análisis exploratorio de datos
- Model: Aplicar técnicas estadísticas para el entendimiento del problema o tomar decisiones
- Communicate: Tratar de mostrar los resultados de tal forma que el resto del mundo los entienda, usando reportes, gráficos, visualizaciones interactivas, integración con herramientas de BI, web apps, etc.



Ejemplo: Data de transacciones bancarias

El Banco del Pacífico requiere mejorar los tiempos de atención al cliente en ventanilla, para ello ha recolectado esta información anónimamente para cada cajero y transacción realizada.

Le suministran un excel con dos hojas:

1. Tiene los datos de las transacciones, columnas: Sucursal, Cajero, ID_Transaccion, Transaccion, Tiempo_Servicio_seg, Nivel de satisfacción, Monto de la transaccion.
2. Otra hoja que indica si en la sucursal se ha puesto o no el nuevo sistema.
3. Datos demográficos de los cajeros



Importar desde excel

Importar desde excel, opción 1

```
# Debe estar seteado el chdir
archivo_xlsx = 'Data//Data_Banco.xlsx' # Ruta al archivo
xlsx = pd.ExcelFile(archivo_xlsx) # Carga todo el spreadsheet
print(xlsx.sheet_names) # Ver hojas del archivo
```

```
## ['Data', 'Data_Sucursal', 'Data_Cajero']
```

```
data_banco = xlsx.parse('Data')
data_banco.head(5)
```

```
##      Sucursal  Cajero  ID_Transaccion  ...  Tiempo_Servicio_seg  Satisfaccion  Monto
## 0           62    4820                2  ...              311.0      Muy Bueno  2889,3
## 1           62    4820                2  ...              156.0           Malo  1670,69
## 2           62    4820                2  ...              248.0      Regular  3172,49
## 3           62    4820                2  ...               99.0      Regular  1764.92
## 4           62    4820                2  ...             123.0      Muy Bueno  1835.69
##
## [5 rows x 7 columns]
```



Importar desde excel

Importar desde excel, opción 2

```
# OPCION 2
data_banco_xlsx = pd.read_excel(archivo_xlsx, sheet_name = 'Data')
data_banco_xlsx.head(5)
```

```
##      Sucursal  Cajero  ID_Transaccion  ...  Tiempo_Servicio_seg  Satisfaccion  Monto
## 0           62    4820           2  ...           311.0      Muy Bueno  2889,3
## 1           62    4820           2  ...           156.0           Malo  1670,69
## 2           62    4820           2  ...           248.0      Regular  3172,49
## 3           62    4820           2  ...            99.0      Regular  1764.92
## 4           62    4820           2  ...           123.0      Muy Bueno  1835.69
##
## [5 rows x 7 columns]
```



Importar desde excel

Importar la otra hoja de excel

```
data_sucursal = pd.read_excel('Data//Data_Banco.xlsx', sheet_name = 'Data_Sucursal')
data_sucursal.head()
```

	ID_Sucursal	Sucursal	Nuevo_Sistema
## 0	62	Riocentro Sur	No
## 1	85	Centro	Si
## 2	267	Alborada	Si
## 3	443	Mall del Sol	Si
## 4	586	Via Daule	No



Importar desde otras fuentes

Panda permite importar desde una gran cantidad de fuentes, incluso conectarse a Bases de Datos.

- `read_csv` para importar desde csv
- `ExcelFile` & `xl.parse` o `read_excel` para importar desde excel
- `openpyxl` también permite manipular excels
- `read_json` para importar desde json
- `read_sql_table` para importar toda una tabla
- Más información en: <https://pandas.pydata.org/docs/reference/io.html>

Ejemplo - Importar

Bien, se han creado dos objetos, **¿qué tipo de estructura hemos importado?**

R. Un `pandas.core.frame.DataFrame`

- Se puede ver la estructura de un objeto con `type()` o `.__class__`

```
type(data_banco)
```

```
## <class 'pandas.core.frame.DataFrame'>
```

```
data_banco.__class__
```

```
## <class 'pandas.core.frame.DataFrame'>
```



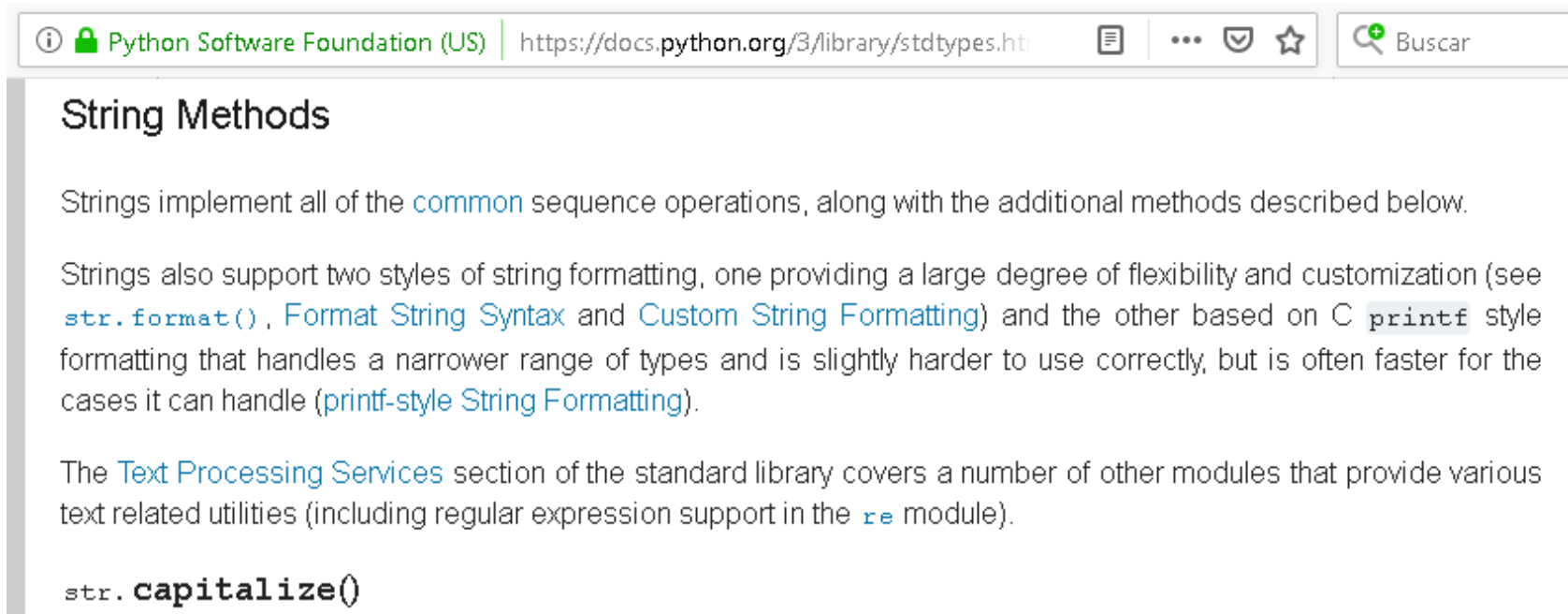
Métodos en Python

Notar que la forma de ejecutar el `type(data_banco)` y el `data_banco.__class__` es un poco diferente, en el primero `data_banco` es un argumento y en el segundo se parte del `data_banco`, esto es porque `type()` es una función y `.__class__` es un método.

En python para aplicar un método se usa `nombre_del_objeto.metodo(parametros)`

Por ejemplo, podemos ver los métodos asociados a un objeto de tipo string en

<https://docs.python.org/3/library/stdtypes.html#string-methods>



The screenshot shows a web browser window displaying the Python documentation for String Methods. The address bar shows the URL `https://docs.python.org/3/library/stdtypes.html`. The page title is "String Methods". The text explains that strings implement all of the common sequence operations, along with the additional methods described below. It also mentions that strings support two styles of string formatting: one providing a large degree of flexibility and customization (see `str.format()`, Format String Syntax and Custom String Formatting) and the other based on C printf style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle (printf-style String Formatting). The text also mentions that the Text Processing Services section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

```
str.capitalize()
```



pandas.core.frame.DataFrame ¿Qué es eso?

pandas Series, pandas DataFrame son **Estructuras de datos**, las cuales no son más que tipos de Objetos dentro de Python. Un entero es un objeto, un número también es un objeto. Ambos se pueden "ordenar" en estructuras como:

- Listas
- Matrices
- Tuples
- numpy arrays
- pandas Series
- pandas DataFrames
- Ver <https://docs.python.org/3/tutorial/datastructures.html>

Se verá rápidamente algunas de ellas



Estructuras de datos | Objetos

Un entero

```
a= 5  
a
```

```
## 5
```

```
type( a)
```

```
## <class 'int'>
```

Estructuras de datos | Objetos

Una numpy.array

```
a= np.array([1, 2, 3])  
a
```

```
## array([1, 2, 3])
```

```
type( a)
```

```
## <class 'numpy.ndarray'>
```



Estructuras de datos | Objetos

Una lista

```
a= [2, 4]  
a
```

```
## [2, 4]
```

```
type(a)
```

```
## <class 'list'>
```



Estructuras de datos | Objetos

Un pandasSeries

```
a = pd.Series([2, 4])  
a
```

```
## 0    2  
## 1    4  
## dtype: int64
```

```
type(a)
```

```
## <class 'pandas.core.series.Series'>
```

```
a.values
```

```
## array([2, 4], dtype=int64)
```



pandas.DataFrame

- DataFrame es un objeto que cumple:
 - Las columnas son vectores de tipo pandas Series
 - Cada columnas puede ser de un tipo de dato distinto
 - Cada elemento, columna es una variable
 - Las columnas tienen el mismo largo
- Se podría decir que un data.frame es como una tabla en una hoja de excel

pandas.DataFrame

Crear un data.frame

```
df_2= pd.DataFrame({  
    'Nombre' : ['Ana', 'Berni', 'Carlos'],  
    'Edad'   : [20,19,20],  
    'Ciudad' : ['Gye', 'Uio', 'Cue']  
})
```

df_2

	Nombre	Edad	Ciudad
0	Ana	20	Gye
1	Berni	19	Uio
2	Carlos	20	Cue

pandas.DataFrame

Index en un dataframe

```
df_3= pd.DataFrame({  
    'Nombre' : ['Ana', 'Berni', 'Carlos'],  
    'Edad'   : [20,19,20],  
    'Ciudad' : ['Gye', 'Uio', 'Cue']  
    }, index= ['a', 'b', 'c'])
```

df_3

	Nombre	Edad	Ciudad
a	Ana	20	Gye
b	Berni	19	Uio
c	Carlos	20	Cue

pandas.DataFrame

Visualizar primeras filas

```
df_3.head(2)
```

	Nombre	Edad	Ciudad
a	Ana	20	Gye
b	Berni	19	Uio



pandas.DataFrame

Visualizar últimas filas

```
df_3.tail(2)
```

	Nombre	Edad	Ciudad
b	Berni	19	Uio
c	Carlos	20	Cue



pandas.DataFrame

.info() permite ver la estructura de cualquier objeto en python.

```
## Visualizar la estructura  
df_3.info()
```

```
## <class 'pandas.core.frame.DataFrame'>  
## Index: 3 entries, a to c  
## Data columns (total 3 columns):  
##  #   Column  Non-Null Count  Dtype  
## ---  ---  
##  0   Nombre  3 non-null      object  
##  1   Edad    3 non-null      int64  
##  2   Ciudad  3 non-null      object  
## dtypes: int64(1), object(2)  
## memory usage: 96.0+ bytes
```

pandas.DataFrame

Modificar nombre de las variables

```
df_3.rename( columns= {'Nombre':'Name', 'Edad':'Age', 'Ciudad':'City'}) # No cambia el objeto
```

```
##      Name  Age  City
## a     Ana   20   Gye
## b    Berni   19   Uio
## c   Carlos   20   Cue
```

```
df_3
```

```
##      Nombre  Edad  Ciudad
## a     Ana    20     Gye
## b    Berni    19     Uio
## c   Carlos    20     Cue
```



pandas.DataFrame

Modificar nombre de las variables

```
df_3.rename( columns= {'Nombre':'Name', 'Edad':'Age', 'Ciudad':'City'},  
inplace= True) # Cambia el objeto  
df_3
```

```
##      Name  Age  City  
## a     Ana   20   Gye  
## b    Berni  19   Uio  
## c   Carlos  20   Cue
```



Entender los datos

Luego de importar se debe entender los datos

- ¿Qué representa cada columna?
- ¿Qué tipo de dato debería tener cada columna?
- ¿Qué granularidad o atomicidad tiene la data?
- Si es que se tiene varios conjuntos de datos ¿Cómo se relacionan los datos?
- A qué periodo de tiempo corresponde la data
- Muchas veces se obtiene la información desde una base de datos y por tanto toca entender la base y el query que genera los datos

Ejemplo - Entender los datos

Podríamos ver las primeras filas

```
# ver las primeras 5 filas  
data_sucursal.head(5)
```

	ID_Sucursal	Sucursal	Nuevo_Sistema
## 0	62	Riocentro Sur	No
## 1	85	Centro	Si
## 2	267	Alborada	Si
## 3	443	Mall del Sol	Si
## 4	586	Via Daule	No



Ejemplo - Entender los datos

O ver la estructura de los dataFrames

```
data_banco_xlsx.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 24299 entries, 0 to 24298
## Data columns (total 7 columns):
##  #   Column                Non-Null Count  Dtype
## ---  -
##  0   Sucursal              24299 non-null  int64
##  1   Cajero                 24299 non-null  int64
##  2   ID_Transaccion        24299 non-null  int64
##  3   Transaccion           24299 non-null  object
##  4   Tiempo_Servicio_seg   24299 non-null  float64
##  5   Satisfaccion          24299 non-null  object
##  6   Monto                 24299 non-null  object
## dtypes: float64(1), int64(3), object(3)
## memory usage: 1.3+ MB
```

Tipos de datos

Para saber qué tipo de dato debería tener cada columna, debemos también conocer los tipos de datos en Python

```
a= 1  
type(a)
```

```
## <class 'int'>
```

```
a= 1.3  
type(a)
```

```
## <class 'float'>
```



Tipos de datos

Tipos datos en Python

```
a= 1 + 2j  
type(a)
```

```
## <class 'complex'>
```

```
a= 'texto'  
type(a)
```

```
## <class 'str'>
```



Tipos de datos

```
from datetime import date  
a= date.fromisoformat('2019-12-04')  
type(a)
```

```
## <class 'datetime.date'>
```

```
b= date(2019, 12, 4)  
type(b)
```

```
## <class 'datetime.date'>
```



Tipos de datos - pd.Categorical

Util para tipos de datos ordinales

- Primero se agrega el vector de información
- Categories: los niveles del factor labels: nombre de los niveles
- El factor puede tener un orden específico

```
# Crear un factor ordenado
a = pd.Categorical( ['alto', 'bajo', 'alto', 'alto'],
                   categories= ['bajo', 'mediano', 'alto'],
                   ordered=True)
```

```
a # Mostrar el factor
```

```
## ['alto', 'bajo', 'alto', 'alto']
## Categories (3, object): ['bajo' < 'mediano' < 'alto']
```

```
type(a)
```

```
## <class 'pandas.core.arrays.categorical.Categorical'>
```



Tipos de datos

Datos lógicos

```
b= True  
type(b)
```

```
## <class 'bool'>
```

```
b
```

```
## True
```



Tipos de datos

Casos especiales

```
1.797e308 # maximo float posible
```

```
## 1.797e+308
```

```
1.798e308 # Resulta en infinito
```

```
## inf
```

```
pos_inf = math.inf      # Constante desde la libreria math  
neg_inf = float('-inf')  # Declarar un float Inf, tb permite nan
```




Tipos de datos

Casos especiales: Not a Number

```
c= math.nan  
type(c)
```

```
## <class 'float'>
```

```
d= float('nan')  
type(d)
```

```
## <class 'float'>
```



Tipos de datos

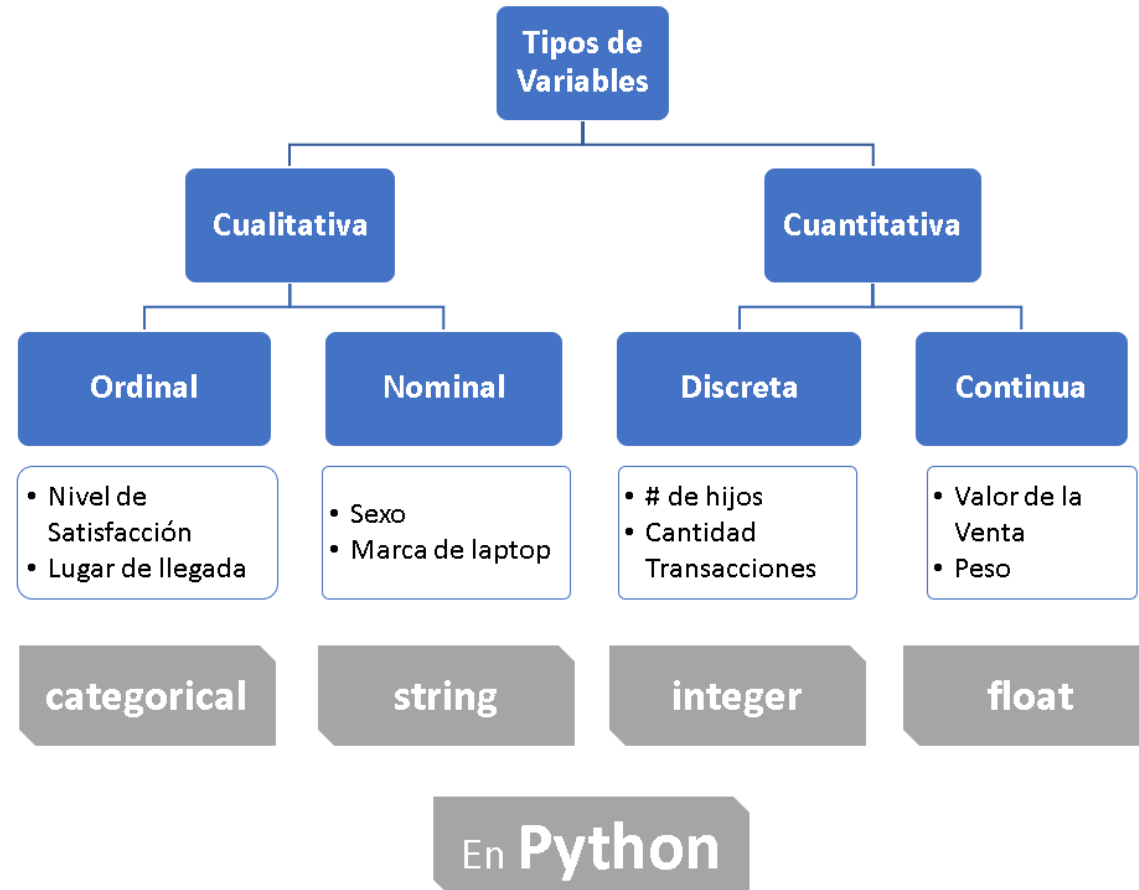
Casos especiales: Not a Number

```
0.0 * neg_inf
```

```
## nan
```

Tipos de variables

Tipos de variables y su correspondencia en Python



Tipos de datos

Revisar que todas las columnas tengan el tipo correcto y además la relación que existe entre los dos dataframes importados.

```
data_banco_xlsx.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 24299 entries, 0 to 24298
## Data columns (total 7 columns):
##  #   Column                Non-Null Count  Dtype
## ---  -
##  0   Sucursal              24299 non-null  int64
##  1   Cajero                 24299 non-null  int64
##  2   ID_Transaccion        24299 non-null  int64
##  3   Transaccion           24299 non-null  object
##  4   Tiempo_Servicio_seg   24299 non-null  float64
##  5   Satisfaccion          24299 non-null  object
##  6   Monto                 24299 non-null  object
## dtypes: float64(1), int64(3), object(3)
## memory usage: 1.3+ MB
```

Tipos de datos

Revisar que todas las columnas tengan el tipo correcto y además la relación que existe entre los dos dataframes importados.

```
data_sucursal.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 5 entries, 0 to 4
## Data columns (total 3 columns):
##  #   Column                Non-Null Count  Dtype
## ---  -
##  0   ID_Sucursal            5 non-null     int64
##  1   Sucursal                5 non-null     object
##  2   Nuevo_Sistema          5 non-null     object
## dtypes: int64(1), object(2)
## memory usage: 248.0+ bytes
```



Entender los datos

Luego de importar se debe entender los datos

- ¿Qué representa cada columna?
- ¿Qué tipo de dato debería tener cada columna?
- ¿Qué granularidad o atomicidad tiene la data?
- Si es que se tiene varios conjuntos de datos ¿Cómo se relacionan los datos?
- A qué periodo de tiempo corresponde la data
- Muchas veces se obtiene la información desde una base de datos y por tanto toca entender la base y el query que genera los datos



Entender los datos - Ejemplo

¿Están bien nuestros tipos de datos?

...

```
# Ver la estructura del data.frame
data_banco_xlsx.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 24299 entries, 0 to 24298
## Data columns (total 7 columns):
##  #   Column                Non-Null Count  Dtype
## ---  -
## 0   Sucursal              24299 non-null  int64
## 1   Cajero                24299 non-null  int64
## 2   ID_Transaccion        24299 non-null  int64
## 3   Transaccion           24299 non-null  object
## 4   Tiempo_Servicio_seg   24299 non-null  float64
## 5   Satisfaccion          24299 non-null  object
## 6   Monto                 24299 non-null  object
## dtypes: float64(1), int64(3), object(3)
## memory usage: 1.3+ MB
```

Entender los datos - Ejemplo

¿Están bien nuestros tipos de datos?

...

```
# Ver la estructura del data.frame  
data_sucursal.info()
```

```
## <class 'pandas.core.frame.DataFrame'>  
## RangeIndex: 5 entries, 0 to 4  
## Data columns (total 3 columns):  
##  #      Column      Non-Null Count  Dtype  
## ---  -  
## 0    ID_Sucursal    5 non-null    int64  
## 1    Sucursal         5 non-null    object  
## 2    Nuevo_Sistema    5 non-null    object  
## dtypes: int64(1), object(2)  
## memory usage: 248.0+ bytes
```




Entender los datos

Del entendimiento de nuestros datos podemos ver que:

- Hay columnas numéricas que deben ser texto,
- Satisfacción debería ser categórica,
- El Monto debemos convertirlo a numérico,
- La data de sucursales se puede agregar a la de transacciones por el código de la sucursal.

Para poder realizar eso debemos aprender a manejar (o manipular) dataFrames, esto es: seleccionar columnas, filtrar filas, modificar columnas, unir dos conjuntos de datos.

Manipulacion de datos - Basico

Curso: Introducción a Python para Análisis de Datos



Seleccionar columnas: `[[,]]`

Seleccionar una columna

```
# Seleccionar una columna  
data_banco_xlsx['Monto']
```

```
## 0          2889,3  
## 1          1670,69  
## 2          3172,49  
## 3          1764.92  
## 4          1835.69  
##          ...  
## 24294         657.38  
## 24295         763.65  
## 24296        3326.79  
## 24297        1237.91  
## 24298        1643.14  
## Name: Monto, Length: 24299, dtype: object
```



Seleccionar columnas: `[[,]]`

Seleccionar una columna

```
# Seleccionar una columna  
data_banco_xlsx.Monto
```

```
## 0          2889,3  
## 1          1670,69  
## 2          3172,49  
## 3          1764.92  
## 4          1835.69  
##          ...  
## 24294         657.38  
## 24295         763.65  
## 24296        3326.79  
## 24297        1237.91  
## 24298        1643.14  
## Name: Monto, Length: 24299, dtype: object
```



Seleccionar columnas: `[[,]]`

Seleccionar una columna

```
# Seleccionar una columna  
data_banco_xlsx[['Monto']]
```

```
##           Monto  
## 0           2889,3  
## 1           1670,69  
## 2           3172,49  
## 3           1764.92  
## 4           1835.69  
## ...           ...  
## 24294        657.38  
## 24295        763.65  
## 24296       3326.79  
## 24297       1237.91  
## 24298       1643.14  
##  
## [24299 rows x 1 columns]
```

Seleccionar columnas: `[[,]]`

Seleccionar varias columnas

```
# Seleccionar varias columnas
data_banco_xlsx[ ['Tiempo_Servicio_seg', 'Sucursal'] ]
```

```
##      Tiempo_Servicio_seg  Sucursal
## 0                311.0         62
## 1                156.0         62
## 2                248.0         62
## 3                 99.0         62
## 4                123.0         62
## ...                ...         ...
## 24294             184.0        586
## 24295             124.0        586
## 24296             141.0        586
## 24297              54.0        586
## 24298             105.0        586
##
## [24299 rows x 2 columns]
```



Seleccionar columnas: `[[,]]`

Para seleccionar dos columnas por número usando *iloc*

```
# Seleccionar varias columnas
data_banco_xlsx.iloc[ :, [1 , 2]]
```

```
##          Cajero  ID_Transaccion
## 0          4820             2
## 1          4820             2
## 2          4820             2
## 3          4820             2
## 4          4820             2
## ...          ...             ...
## 24294        4424             10
## 24295        4424             10
## 24296        4424             10
## 24297        4424             10
## 24298        4424             10
##
## [24299 rows x 2 columns]
```



Seleccionar columnas: `[[,]]`

Para seleccionar dos columnas por número usando *iloc*

```
# Seleccionar varias columnas
data_banco_xlsx.iloc[ :, 0:4]
```

```
##          Sucursal  Cajero  ID_Transaccion  Transaccion
## 0           62    4820           2    Cobro/Pago (Cta externa)
## 1           62    4820           2    Cobro/Pago (Cta externa)
## 2           62    4820           2    Cobro/Pago (Cta externa)
## 3           62    4820           2    Cobro/Pago (Cta externa)
## 4           62    4820           2    Cobro/Pago (Cta externa)
## ...         ...      ...         ...         ...
## 24294        586    4424          10    Cobrar cheque (Cta del Bco)
## 24295        586    4424          10    Cobrar cheque (Cta del Bco)
## 24296        586    4424          10    Cobrar cheque (Cta del Bco)
## 24297        586    4424          10    Cobrar cheque (Cta del Bco)
## 24298        586    4424          10    Cobrar cheque (Cta del Bco)
##
## [24299 rows x 4 columns]
```


Seleccionar columnas: `[[,]]`

Seleccionar varias columnas usando un slice

```
# Seleccionar varias columnas
data_banco_xlsx.loc[:, 'Cajero':'Transaccion']
```

```
##          Cajero  ID_Transaccion          Transaccion
## 0          4820           2      Cobro/Pago (Cta externa)
## 1          4820           2      Cobro/Pago (Cta externa)
## 2          4820           2      Cobro/Pago (Cta externa)
## 3          4820           2      Cobro/Pago (Cta externa)
## 4          4820           2      Cobro/Pago (Cta externa)
## ...          ...           ...          ...
## 24294       4424          10  Cobrar cheque (Cta del Bco)
## 24295       4424          10  Cobrar cheque (Cta del Bco)
## 24296       4424          10  Cobrar cheque (Cta del Bco)
## 24297       4424          10  Cobrar cheque (Cta del Bco)
## 24298       4424          10  Cobrar cheque (Cta del Bco)
##
## [24299 rows x 3 columns]
```

Filtrar Filas usando un slice

Filtrar filas usando un slice

```
# Filtrar filas usando un slice  
data_banco_xlsx[2:7]
```

```
##      Sucursal  Cajero  ID_Transaccion  ...  Tiempo_Servicio_seg  Satisfaccion  Monto  
## 2          62    4820          2  ...          248.0        Regular  3172,49  
## 3          62    4820          2  ...           99.0        Regular  1764.92  
## 4          62    4820          2  ...          123.0      Muy Bueno  1835.69  
## 5          62    4820          2  ...          172.0        Bueno  2165.42  
## 6          62    4820          2  ...          140.0        Regular  1304.9  
##  
## [5 rows x 7 columns]
```



Filtrar filas por posición o condición

Para Filtrar filas según número de fila o según el cumplimiento de condiciones se usa `.i loc` y `.loc` respectivamente, ojo que para usar `.loc` debemos saber los operadores de relación y lógicos en Python.

Operadores de relación en Python

```
3 == 3 # Igualdad
```

```
## True
```

```
3 == 3.0 # Igualdad
```

```
## True
```

```
3 >= 1 # Mayor igual
```

```
## True
```

```
3 < 2 # Menor
```

```
## False
```

```
3 != 7 # Diferente
```

```
## True
```

Operadores de relación en Python

Para aprender a filtrar por condiciones, debemos aprender a "preguntar"

```
lista= ['a', 'b', 'c', 'd']  
'c' in lista
```

```
## True
```

```
'z' in lista
```

```
## False
```

```
'c' not in lista
```

```
## False
```

```
'z' not in lista
```

```
## True
```



Operadores de relación en Python

Para aprender a filtrar por condiciones, debemos aprender a "preguntar"

```
a = 'x'  
a in lista
```

```
## False
```

```
a = 'c'  
a in lista
```

```
## True
```

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

```
## True
```

```
type(a) is not str
```

```
## False
```



Operadores de relación en Python

Para aprender a filtrar por condiciones, debemos aprender a "preguntar"

```
True & True
```

```
## True
```

```
True & False
```

```
## False
```

```
False | False
```

```
## False
```

```
False | True
```

```
## True
```

Filtrar Filas por número

Filtrar Filas por número

```
# Filtrar filas por numero de fila  
# data_banco_xlsx.iloc[[5,6,7]]  
data_banco_xlsx.loc[[5,6,7]]
```

```
##      Sucursal  Cajero  ID_Transaccion  ...  Tiempo_Servicio_seg  Satisfaccion  Monto  
## 5           62    4820           2  ...           172.0           Bueno  2165.42  
## 6           62    4820           2  ...           140.0          Regular  1304.9  
## 7           62    4820           2  ...           247.0           Bueno  4080.05  
##  
## [3 rows x 7 columns]
```


Filtrar Filas por número

Filtrar Filas por número usando un slice (sólo iloc)

```
# Filtrar Filas por número usando un slice (sólo iloc)
data_banco_xlsx.iloc[ 0:4]
```

```
##      Sucursal  Cajero  ID_Transaccion  ...  Tiempo_Servicio_seg  Satisfaccion  Monto
## 0           62    4820                2  ...              311.0      Muy Bueno  2889,3
## 1           62    4820                2  ...              156.0           Malo  1670,69
## 2           62    4820                2  ...              248.0      Regular  3172,49
## 3           62    4820                2  ...               99.0      Regular  1764.92
##
## [4 rows x 7 columns]
```



Filtrar filas según una condición

Filtrar filas según una condición *loc*, filtrar transacciones cuyo tiempo de servicio sea mayor a 100

```
# Filtrar filas según una condición loc
# Filtrar transacciones cuyo tiempo de servicio sea mayor a 100
data_banco_xlsx.loc[data_banco_xlsx["Tiempo_Servicio_seg"] > 100]
```

```
##          Sucursal  Cajero  ...  Satisfaccion  Monto
## 0             62    4820  ...      Muy Bueno  2889,3
## 1             62    4820  ...           Malo  1670,69
## 2             62    4820  ...      Regular  3172,49
## 4             62    4820  ...      Muy Bueno  1835.69
## 5             62    4820  ...          Bueno  2165.42
## ...          ...      ...  ...          ...      ...
## 24292         586    4424  ...           Malo   2582.1
## 24294         586    4424  ...      Muy Malo   657.38
## 24295         586    4424  ...          Bueno   763.65
## 24296         586    4424  ...          Bueno  3326.79
## 24298         586    4424  ...           Malo  1643.14
##
## [14809 rows x 7 columns]
```



Filtrar filas según una condición

Filtrar filas según una condición *loc*, filtrar transacciones cuyo tiempo de servicio sea mayor a 100 y que se hayan realizado en la sucursal 62

```
# Filtrar filas según una condición loc
data_banco_xlsx.loc[ (data_banco_xlsx["Tiempo_Servicio_seg"] > 100) &
                     (data_banco_xlsx["Sucursal"] == 62) ]
```

```
##          Sucursal  Cajero  ...  Satisfaccion  Monto
## 0             62    4820  ...      Muy Bueno  2889,3
## 1             62    4820  ...           Malo  1670,69
## 2             62    4820  ...      Regular  3172,49
## 4             62    4820  ...      Muy Bueno  1835.69
## 5             62    4820  ...           Bueno  2165.42
## ...          ...      ...  ...          ...      ...
## 2831           62    5286  ...           Malo  3265.79
## 2832           62    5286  ...           Bueno  1144.88
## 2833           62    5286  ...      Regular  1779.14
## 2834           62    5286  ...           Malo    847.3
## 2837           62    5286  ...      Regular  1231.13
##
## [886 rows x 7 columns]
```



Filtrar filas y Seleccionar columnas

filtrar por numero de fila y Seleccionar según número de Columna

```
# Filtrar por numero de fila y Seleccionar según número de Columna  
data_banco_xlsx.iloc[[5,6,7], [4, 1]]
```

##	Tiempo_Servicio_seg	Cajero
## 5	172.0	4820
## 6	140.0	4820
## 7	247.0	4820



Filtrar filas y Seleccionar columnas

Filtrar por numero de fila y Seleccionar según nombre de Columna

```
# Filtrar por numero de fila y Seleccionar según nombre de Columna  
data_banco_xlsx.loc[[5,6,7], ['Tiempo_Servicio_seg', 'Sucursal']]
```

##	Tiempo_Servicio_seg	Sucursal
## 5	172.0	62
## 6	140.0	62
## 7	247.0	62



Filtrar filas y Seleccionar columnas

Filtrar por numero de fila y Seleccionar según nombre de Columna

```
# Filtrar filas según una condición, Seleccionar según número de Columna
data_banco_xlsx.loc[ (data_banco_xlsx["Tiempo_Servicio_seg"] > 100) &
    (data_banco_xlsx["Sucursal"] == 62), ['Tiempo_Servicio_seg', 'Sucursal']]
```

```
##      Tiempo_Servicio_seg  Sucursal
## 0                311.0         62
## 1                156.0         62
## 2                248.0         62
## 4                123.0         62
## 5                172.0         62
## ...                ...         ...
## 2831             220.0         62
## 2832             142.0         62
## 2833             113.0         62
## 2834             134.0         62
## 2837             156.0         62
##
## [886 rows x 2 columns]
```

Crear o modificar columnas/variables

`dataFrame['nueva_Var'] =`

Crear una nueva columna con el tiempo en minutos, opc

```
# Crear una nueva columna con el tiempo en minutos
data_banco_xlsx['Tiempo_Servicio_Min'] = data_banco_xlsx['Tiempo_Servicio_seg']/60
data_banco_xlsx.head(5)
```

```
##      Sucursal  Cajero  ID_Transaccion  ...  Satisfaccion    Monto  Tiempo_Servicio_Min
## 0           62    4820           2  ...    Muy Bueno    2889,3           5.183333
## 1           62    4820           2  ...           Malo    1670,69           2.600000
## 2           62    4820           2  ...    Regular    3172,49           4.133333
## 3           62    4820           2  ...    Regular    1764.92           1.650000
## 4           62    4820           2  ...    Muy Bueno    1835.69           2.050000
##
## [5 rows x 8 columns]
```

Crear o modificar columnas/variables

nuevo_df = dataframe.assign(nueva_Var= lambda x: ...), a un nuevo DF
dataframe = dataframe.assign(nueva_Var= lambda x: ...), a mismo DF

Crear una nueva columna con el tiempo en minutos, opc

```
# Crear una nueva columna con el tiempo en minutos
data_banco_xlsx = data_banco_xlsx.assign(
    Tiempo_Servicio_Min2= lambda x: x.Tiempo_Servicio_seg/60)
data_banco_xlsx.head(5)
```

```
##      Sucursal  Cajero  ...  Tiempo_Servicio_Min  Tiempo_Servicio_Min2
## 0           62    4820  ...           5.183333           5.183333
## 1           62    4820  ...           2.600000           2.600000
## 2           62    4820  ...           4.133333           4.133333
## 3           62    4820  ...           1.650000           1.650000
## 4           62    4820  ...           2.050000           2.050000
##
## [5 rows x 9 columns]
```


lambda x -> función anónima o función lambda

Las funciones lambda son funciones pequeñas y anónimas que se pueden usar en el lugar donde se requiere una función. En el contexto de Pandas, lambda x se usa comúnmente en combinación con métodos como `apply()`, `map()`, `assign()`, etc. Permite aplicar una operación específica a cada elemento de una Serie o DataFrame.

```
# Crear una nueva columna con el tiempo en minutos
data_banco_xlsx = data_banco_xlsx.assign(
    Tiempo_Servicio_Min2= lambda x: x.Tiempo_Servicio_seg/60)
data_banco_xlsx.head(5)
```

```
##      Sucursal  Cajero  ...  Tiempo_Servicio_Min  Tiempo_Servicio_Min2
## 0           62    4820  ...           5.183333           5.183333
## 1           62    4820  ...           2.600000           2.600000
## 2           62    4820  ...           4.133333           4.133333
## 3           62    4820  ...           1.650000           1.650000
## 4           62    4820  ...           2.050000           2.050000
##
## [5 rows x 9 columns]
```



Modificar o crear columnas

Podemos crear una nueva columna calculada y quedarnos sólo con dicha columna haciendo:

```
# Crear una columna pero sólo mantener dicha columna  
data_banco_xlsx.apply(lambda x: x.Tiempo_Servicio_seg/60, axis= 1)
```

```
## 0          5.183333  
## 1          2.600000  
## 2          4.133333  
## 3          1.650000  
## 4          2.050000  
##          ...  
## 24294       3.066667  
## 24295       2.066667  
## 24296       2.350000  
## 24297       0.900000  
## 24298       1.750000  
## Length: 24299, dtype: float64
```

Modificar o crear columnas

Crear una nueva columna usando `.apply` y asignarla a una nueva columna

```
# Crear una nueva columna usando .apply y asignarla a una nueva columna
data_banco_xlsx['Tiempo_Servicio_Min3'] = data_banco_xlsx.apply(
    lambda x: x.Tiempo_Servicio_seg/60, axis= 1)
data_banco_xlsx.head(5)
```

```
##      Sucursal  Cajero  ...  Tiempo_Servicio_Min2  Tiempo_Servicio_Min3
## 0           62    4820  ...                5.183333                5.183333
## 1           62    4820  ...                2.600000                2.600000
## 2           62    4820  ...                4.133333                4.133333
## 3           62    4820  ...                1.650000                1.650000
## 4           62    4820  ...                2.050000                2.050000
##
## [5 rows x 10 columns]
```

La utilidad de apply y las funciones lambda

Supongamos que deseamos filtrar las filas que tengan más de 2 palabras en la columna Transaccion, intentemos:

```
# Filtrar las filas que tengan más de 2 palabras en la Transaccion (primer intento fallido)
# Para saber que tenemos más de dos palabras, vamos a cortar la frase usando los espacios
# con el metodo split(" ") y luego contaremos las palabras que queden.
data_banco_xlsx[len(data_banco_xlsx['Transaccion'].split(" "))>2] ## ERROR
# Incluso sólo la parte del split() también da error
data_banco_xlsx['Transaccion'].split(" ") ## ERROR
```

¡Pero en el string independiente sí se puede! ¿Por qué? Pues porque split es un método de los string, no de las listas ni Series

```
a= "Esto es un string"
a.split(" ") # Probar split en un string
```

```
## ['Esto', 'es', 'un', 'string']
```

```
len( a.split(" ") ) # Contar cantidad de palabras
```

```
## 4
```



La utilidad de apply y las funciones lambda

Se puede utilizar las bondades de `.apply` para hacer filtros/cálculos complicados.

Supongamos que deseamos filtrar las filas que tengan más de 2 palabras en la columna Transaccion; para saber que tenemos más de dos palabras, vamos a cortar la frase usando los espacios con el método `split(" ")` y luego contaremos las palabras que queden, pero ya vimos que `split()` es un método de los string, así que **debo aplicarlo a cada elemento de mi columna, para eso se usa `.apply`**

Más en: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.apply.html>

```
# Con .apply se genera un vector booleano (true - false)
# axis{0 or 'index', 1 or 'columns'}, default 0
data_banco_xlsx.apply( lambda x: len(x['Transaccion'].split(" "))>2, axis= 1 ) # Vector Booleano
```

```
## 0      True
## 1      True
## 2      True
## 3      True
## 4      True
##      ...
## 24294   True
## 24295   True
## 24296   True
## 24297   True
## 24298   True
## Length: 24299, dtype: bool
```



La utilidad de apply y las funciones lambda

Se puede utilizar las bondades de `.apply` para hacer filtros complicados.

Supongamos que deseamos filtrar las filas que tengan más de 2 palabras en la columna Transaccion; para saber que tenemos más de dos palabras, vamos a cortar la frase usando los espacios con el metodo `split(" ")` y luego contaremos las palabras que queden, pero ya vimos que `split()` es un método de los string, así que **debo aplicarlo a cada elemento de mi columna, para eso se usa `.apply`**

```
# Con .apply se genera un vector booleano (true - false)
# Y dicho vector se usa para filtrar
data_banco_xlsx.loc[ data_banco_xlsx.apply(
    lambda x: len(x['Transaccion'].split(" "))>2, axis= 1 ) ]
```

```
##      Sucursal  Cajero  ...  Tiempo_Servicio_Min2  Tiempo_Servicio_Min3
## 0           62   4820  ...           5.183333           5.183333
## 1           62   4820  ...           2.600000           2.600000
## 2           62   4820  ...           4.133333           4.133333
## 3           62   4820  ...           1.650000           1.650000
## 4           62   4820  ...           2.050000           2.050000
## ...      ...      ...  ...           ...           ...
## 24294        586   4424  ...           3.066667           3.066667
## 24295        586   4424  ...           2.066667           2.066667
## 24296        586   4424  ...           2.350000           2.350000
## 24297        586   4424  ...           0.900000           0.900000
## 24298        586   4424  ...           1.750000           1.750000
##
## [8412 rows x 10 columns]
```



La utilidad de apply y las funciones lambda

Veamos las transacciones unicas para confirmar que filtró bien el paso anterior

```
# Para verificar el resultado del filtro, podría obtener los valores unicos  
# de las transacciones que quedaaron luego del filtro  
data_banco_xlsx.loc[ data_banco_xlsx.apply(  
    lambda x: len(x['Transaccion'].split(" "))>2,  
    axis= 1 )].Transaccion.unique()
```

```
## array(['Cobro/Pago (Cta externa)', 'Cobrar cheque (Cta del Bco)'],  
##      dtype=object)
```



Eliminar columnas

Para eliminar se usa `del` y `.drop`

```
# borrar una columna "in-place"  
del data_banco_xlsx['Tiempo_Servicio_Min3']  
# borrar varias columnas (se debe asignar)  
# data_banco_xlsx.drop(columns= ['Tiempo_Servicio_Min', 'Tiempo_Servicio_Min2'])
```




Ordenar los datos

Para ordenar los datos usamos `.sort_values()` así:

`df.sort_values("columna")`

`df.sort_values("columna", ascenascending=False) <- descendente`

```
data_banco_xlsx.sort_values( "Tiempo_Servicio_seg")
```

```
##          Sucursal  Cajero  ...  Tiempo_Servicio_Min  Tiempo_Servicio_Min2
## 10425          85    3983  ...           0.302196           0.302196
## 7162           85     472  ...           0.302490           0.302490
## 11871          85    3983  ...           0.316781           0.316781
## 12021          85    3983  ...           0.332725           0.332725
## 1211           62   5211  ...           0.333333           0.333333
## ...          ...     ...  ...           ...           ...
## 21032         443   4208  ...          20.220172          20.220172
## 10368          85    3983  ...          20.800597          20.800597
## 5735           85     472  ...          21.095559          21.095559
## 8325           85    3678  ...          22.276138          22.276138
## 10330          85    3983  ...          26.711639          26.711639
##
## [24299 rows x 9 columns]
```



Ordenar los datos

Para ordenar los datos usamos `.sort_values()` así:

`df.sort_values("columna")`

`df.sort_values("columna", ascenascending=False) <- descendente`

```
data_banco_xlsx.sort_values( ["Transaccion", "Tiempo_Servicio_seg"], ascending= [True, False])
```

```
##          Sucursal  Cajero  ...  Tiempo_Servicio_Min  Tiempo_Servicio_Min2
## 16916         267    2556  ...          15.218604          15.218604
## 20634         443    3732  ...          13.814274          13.814274
## 20644         443    3732  ...          13.492129          13.492129
## 17960         267    4796  ...          13.226732          13.226732
## 9924          85    3678  ...          12.964907          12.964907
## ...          ...      ...  ...              ...              ...
## 24245         586    4424  ...           0.333333           0.333333
## 12021          85    3983  ...           0.332725           0.332725
## 11871          85    3983  ...           0.316781           0.316781
## 7162          85     472  ...           0.302490           0.302490
## 10425          85    3983  ...           0.302196           0.302196
##
## [24299 rows x 9 columns]
```



Entender los datos - Ejemplo

¿Está bien nuestros tipos de datos?

Si no lo están entonces debemos transformarlos.

```
# Ver la estructura del data.frame
data_banco_xlsx.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 24299 entries, 0 to 24298
## Data columns (total 9 columns):
##  #      Column              Non-Null Count  Dtype
## ---  -
## 0      Sucursal              24299 non-null  int64
## 1      Cajero                  24299 non-null  int64
## 2      ID_Transaccion          24299 non-null  int64
## 3      Transaccion             24299 non-null  object
## 4      Tiempo_Servicio_seg     24299 non-null  float64
## 5      Satisfaccion            24299 non-null  object
## 6      Monto                   24299 non-null  object
## 7      Tiempo_Servicio_Min     24299 non-null  float64
## 8      Tiempo_Servicio_Min2    24299 non-null  float64
## dtypes: float64(3), int64(3), object(3)
## memory usage: 1.7+ MB
```



Ejemplo - Manipulacion de datos

Lo primero que necesitamos es corregir los tipos de datos, nótese que

- **Monto** tiene una mezcla de "," y "."
- **Sucursal** y **Cajero** deberían ser de tipo character
- **Satisfaccion** debe ser factor ordenado

```
# Modificar la coma por punto en Monto luego transformar a numérico
data_banco_xlsx['Monto'] = data_banco_xlsx['Monto'].replace(',', '.', regex=True)
data_banco_xlsx["Monto"] = pd.to_numeric(data_banco_xlsx.Monto, errors='coerce')
# data_banco_xlsx['Monto'].head(4)
```



Ejemplo - Manipulacion de datos

Lo primero que necesitamos es corregir los tipos de datos, nótese que

- **Monto** tiene una mezcla de "," y "."
- **Sucursal** y **Cajero** deberían ser de tipo character
- **Satisfaccion** debe ser factor ordenado

```
# Modificar a String
data_banco_xlsx['Sucursal'] = data_banco_xlsx['Sucursal'].astype(str)
data_banco_xlsx['Cajero'] = data_banco_xlsx['Cajero'].astype(str)
data_banco_xlsx['ID_Transaccion'] = data_banco_xlsx['ID_Transaccion'].astype(str)
```



Ejemplo - Manipulacion de datos

Lo primero que necesitamos es corregir los tipos de datos, nótese que

- **Monto** tiene una mezcla de "," y "."
- **Sucursal** y **Cajero** deberían ser de tipo character
- **Satisfaccion** debe ser factor ordenado

```
## Dato Categorical
data_banco_xlsx['Satisfaccion'] = pd.Categorical(
    data_banco_xlsx['Satisfaccion'],
    categories= ['Muy Malo', 'Malo', 'Regular', 'Bueno', 'Muy Bueno'],
    ordered=True)
data_banco_xlsx.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 24299 entries, 0 to 24298
## Data columns (total 9 columns):
## #      Column              Non-Null Count  Dtype
## ---  -
## 0      Sucursal              24299 non-null  object
## 1      Cajero                 24299 non-null  object
## 2      ID_Transaccion          24299 non-null  object
## 3      Transaccion             24299 non-null  object
## 4      Tiempo_Servicio_seg     24299 non-null  float64
```

Resumir/Agregar los Datos

Taller: Python (Pandas) para análisis de datos

Néstor Montaña P.



Estadística descriptiva - Estadísticos | Medidas

Pandas tiene ya desarrollado algunos de las medidas estadísticas más usadas, pero además en python tenemos varios paquetes adicionales como por ejemplo:

- `import statistics`
- `from scipy import stats` o `import scipy.stats`
- Más información de lo que se puede calcular en pandas en <https://pandas.pydata.org/pandas-docs/stable/reference/frame.html#computations-descriptive-stats>



Estadística descriptiva - Estadísticos | Medidas

Tenemos por ejemplo:

Media.- Promedio de los valores

- Se la puede entender como el punto de equilibrio
- Muy sensible a valores aberrantes
- `dataframe.mean(x, na.rm= TRUE)`

Media Acotada.- Promedio de los valores, pero quitando un porcentaje de valores extremos.

- Es menos sensible a valores aberrantes, pero se puede perder información importante
- `scipy.stats.trim_mean(, trim es porcentaje a quitar a cada lado)`
- En pandas se puede combinar `.clip` con `.mean` pero es trabajoso

Veamos algunas de las cosas que se puede calcular con Python

Medidas de Tendencia Central

```
# Media del tiempo de servicio  
data_banco_xlsx['Tiempo_Servicio_seg'].mean()
```

```
## 155.579993233514
```

```
# Media acotada del Monto  
scipy.stats.trim_mean(data_banco_xlsx.Monto , 0.05)
```

```
## 1982.6435558502126
```

```
# Mediana del tiempo de servicio  
data_banco_xlsx['Tiempo_Servicio_seg'].median()
```

```
## 122.45229035353
```



Medidas de Posición

Calcular las medidas de Posición para el tiempo de servicio data de Banco

```
# Mínimo y Máximo
data_banco_xlsx.Tiempo_Servicio_seg.min()
```

```
## 18.1317703726497
```

```
data_banco_xlsx.Tiempo_Servicio_seg.max()
```

```
## 1602.69831855495
```

```
# Mínimo y Máximo
# Cuartiles
data_banco_xlsx.Tiempo_Servicio_seg.quantile( [0.25, 0.50, 0.75] )
```

```
## 0.25      75.691187
## 0.50     122.452290
## 0.75     197.730457
## Name: Tiempo_Servicio_seg, dtype: float64
```

```
# Percentiles específicos
data_banco_xlsx.quantile( [0, 0.03, 0.25, 0.50, 0.75, 0.97, 1] )
```

```
##      Tiempo_Servicio_seg      Monto  Tiempo_Servicio_Min  Tiempo_Servicio_Min2
```



Medidas de Posición - Boxplot

Boxplot.- Muestra gráficamente las medidas de posición, se puede usar varios paquetes para realizar gráficos tanto estáticos como dinámicos en python, esto no se verá en el presente curso, sin embargo se muestra un ejemplo:

```
# Un primer Boxplot  
sns.boxplot(x= "Tiempo_Servicio_seg", data= data_banco_xlsx)
```



Varias medidas estadísticas en un sólo comando

Con `.describe()` se obtienen algunas estadísticas descriptivas

```
data_banco_xlsx.describe()
```

```
##      Tiempo_Servicio_seg  ...  Tiempo_Servicio_Min2
## count      24299.000000  ...      24299.000000
## mean        155.579993  ...        2.593000
## std         120.009457  ...        2.000158
## min         18.131770  ...        0.302196
## 25%         75.691187  ...        1.261520
## 50%        122.452290  ...        2.040872
## 75%        197.730457  ...        3.295508
## max        1602.698319  ...       26.711639
##
## [8 rows x 4 columns]
```



Y ahora, todo junto

Con `.describe()` se obtienen algunas estadísticas descriptivas

```
data_banco_xlsx.describe( percentiles= [0, 0.03, 0.25, 0.50, 0.75, 0.97, 1] )
```

```
##      Tiempo_Servicio_seg  ...  Tiempo_Servicio_Min2
## count      24299.000000  ...      24299.000000
## mean        155.579993  ...        2.593000
## std         120.009457  ...        2.000158
## min         18.131770  ...        0.302196
## 0%          18.131770  ...        0.302196
## 3%          33.033362  ...        0.550556
## 25%          75.691187  ...        1.261520
## 50%         122.452290  ...        2.040872
## 75%         197.730457  ...        3.295508
## 97%         456.783428  ...        7.613057
## 100%        1602.698319  ...       26.711639
## max         1602.698319  ...       26.711639
##
## [12 rows x 4 columns]
```



Cálculos con agrupamiento

Pandas permite obtener resúmenes o cálculos agrupando por los valores de una variable del dataframe, como `summarise + group_by` en R o un `Select, from, group by` en SQL.

```
## Obtener las descriptivas del Monto por Transaccion
data_banco_xlsx.groupby('Transaccion')['Tiempo_Servicio_seg'].describe()
```

```
##              count      mean  ...      75%      max
## Transaccion
## Cobrar cheque (Cta del Bco)    5407.0    185.865204  ...    240.299657    913.116263
## Cobro/Pago (Cta externa)      3005.0    301.428249  ...    386.408048   1602.698319
## Deposito                      15887.0    117.685731  ...    153.712160    594.796607
##
## [3 rows x 8 columns]
```



Cálculos con agrupamiento

Pandas permite obtener resúmenes o cálculos agrupando por los valores de una variable del dataframe, como `summarise + group_by` en R o un `Select, from, group by` en SQL.

```
## Obtiene las descriptivas por Transaccion
data_banco_xlsx.groupby('Transaccion').describe()
```

```
##                Tiempo_Servicio_seg  ... Tiempo_Servicio_Min2
##                                count  ...                      max
## Transaccion                        ...
## Cobrar cheque (Cta del Bco)        5407.0  ...             15.218604
## Cobro/Pago (Cta externa)          3005.0  ...             26.711639
## Deposito                          15887.0  ...              9.913277
##
## [3 rows x 32 columns]
```




Cálculos con agrupamiento

Pandas permite obtener resúmenes o cálculos agrupando por los valores de una variable del dataframe, como `summarise + group_by` en R o un `Select, from, group by` en SQL.

```
## Obtiene las Media del Tiempo y Monto por Transaccion  
data_banco_xlsx.groupby('Transaccion')[['Tiempo_Servicio_seg', 'Monto']].mean()
```

##	Tiempo_Servicio_seg	Monto
## Transaccion		
## Cobrar cheque (Cta del Bco)	185.865204	2079.749432
## Cobro/Pago (Cta externa)	301.428249	2448.715328
## Deposito	117.685731	1882.105087



Cálculos con agrupamiento

También soporta agrupar por varias columnas así como varios cálculos

```
## Obtiene las Media del Tiempo y Monto por Sucursal y Nivel de Satisfaccion
data_banco_xlsx.groupby(['Sucursal', 'Satisfaccion'])[['Tiempo_Servicio_seg', 'Monto']].mean()
```

##		Tiempo_Servicio_seg	Monto
##	Sucursal Satisfaccion		
## 267	Muy Malo	178.288403	2070.285083
##	Malo	176.860435	2058.706568
##	Regular	175.813945	1994.600732
##	Bueno	182.631086	2077.691598
##	Muy Bueno	202.188738	2122.173794
## 443	Muy Malo	150.047746	1969.573623
##	Malo	162.763379	2026.256820
##	Regular	192.666638	2121.600182
##	Bueno	185.257154	2102.173927
##	Muy Bueno	191.550564	2101.123532
## 586	Muy Malo	80.352239	1734.234866
##	Malo	77.351706	1670.242178
##	Regular	80.347826	1734.118435
##	Bueno	82.795337	1705.907176
##	Muy Bueno	89.141907	1751.864612
## 62	Muy Malo	85.751740	1782.139327
##	Malo	86.350993	1741.769421
##	Regular	88.242308	1770.012019
##	Bueno	92.529240	1735.567164
##	Muy Bueno	92.495826	1773.558514

Cálculos con agrupamiento con agg

También soporta agrupar por varias columnas así como varios cálculos

```
## Obtiene las Media y Mediana del
## Tiempo y Monto por Sucursal y Nivel de Satisfaccion
## Se usa .agg()
data_banco_xlsx[['Tiempo_Servicio_seg', 'Monto', 'Sucursal',
                  'Satisfaccion']].groupby(['Sucursal', 'Satisfaccion']).agg(["mean", "median"])
```

##		Tiempo_Servicio_seg		Monto	
##		mean	median	mean	median
##	Sucursal Satisfaccion				
##	267 Muy Malo	178.288403	149.466417	2070.285083	2108.840
##	Malo	176.860435	148.075930	2058.706568	2143.855
##	Regular	175.813945	142.179657	1994.600732	2095.580
##	Bueno	182.631086	145.554547	2077.691598	2163.910
##	Muy Bueno	202.188738	159.871332	2122.173794	2194.420
##	443 Muy Malo	150.047746	129.687657	1969.573623	2073.580
##	Malo	162.763379	132.723704	2026.256820	2083.710
##	Regular	192.666638	156.922623	2121.600182	2179.240
##	Bueno	185.257154	145.259379	2102.173927	2170.020
##	Muy Bueno	191.550564	150.487781	2101.123532	2170.550
##	586 Muy Malo	80.352239	72.000000	1734.234866	1854.810
##	Malo	77.351706	69.000000	1670.242178	1727.580
##	Regular	80.347826	69.000000	1734.118435	1793.450
##	Bueno	82.795337	70.000000	1705.907176	1813.520
##	Muy Bueno	89.141907	76.000000	1751.864612	1901.480

reset_index() para que los índices sean columnas

reset_index() es un método en Pandas que restablece el índice de un DataFrame o Serie, genera un nuevo índice numérico y el índice anterior se convertirá en una nueva columna en el DataFrame.

```
## Obtiene las Media y Mediana del
## Tiempo y Monto por Sucursal y Nivel de Satisfaccion
## Se usa .agg()
data_banco_xlsx[['Tiempo_Servicio_seg', 'Monto', 'Sucursal',
                  'Satisfaccion']].groupby(['Sucursal', 'Satisfaccion']).agg(["mean",
                                                                              "median"]).reset_index()
```

	Sucursal	Satisfaccion	Tiempo_Servicio_seg	Monto
			mean	median
## 0	267	Muy Malo	178.288403	149.466417
## 1	267	Malo	176.860435	148.075930
## 2	267	Regular	175.813945	142.179657
## 3	267	Bueno	182.631086	145.554547
## 4	267	Muy Bueno	202.188738	159.871332
## 5	443	Muy Malo	150.047746	129.687657
## 6	443	Malo	162.763379	132.723704
## 7	443	Regular	192.666638	156.922623
## 8	443	Bueno	185.257154	145.259379
## 9	443	Muy Bueno	191.550564	150.487781
## 10	586	Muy Malo	80.352239	72.000000
## 11	586	Malo	77.351706	69.000000
## 12	586	Regular	80.347826	69.000000
## 13	586	Bueno	82.795337	70.000000

reset_index() para que los índices sean columnas

reset_index() es un método en Pandas que restablece el índice de un DataFrame o Serie, genera un nuevo índice numérico y el índice anterior se convertirá en una nueva columna en el DataFrame.

```
## Obtiene las Media y Mediana del
## Tiempo y Monto por Sucursal y Nivel de Satisfaccion
## Se usa .agg()
data_banco_xlsx[['Tiempo_Servicio_seg', 'Monto', 'Sucursal',
                 'Satisfaccion']].groupby(['Sucursal', 'Satisfaccion']).agg(["mean",
                                                                              "median"]).reset_index().info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 25 entries, 0 to 24
## Data columns (total 6 columns):
##  #      Column                                Non-Null Count  Dtype
## ---  -
##  0      (Sucursal, )                             25 non-null    object
##  1      (Satisfaccion, )                         25 non-null    category
##  2      (Tiempo_Servicio_seg, mean)               25 non-null    float64
##  3      (Tiempo_Servicio_seg, median)             25 non-null    float64
##  4      (Monto, mean)                             25 non-null    float64
##  5      (Monto, median)                           25 non-null    float64
## dtypes: category(1), float64(4), object(1)
## memory usage: 1.3+ KB
```

Cálculos con agrupamiento usando `.apply()`

Finalmente, se puede usar `.apply()` para los cálculos a realizar.

```
## Funcion que controla los calculos
def DESCR(x):
    return pd.Series({
        'Monto_Media': x.Monto.mean(),
        'Monto_Mediana': x.Monto.median(),
        'Tiempo_Media': x.Tiempo_Servicio_seg.mean(),
        'Tiempo_Mediana': x.Tiempo_Servicio_seg.median()})

## Probar la función
DESCR(data_banco_xlsx)
```

```
## Monto_Media      1996.156149
## Monto_Mediana    2087.430000
## Tiempo_Media     155.579993
## Tiempo_Mediana   122.452290
## dtype: float64
```

Cálculos con agrupamiento usando `.apply()`

Finalmente, se puede usar `.apply()` para los cálculos a realizar.

```
## Ya teniendo la funcion DESCR creada
## Usamos la función en .apply
data_banco_xlsx[['Tiempo_Servicio_seg', 'Monto', 'Sucursal']].groupby(
    'Sucursal').apply(DESCR)
```

##	Monto_Media	Monto_Mediana	Tiempo_Media	Tiempo_Mediana
## Sucursal				
## 267	2063.555879	2144.200	182.627140	148.095875
## 443	2078.908518	2139.390	181.011574	144.276091
## 586	1719.796059	1815.595	82.334563	71.000000
## 62	1758.289824	1851.145	89.392530	76.000000
## 85	2048.338975	2121.110	166.395467	135.590069

Tablas de frecuencia

Taller: Python (Pandas) para análisis de datos

Néstor Montaña P.



Frecuencia en variable numérica

En python para hacer una tabla de frecuencias se usa la función `.value_counts`

```
## Frecuencias del tiempo de servicio en segundos  
data_banco_xlsx['Tiempo_Servicio_seg'].value_counts(bins=7).reset_index(  
    name='Frecuencia')
```

##	index	Frecuencia
## 0	(16.546, 244.498]	20430
## 1	(244.498, 470.865]	3207
## 2	(470.865, 697.232]	516
## 3	(697.232, 923.598]	122
## 4	(923.598, 1149.965]	17
## 5	(1149.965, 1376.332]	6
## 6	(1376.332, 1602.698]	1



Frecuencia en variable numérica

En python para hacer una tabla de frecuencias se usa la función `.value_counts`

Para completar la tabla podemos usar las funciones que ya sabemos

```
## Frecuencias del tiempo de servicio en segundos
frec_Tiempo= data_banco_xlsx['Tiempo_Servicio_seg'].value_counts(
    bins=7).reset_index(name='Frecuencia')
frec_Tiempo['Frec_Acumulada'] = frec_Tiempo.Frecuencia.cumsum()
frec_Tiempo['Frec_Relativa'] = 100*( frec_Tiempo.Frecuencia /
    frec_Tiempo.Frecuencia.sum() ).round(4)
frec_Tiempo['Frec_Relativa_Acumulada'] = 100*( frec_Tiempo.Frec_Acumulada /
    frec_Tiempo.Frecuencia.sum() ).round(4)
frec_Tiempo
```

##	index	Frecuencia	...	Frec_Relativa	Frec_Relativa_Acumulada
## 0	(16.546, 244.498]	20430	...	84.08	84.08
## 1	(244.498, 470.865]	3207	...	13.20	97.28
## 2	(470.865, 697.232]	516	...	2.12	99.40
## 3	(697.232, 923.598]	122	...	0.50	99.90
## 4	(923.598, 1149.965]	17	...	0.07	99.97
## 5	(1149.965, 1376.332]	6	...	0.02	100.00
## 6	(1376.332, 1602.698]	1	...	0.00	100.00
##					
##	[7 rows x 5 columns]				

Frecuencia en variable numérica

Si queremos intervalos personalizados podemos definir los quiebres y hacer la tabla de frecuencia en dos pasos, primero crear una variable con los intervalos y luego obtener las frecuencias

```
limites = np.arange(0, 600, 60)
limites = np.append(limites, 1610)
data_banco_xlsx['Tiempo_intervalo'] = pd.cut(x=data_banco_xlsx[
    'Tiempo_Servicio_seg'], bins= limites )
data_banco_xlsx['Tiempo_intervalo'].value_counts( sort= False)
```

```
## (0, 60]          3861
## (60, 120]        8037
## (120, 180]       5201
## (180, 240]       3140
## (240, 300]       1661
## (300, 360]        933
## (360, 420]        550
## (420, 480]        296
## (480, 540]        223
## (540, 1610]      397
## Name: Tiempo_intervalo, dtype: int64
```



Frecuencia en variable numérica

En python para hacer una tabla de frecuencias se usa la función `.value_counts`
Para completar la tabla podemos usar las funciones que ya sabemos

```
## Frecuencias del tiempo de servicio en segundos
limites = np.arange(0, 600, 60)
limites = np.append(limites, 1610)
data_banco_xlsx['Tiempo_intervalo'] = pd.cut(x=data_banco_xlsx[
    'Tiempo_Servicio_seg'], bins= limites )
## Tabla de Frecuencias para el Tiempo de servicio en segundos
tbl_frec_tiempo= data_banco_xlsx['Tiempo_intervalo'].value_counts(sort= False).reset_index(name=
tbl_frec_tiempo['Frec_Acumulada'] = tbl_frec_tiempo.Frecuencia.cumsum()
tbl_frec_tiempo['Frec_Relativa'] = 100*( tbl_frec_tiempo.Frecuencia /
    tbl_frec_tiempo.Frecuencia.sum() ).round(4)
tbl_frec_tiempo['Frec_Relativa_Acumulada'] = 100*( tbl_frec_tiempo.Frec_Acumulada /
    tbl_frec_tiempo.Frecuencia.sum() ).round(4)
## Notar que \ al final de la linea le dice a Python que la linea continúa abajo
```

Frecuencia en variable numérica

En python para hacer una tabla de frecuencias se usa la función `.value_counts`
Para completar la tabla podemos usar las funciones que ya sabemos

```
tbl_frec_tiempo
```

```
##          index  Frecuencia  ...  Frec_Relativa  Frec_Relativa_Acumulada
## 0      (0, 60]      3861  ...           15.89           15.89
## 1      (60, 120]     8037  ...           33.08           48.96
## 2     (120, 180]     5201  ...           21.40           70.37
## 3     (180, 240]     3140  ...           12.92           83.29
## 4     (240, 300]     1661  ...            6.84           90.13
## 5     (300, 360]      933  ...            3.84           93.97
## 6     (360, 420]      550  ...            2.26           96.23
## 7     (420, 480]      296  ...            1.22           97.45
## 8     (480, 540]      223  ...            0.92           98.37
## 9    (540, 1610]      397  ...            1.63          100.00
##
## [10 rows x 5 columns]
```



Frecuencia en variable numérica

En python para hacer una tabla de frecuencias se usa la función `.value_counts`
Para completar la tabla podemos usar las funciones que ya sabemos

```
# sns.barplot( tbl_frec_tiempo, x= 'index', y='Frec_Relativa')
```

Frecuencia en variable categórica

Para las variables categóricas también se usa la función `.value_counts()` s

```
## Tabla de Frecuencias del Nivel de satisfaccion
frec_Satis= data_banco_xlsx['Satisfaccion'].value_counts().reset_index(
    name='Frecuencia')
frec_Satis['Frec_Relativa'] = 100*( frec_Satis.Frecuencia /
    frec_Satis.Frecuencia.sum() ).round(4)
frec_Satis
```

##		index	Frecuencia	Frec_Relativa
## 0	Muy	Bueno	6262	25.77
## 1		Bueno	5915	24.34
## 2		Regular	4639	19.09
## 3		Malo	4474	18.41
## 4	Muy	Malo	3009	12.38



Frecuencia en variable categórica

Para las variables categóricas también se usa la función `.value_counts()`

```
## Tabla de Frecuencias del Nivel de satisfaccion  
pd.crosstab(index= data_banco_xlsx['Satisfaccion'],  
            columns="Frecuencia")
```

```
## col_0      Frecuencia  
## Satisfaccion  
## Muy Malo      3009  
## Malo          4474  
## Regular       4639  
## Bueno         5915  
## Muy Bueno     6262
```


Tabla cruzada: 2 Var Categ

Tablas cruzadas entre Sucursales y Nivel de satisfaccion

```
## Tabla cruzada entre Sucursal
satisf_vs_suc = pd.crosstab(index= data_banco_xlsx['Sucursal'],
                             columns= data_banco_xlsx['Satisfaccion'])
satisf_vs_suc
```

```
## Satisfaccion    Muy Malo    Malo    Regular    Bueno    Muy Bueno
## Sucursal
## 267              539     848         642     707           593
## 443              461     673         823    1044          1189
## 586              335     381         345     386           451
## 62               431     604         520     684           599
## 85              1243    1968        2309    3094          3430
```

Tabla cruzada: 2 Var Categ

Tablas cruzadas entre Sucursales y Nivel de satisfaccion

```
## Tabla cruzada entre Sucursal
satisf_vs_suc = pd.crosstab(index= data_banco_xlsx['Sucursal'],
                             columns= data_banco_xlsx['Satisfaccion'],
                             margins= True)

satisf_vs_suc
```

## Satisfaccion	Muy Malo	Malo	Regular	Bueno	Muy Bueno	All
## Sucursal						
## 267	539	848	642	707	593	3329
## 443	461	673	823	1044	1189	4190
## 586	335	381	345	386	451	1898
## 62	431	604	520	684	599	2838
## 85	1243	1968	2309	3094	3430	12044
## All	3009	4474	4639	5915	6262	24299

Tabla cruzada: 2 Var Categ

Tablas cruzadas entre Sucursales y Nivel de satisfaccion

```
## Tabla cruzada entre Sucursal
nombresCol = np.append( satisf_vs_suc.columns[0:5], 'totalSatisf')
nombresFila = np.append( satisf_vs_suc.index[0:5], 'totalSucur')
satisf_vs_suc.columns = nombresCol
satisf_vs_suc.index= nombresFila
satisf_vs_suc
```

##	Muy Malo	Malo	Regular	Bueno	Muy Bueno	totalSatisf
## 267	539	848	642	707	593	3329
## 443	461	673	823	1044	1189	4190
## 586	335	381	345	386	451	1898
## 62	431	604	520	684	599	2838
## 85	1243	1968	2309	3094	3430	12044
## totalSucur	3009	4474	4639	5915	6262	24299



Tabla cruzada: 2 Var Categ

Tablas cruzadas entre Sucursales y Nivel de satisfaccion, **en porcentaje**

```
## Total
satisf_vs_suc.loc["totalSucur", "totalSatisf"]
```

```
## 24299
```

```
## Tabla cruzada entre Sucursal
(satisf_vs_suc/satisf_vs_suc.loc["totalSucur", "totalSatisf"]).round(4) * 100
```

##	Muy Malo	Malo	Regular	Bueno	Muy Bueno	totalSatisf
## 267	2.22	3.49	2.64	2.91	2.44	13.70
## 443	1.90	2.77	3.39	4.30	4.89	17.24
## 586	1.38	1.57	1.42	1.59	1.86	7.81
## 62	1.77	2.49	2.14	2.81	2.47	11.68
## 85	5.12	8.10	9.50	12.73	14.12	49.57
## totalSucur	12.38	18.41	19.09	24.34	25.77	100.00

Tabla cruzada: 2 Var Categ - Ahora condicionando por filas

Tablas cruzadas entre Sucursales y Nivel de satisfaccion

```
## Tabla cruzada entre Sucursal
satisf_vs_suc.div( satisf_vs_suc.loc[:, "totalSatisf"], axis=0).round(4) * 100
```

##	Muy Malo	Malo	Regular	Bueno	Muy Bueno	totalSatisf
## 267	16.19	25.47	19.29	21.24	17.81	100.0
## 443	11.00	16.06	19.64	24.92	28.38	100.0
## 586	17.65	20.07	18.18	20.34	23.76	100.0
## 62	15.19	21.28	18.32	24.10	21.11	100.0
## 85	10.32	16.34	19.17	25.69	28.48	100.0
## totalSucur	12.38	18.41	19.09	24.34	25.77	100.0

Tabla cruzada: 2 Var Categ - condicionando por Columnas

Tablas cruzadas entre Sucursales y Nivel de satisfaccion

```
## Tabla cruzada entre Sucursal
satisf_vs_suc.div( satisf_vs_suc.loc["totalSucur",:], axis=1).round(4) * 100
```

##	Muy Malo	Malo	Regular	Bueno	Muy Bueno	totalSatisf
## 267	17.91	18.95	13.84	11.95	9.47	13.70
## 443	15.32	15.04	17.74	17.65	18.99	17.24
## 586	11.13	8.52	7.44	6.53	7.20	7.81
## 62	14.32	13.50	11.21	11.56	9.57	11.68
## 85	41.31	43.99	49.77	52.31	54.77	49.57
## totalSucur	100.00	100.00	100.00	100.00	100.00	100.00



Exportar a Excel

Tablas cruzadas entre Sucursales y Nivel de satisfaccion

```
## Tabla cruzada entre Sucursal  
rep = satisf_vs_suc.div( satisf_vs_suc.loc["totalSucur",:], axis=1).round(4) * 100  
rep.to_excel("reporte_tabl_cruzada.xlsx")
```

FIN

Taller: Python (Pandas) para análisis de datos

Néstor Montaña P.