{desafío} latam_

Probabilidades y funciones

Alcance de la lectura:

- Utilizar funciones para reutilizar código. (Principio D.R.Y)
- Convertir una formula matemática a una función en python
- Construir y utilizar funciones orientadas al análisis de datos.
- Optimizar funciones remplazandolas por funciones vectorizadas.
- Utilizar conceptos básicos de probabilidad.
- Generar segmentaciones de un `pd.DataFrame` en base a indexación y selección.

Probabilidad Básica

En esta sesión trabajaremos con una base de datos que registró todas las batallas de la Guerra de los Cinco Reyes, parte de la saga *A Song of Fire and Ice* de George R.R. Martin. La base de datos la creó Chris Albon, un científico de datos con background en ciencias sociales.

La base de datos y el libro de códigos asociado se encuentran en el siguiente repositorio de Github https://github.com/chrisalbon/war_of_the_five_kings_dataset

Comencemos por incluir los módulos con los cuales trabajamos la semana pasada

```
import pandas as pd
import numpy as np
import lec2_graphs as gfx
import matplotlib.pyplot as plt
import scipy.stats as stats

plt.style.use('seaborn-whitegrid') # Gráficos estilo seaborn
plt.rcParams["figure.figsize"] = (4, 6) # Tamaño gráficos
plt.rcParams["figure.dpi"] = 100 # resolución gráficos
```

```
df = pd.read_csv('got_battles.csv')
```

Revisemos las primeras observaciones con `head`.

Cada observación ingresada en las filas representa una batalla dentro de la Guerra de los Cinco Reyes.

Las columnas registran información sobre los participantes de la batalla, el tipo de batalla, resultados relevantes como la tasa de victorias del atacante, el tipo de batalla y muertes relevantes.

```
df.columns
```

```
Index(['me', 'year', 'battle_number', 'attacker_king', 'defender_king',
       'attacker_1', 'attacker_2', 'attacker_3', 'attacker_4', 'defender_1',
       'defender_2', 'defender_3', 'defender_4', 'attacker_outcome',
       'battle_type', 'major_death', 'major_capture', 'attacker_size',
       'defender_size', 'attacker_commander', 'defender_commander', 'summer',
       'location', 'region', 'note'],
      dtype='object')
```

`shape` nos dice que tenemos una tabla de datos con 38 observaciones y 25 columnas.

```
df.shape
```

```
(38, 25)
```

Analicemos los principales iniciadores de batallas. Para eso utilizamos la función `value_counts()` de `pandas` que devuelte una lista de la cantidad de objetos únicos en una Serie.

> Recuerden que cuando hablamos de Series, nos referimos a una lista unidimensional.

Observamos que los Baratheon concentran 14 de 38 batallas, seguidos por los Starks. Stannis Baratheon, el "tío" de Joffrey, se posición como el que generó menos batallas con 5 de 35.

```
df['attacker_king'].value_counts()
```

```
Joffrey/Tommen Baratheon    14
Robb Stark                  10
Balon/Euron Greyjoy          7
Stannis Baratheon            5
Name: attacker_king, dtype: int64
```

Estas cifras se conocen como frecuencias, que son la cantidad de eventos sobre el total muestral.

Para obtener una panorámica más clara sobre los atacantes, podemos dividir cada cifra por el total de casos:

$$\Pr(\text{Evento}) = \frac{\text{Número de elementos del evento}}{\text{Posibles resultados en } \Omega}$$

```
df['attacker_king'].value_counts() / len(df)
```

```
Joffrey/Tommen Baratheon    0.368421
Robb Stark                  0.263158
Balon/Euron Greyjoy         0.184211
Stannis Baratheon           0.131579
Name: attacker_king, dtype: float64
```

Para obtener las frecuencias en términos porcentuales, podemos aplicar una división a nuestra operación con `value_counts`. Esto se conoce con **operación vectorizada** y es un principio imporante en la programación orientada al análisis de datos.

La operación vectorizada se puede resumir en la máxima: "*procurar utilizar operaciones en listas por sobre operaciones con iteradores*".

Esto suena complejo, pero simplemente es un principio rector para utilizar funciones que acepten vectores (listas como las `Series`) por sobre procesar cada elemento de una lista.

Se observa que entre Joffrey y Tomen Baratheon se concentra el 37% de las batallas, seguidas por un 26% de Robb Stark.

Definición clásica de probabilidad

Estos porcentajes nos permite hablar de juicios de probabilidad. Acorde a los registros de las batallas, la probabilidad que una sea iniciada por un Baratheon (sumando Joffrey/Tommen y Stannis) es del 50%.

Cuando realizamos una declaración probabilística como ésta, *generamos juicios sobre experimentos* ***aleatorios*** *que producen una serie de resultados*.

Cuando decimos que la probabilidad que un Baratheon inicie una batalla es del 50%, generamos un modelo de probabilidad con dos elementos:

- Un *espacio muestral* (Ω) que hace referencia a todos los posibles resultados. En este caso, las 38 observaciones son el espacio muestral.
- Una *función de probabilidad* donde un elemento A se le asigna un número no-negativo $\Pr(A)$ que representa la verosimilitud de ocurrencia. En este caso, las 19 observaciones son el elemento A que se convierten en $\Pr(\mathrm{AtaqueBaratheon}) = 19/38 \rightarrow .50$.

Digresión: Axiomas de Probabilidad

Toda probabilidad debe seguir conjunto de reglas que definen su comportamiento. Dado un espacio muestral Ω para un experimento específico, la función de probabilidad asociada a éste debe satisfacer tres axiomas:

1. **Nonegatividad:** $\Pr(A) \geq 0 \forall A \subset \Omega$. Toda probabilidad dentro del espacio muestral será mayor a 0.
2. **Aditividad:** Para eventos mutuamente excluyentes $a_1, a_2, \ldots$

$$\Pr\left(\bigcup_{i=1}^{\infty} a_i\right) = \sum_{i=1}^{\infty} \Pr(E_i)$$

3. **Normalización:** $\Pr(\Omega) = 1$. La probabilidad del espacio muestral será 1.
4. En base a estos tres principios, se pueden derivar una serie de reglas generales sobre la probabilidad.

Casos Independientes

Por casos independientes hacemos referencia a eventos aislados que no están condicionados por otro.

Un ejemplo de ésto es la probabilidad que se produzca una muerte importante (medida con la variable `major_death`)

```
df['major_death'].value_counts() / len(df)
```

```
0.0    0.631579
1.0    0.342105
Name: major_death, dtype: float64
```

La variable está codificada como *dummy*, y toma dos valores: 1 para aquellas batallas que presentaron la muerte de un personaje importante; 0 para aquellas batallas que no.

La información solicitada es que hay un 34% de probabilidad que en una batalla cualquiera muera un personaje importante.

Decimos que es un evento independiente dado que si observamos dos batallas, el resultado de la primera no nos da información sobre la probabilidad de muerte de la segunda batalla.

Casos Dependientes

De forma lógica, un caso dependiente se refiere donde la ocurrencia de un evento depende de la ocurrencia de otro.

Siguiendo con el ejemplo de la probabilidad de muerte, un evento dependiente sería la probabilidad que el personaje haya estado enfermo antes o durante la batalla.

El estar enfermo tiene una repercusión en la probabilidad de muerte, dado que afecta a su nivel de energía y preparación en la batalla.

Intersección de Eventos

Podemos calcular la probabilidad de dos o más eventos en Ω. Para ello hay que considerar dos casos:

Caso donde buscamos ver la probabilidad que **por lo menos uno de dos eventos ocurra**. Este caso se puede operacionalizar considerando si los eventos son independientes o dependientes.

- Para un caso **independiente** la probabilidad de sucesos no compatibles viene dada por:

$$\Pr(A \cup B) = \Pr(A) + \Pr(B)$$

- Para un caso **dependiente** la probabilidad de sucesos compatibles viene dada por:

$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B)$$

Donde ajustamos la fórmula de eventos independientes por la probabilidad de *intersección* de los eventos A y B. De esta manera evitamos contar dos veces ciertos eventos con características en conjunto.

Caso donde buscamos ver la probabilidad que **ambos eventos ocurran**. En este caso se puede operacionalizar considerando si los eventos son independientes o dependientes.

- Para un caso **independiente** la probabilidad de sucesos no compatibles viene dada por:

$$\Pr(A \cap B) = \Pr(A) \times \Pr(B)$$

- Para un caso **dependiente** la probabilidad de sucesos compatibles viene dada por:

$$\Pr(A \cap B) = \Pr(A/B) \times \Pr(B)$$

Contando unión e intersección de eventos

Existen varias formas de evaluar unión e intersección de eventos en Python. En esta lectura enseñaremos las dos variantes más utilizadas con la suite de Ciencia de Datos.

Si estamos interesados en evaluar un caso de intersección donde ambas características se deben encontrar en una observación, nuestra primera implementación con `iterrows` involucraría generar un contador y actualizarlo cuando ambas condiciones sean verdaderas.

```
counter = 0
for rowname, rowserie in df.iterrows():
    if rowserie['major_death'] == 1.0 and rowserie['major_capture'] == 1.0:
        counter += 1
print(counter)
```

```
6
```

Una opción implementando `numpy` sería utilizar la función `np.logical_and` que genera una forma vectorizada de evaluar la existencia de ambas condiciones en cada par de registro.

```
counter = np.logical_and(df['major_death'] == 1.0,
                         df['major_capture'] == 1.0)
np.unique(counter, return_counts=True)
```

```
(array([False,  True]), array([32,  6]))
```

Para generar el operador de intersección resulta ser or, el cual tiene su análogo de `numpy` como `np.logical_or`.

```
counter = 0

for rowname, rowserie in df.iterrows():
    if rowserie['major_death'] == 1.0 or rowserie['major_capture'] == 1.0:
        counter += 1
print(counter)
```

```
18
```

```
counter = np.logical_or(df['major_death'] == 1.0,
                        df['major_capture'] == 1.0)

np.unique(counter, return_counts=True)
```

```
(array([False,  True]), array([20, 18]))
```

Probabilidad Condicional

Imaginen que un fan de George R.R Martin les pregunta por la probabilidad que muera un personaje importante cuando la batalla fue iniciada por un Baratheon. Para ello nos basamos en la **probabilidad condicional**. Ésta busca responder la verosimilitud de un evento A sabiendo que sucede en B.

Para obtener esta verosimilitud, debemos buscar la probabilidad $\Pr(A|B)$.

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)}$$

Comencemos por definir los componentes del problema:

- Evento A ($\Pr(A)$): Probabilidad que la batalla haya presenciado una muerte importante.
- Evento B ($\Pr(B)$): Probabilidad que la batalla se haya originado por Joffrey o Tommen Baratheon.
- Evento A **y** B ($\Pr(A \cap B)$): Probabilidad que se haya producido una muerte importante **en** un batalla la originó Joffrey o Tommen Baratheon.

- El evento $\Pr(A \cap B)$ se conoce como *intersección*, porque sólo cuenta aquellos casos donde las condiciones A y B están presentes.

También existe el operador $\Pr(A \cup B)$, que se conoce como *unión*, que busca contar todos los casos donde **por lo menos una de las condiciones A o B esté presente**.

Estimemos la probabilidad de ocurrencia de la muerte en batallas originadas por Baratheon ($\Pr(A|B)$). Reemplazando valores en la ecuación de arriba, nuestro problema se reformula como.

$$\Pr(\text{Muerte} \mid \text{Baratheon}) = \frac{\Pr(\text{Muerte y Baratheon})}{\Pr(\text{Baratheon})}$$

```
# iniciemos un contador
muerte_baratheon = 0
```

Para iterar en `DataFrames`, `pandas` presenta el método `iterrows` que define un iterador para `Indices` de nuestra tabla (señalados con `i` en el loop) y `Series` (señalados con `r` en el loop).

```
muerte_baratheon = 0
# por cada fila en nuestra tabla
for i, r in df.iterrows():
    # si la batalla la inicio un Baratheon y hubo una muerte importante
    if (r['attacker_king'] == 'Joffrey/Tommen Baratheon' and r['major_death'] == 1):
        # agregar uno a nuestro contador
        muerte_baratheon += 1
print(muerte_baratheon)
```

```
5
```

Ahora podemos solicitar la información de nuestro contador. En total, hay 5 eventos que satisfacen esta condición, que representan el 13% de la muestra.

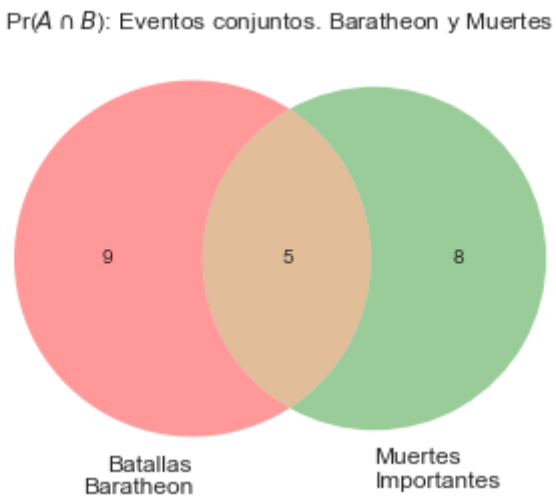
La probabilidad de observar este evento es del 13%

```
print("Cantidad de muertes importantes en batallas Baratheon: ", muerte_baratheon)
print("Pr(Muerte | Baratheon): ", muerte_baratheon / len(df))
```

```
Cantidad de muertes importantes en batallas Baratheon:  5
Pr(Muerte | Baratheon):  0.13157894736842105
```

La intersección de cantidad de muertes importantes en batallas Baratheon también se puede representar de forma gráfica con un diagrama de Venn:

```
gfx.graph_venn()
```




```
print("Cantidad de muertes importantes en batallas Baratheon: ", muerte_baratheon)
print("Pr(Muerte | Baratheon): ", muerte_baratheon / len(df))
```

```
Cantidad de muertes importantes en batallas Baratheon:  5
Pr(Muerte | Baratheon):  0.13157894736842105
```

Podríamos estar tentados a interpretar este 13% como la probabilidad condicional del evento, pero hay que considerar que ésta cifra se calcula en comparación a la muestra, no con la muestra de las batallas iniciadas por Baratheons. Por eso es que ajustamos (i.e: dividimos) por las batallas Baratheon.

```
batallas_baratheon = df['attacker_king'].value_counts().get('Joffrey/Tommen Baratheon')
print("Pr(Muerte|Baratheon):", muerte_baratheon / batallas_baratheon)
```

```
Pr(Muerte|Baratheon): 0.35714285714285715
```

Así, observamos que *dentro de todas las batallas iniciadas por un Baratheon, la probabilidad que un personaje importante muera es del 35%*.

Funciones

Las funciones nos permiten abstraernos del código para resolver un problema. Una vez que tenemos definida un función podemos simplemente llamarla cuando la necesitemos. Las definiremos y/o ocuparemos funciones definidas por otros todo el tiempo, es muy importante que aprendamos bien su uso y sintaxis.

Por ejemplo podemos crear una función para calcular la hipotenusa.

```
import math

def hipotenusa(x, y):
    tmp = (x ** 2 + y ** 2)
    tmp = math.sqrt(tmp)
    return(tmp)
```

y luego podemos calcular la hipotenusa de cualquier conjunto de catetos utilizando:

```
hipotenusa(3,4)
```

```
5.0
```

Este principio de abstracción es muy importante para crear código ordenado y que podamos reutilizar cada vez que tengamos una situación similar.

El principio DRY

Uno de los principios básicos de la programación es el **DRY: D**on't **R**epeat **Y**ourself. Si vamos a utilizar una expresión más de 2 veces, es mejor generar una función a partir de ella. No solo es una práctica inteligente, también evita problemas en el código al compartir definiciones y procedimientos

Dada la amable naturaleza sintáctica de Python, generarlas es muy fácil.

Generemos nuestra primera función. Lo que deseamos es que la función nos salude cuando la invoquemos.

Creando nuestra primera función

```
def saludar():
    print('Hola Mundo!')

saludar()
```

```
Hola Mundo!
```

Anatomía mínima de una función

La definición de una función se compone de los siguientes elementos:

- Declaración `def`: Mediante ella señalamos que todo lo escrito será considerado como una función.
- Nombre de la función `saludar()`: Este es el identificador que utilizaremos para llamarla posteriormente. Cabe destacar que los paréntesis y el doble punto al final de la función son obligatorias.
- La indentación (sangrado) es parte importante, le permite a python saber que parte del código escrito pertenece a la función.
- Expresiones a ejecutar `print('Hola Mundo!')` estas expresiones tienne que estar correctamente indentadas, el estándar es de **4 espacios**.

¿Qué sucede si no indentamos?

Hay dos situaciones posibles, la primera es que la función no tenga ningún bloque identado, esto mostrará un error.

```
def saludar():
print("esto no se imprimirá porque hay un error")

saludar()
```

```
  File "<ipython-input-21-9e079350a778>", line 2
    print("esto no se imprimirá porque hay un error")
        ^
IndentationError: expected an indented block
```

La segunda opción es que parte de la función quede fuera de la función.

```
def saludar():
    print("este código si pertenece a la función")
    print("este código si pertenece a la función")
print("este código no pertenece a la función")
saludar()
saludar()
saludar()
```

```
este código no pertenece a la función
este código si pertenece a la función
este código si pertenece a la función
este código si pertenece a la función
este código si pertenece a la función
este código si pertenece a la función
este código si pertenece a la función
```

Parámetros

Para que las funciones sean flexibles pueden recibir parámetros.

Por ejemplo podríamos hacer mucho mas flexible la función de saludar si le pasamos como parámetro el saludo.

```
def saludar(saludo):
    print(saludo)

saludar("hola")
saludar("hola mundo")
```

```
hola
hola mundo
```

Una función puede recibir múltiples parámetros separados por una `,` cada uno.

```
def saludar(saludo1, saludo2):
    print(saludo1)
    print(saludo2)

saludar("hola", "mundo")
```

```
hola
mundo
```

Los parámetros son obligatorios

Si una función recibe parámetros y nos los especificamos obtendremos un error.

```
def saludar(saludo1, saludo2):
    print(saludo1)
    print(saludo2)

saludar("hola")
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-25-a9e63139a8a4> in <module>
      3     print(saludo2)
      4
----> 5 saludar("hola")
```

```
TypeError: saludar() missing 1 required positional argument: 'saludo2'
```

Parámetros con valores por defecto

En algunos casos es muy útil que una función reciba un valor por defecto.

```
def saludar(saludo1, saludo2="valor por defecto"):
    print(saludo1, saludo2)

saludar("hola")
saludar("hola", "valor ingresado manualmente")
```

```
hola valor por defecto
hola valor ingresado manualmente
```

El retorno

Las funciones pueden devolver valores con los que podemos seguir trabajando, veamos un ejemplo muy sencillo.

```
def incrementar(a, b = 1):
    return a + b


print(incrementar(2))
print(incrementar(5, 4))
```

```
3
9
```

Las funciones normalmente devuelven un valor o lo muestran en pantalla. Son muy poco frecuentes los casos distintos.

De la fórmula a Python

Es importante que aprendamos a convertir una ecuación matemática a una función de python. Para empezar recalculemos la hipotenusa.

$$\text{Hipotenusa} = \sqrt{x^2 + y^2}$$

Escribir el código para resolver el problema es sencillo, elevamos dos números al cuadrado y luego sacamos la raíz de la suma. Por ejemplo para los catetos de tamaño 3 y 4 lo calcularíamos como:

```
np.sqrt((3 **2 + 4 ** 2))
```

```
5.0
```

Para poder reutilizar el código necesitamos entender, cualos son los valores de ingreso, y los de salida. Si queremos calcular la hipotenusa a partir de los catetos entonces los catetos son la entrada y la hipotenusa la salida.

```
def hypotenuse(x, y):
    return np.sqrt((x **2 + y ** 2))
print(hypotenuse(3, 4))
```

```
5.0
```

Aplicando funciones

Ahora vamos a generar una serie de funciones para sistematizar nuestro análisis de las batallas.

Consideremos los siguientes objetivos:

- Deseamos una función que nos entregue las medidas de dispersión para las variables contínuas, y que entregue una excepción en las strings.
- Generar una función para calcular las probabilidades condicionales en base a dos atributos.

Función descriptiva

Primer Intento: llamando funciones

Nuestro primer intento para esta función es incluír dentro del cuerpo de la función todas las llamadas a las funciones que aprendimos en la primera semana: `np.mean` para estimar la media, `stats.mode` para la moda y `np.std` para la desviación.

Con estos valores podemos tener una visión general del cómo se comporta la variable en cuanto a tendencia y dispersión.

```
def fetch_descriptives(x):
    np.mean(x)
    stats.mode(x)
    np.std(x)

fetch_descriptives(df['attacker_size'])
```

Si llamamos a `fetch_descriptives`, no obtendremos respuesta alguna dado que olvidamos incluír `print` o `return`. Refactorizemos en un segundo intento.

Una salvedad, esto no implica que el cuerpo de la función no se haya ejecutado.

Segundo intento: incluir prints

Nuestra función falla en reportar cualquier cálculo. Para ello utilizaremos `prints` en cada llamada para devolver el valor de cada función.

```
def fetch_descriptives(x):
    print(np.mean(x))
    print(stats.mode(x))
    print(np.std(x))

fetch_descriptives(df['attacker_size'])
```

```
9942.541666666666
ModeResult(mode=array([6000.]), count=array([3]))
19856.031772442948
```

Ahora si obtenemos resultados sobre la variable `attacker_size`. Un percance es que los datos se retornan en strings. ¿Qué pasa si deseamos que la función retorne valores asignables a objetos?

Tercer intento: generando objetos dentro de la función

Para resolver el problema descrito, podemos asignar un objeto a cada función ejecutada en el cuerpo de la función. De esta forma obtenemos resultados más flexibles.

Para que nuestra función retorne las variables, es necesario utilizar `return`.

```
def fetch_descriptives(x):
    mu = np.mean(x)
    mode = stats.mode(x)
    sigma = np.std(x)
    return mu, mode, sigma
```

```
fetch_descriptives(df['attacker_size'])
```

```
(9942.541666666666,
 ModeResult(mode=array([6000.]), count=array([3])),
 19856.031772442948)
```

Si ejecutamos la función y no asignamos su resultado a una variable, obtendremos exactamente lo mismo.

```
resultado = fetch_descriptives(df['attacker_size'])
resultado
```

```
(9942.541666666666,
 ModeResult(mode=array([6000.]), count=array([3])),
 19856.031772442948)
```

Pero si asignamos la misma cantidad de variables que de resultados de la función, obtendremos algo mucho más trabajable

```
media, moda, varianza = fetch_descriptives(df['attacker_size'])
print(media)
print(moda)
print(varianza)
```

```
9942.541666666666
ModeResult(mode=array([6000.]), count=array([3]))
19856.031772442948
```

Cuarto intento:

Ya estamos un poco más encaminados con nuestra función, pero hay una serie de detalles:

1. Los números se presentan en formatos incomprensibles: Para ello redondearemos la media y la varianza, y selecionaremos el valor correspondiente a la moda.
2. Nuestra función todavía no genera el mensaje para los strings.

Reordenando números

```
def fetch_descriptives(x):
    # utilizamos round para redondear al tercer decimal
    mu = round(np.mean(x), 3)
    # solicitamos primer elemento (6000) dentro del primer resultado de la moda
    mode = stats.mode(x)[0][0]
    # utilizamos round para redondear al tercer decimal
    sigma = round(np.std(x), 3)
    return mu, mode, sigma

media, moda, varianza = fetch_descriptives(df['attacker_size'])
print(media)
print(moda)
print(varianza)
```

```
9942.542
6000.0
19856.032
```

Primera Validación: Diferenciando entre series y no series

Resulta que el comportamiento de nuestra función debe ser sensible al tipo de dato ingresado. Mediante las validaciones de los parámetros ingresados en nuestra función, nos aseguramos de definir un caso de uso específico y qué debe hacer la función cuando no se cumplan las condiciones.

Para diferenciar si nuestros datos ingresados corresponden al tipo `pd.Series`, utilizaremos el método `isinstance` para preguntar si el objeto ingresado pertenece a una clase específica. Su implementación es la siguiente, y retornará un booleano dependiendo si satisface la condición o no.

```
isinstance(df['attacker_size'], pd.Series)
```

```
True
```

El método mencionado arriba se encapsulará en un `if` para discriminar entre datos numéricos y no numéricos.

De manera adicional, generaremos mensaje que definirá el tipo de error mediante `raise ValueError`.

```
def fetch_descriptives(x):
    if isinstance(x, pd.Series) is True:
        mu = round(np.mean(x), 3)
        median = np.median(x)
        mode = stats.mode(x)[0][0]
        sigma = round(np.std(x), 3)

    else:
        raise ValueError('El tipo de dato ingresado no es un objeto pd.Series')
    return mu, median, mode, sigma
```

```
fetch_descriptives('Lorem Ipsum Dolor Sit Amet')
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-41-3480f1348e8a> in <module>
----> 1 fetch_descriptives('Lorem Ipsum Dolor Sit Amet')
```

```
<ipython-input-40-22bde6fd751b> in fetch_descriptives(x)
      7
      8     else:
----> 9         raise ValueError('El tipo de dato ingresado no es un objeto pd.Series')
     10     return mu, median, mode, sigma
```

```
ValueError: El tipo de dato ingresado no es un objeto pd.Series
```

Segunda Validación: diferenciando entre datos numéricos y no numéricos

Resulta que nuestra función puede identificar correctamente el tipo de objeto y qué hacer dentro del scope.

Dado que las medidas de estadística univariada funcionan sólo con datos numéricos, el siguiente punto es la diferenciación entre objetos `pd.Series` que contengan números u otro valor.

Implementaremos el método `.dtype` que existen en los `pd.Series` para evaluar si el dato es un objeto (un dato no numérico según `pandas`) o si es numérico. La implementación de esta parte queda definida en la siguiente línea de código

```
# este es un tipo genérico, dado que son cadenas alojadas en las celdas
df['attacker_king'].dtype
```

```
dtype('O')
```

```
# este es un tipo numérico

df['attacker_size'].dtype
```

```
dtype('float64')
```

La implementación de la función quedaría así. Cabe destacar que de manera adicional al `if` que evalúe esta condición, implementaremos otro `raise ValueError`.

```
def fetch_descriptives(x):
    if isinstance(x, pd.Series) is True:
        if x.dtype != 'object':
            x = x.dropna()
            mu = round(np.mean(x), 3)
            median = np.median(x)
            mode = stats.mode(x)[0][0]
            sigma = round(np.std(x), 3)
        else:
            raise ValueError('El objeto pd.Series no presenta valores numéricos.')

    else:
        raise ValueError('El tipo de dato ingresado no es un objeto pd.Series')
    return mu, median, mode, sigma
```

```
fetch_descriptives(df['attacker_king'])
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-55-47c7449ed58e> in <module>
----> 1 fetch_descriptives(df['attacker_king'])
```

```
<ipython-input-54-2e2161783ab5> in fetch_descriptives(x)
      8             sigma = round(np.std(x), 3)
      9         else:
---> 10             raise ValueError('El objeto pd.Series no presenta valores numéricos.')
     11
     12     else:
```

```
ValueError: El objeto pd.Series no presenta valores numéricos.
```

```
mu, median, mode, var = fetch_descriptives(df['attacker_size'])

print(mu, median, mode, var)
```

```
9942.542 4000.0 6000.0 19856.032
```

Función para estimar probabilidades condicionales

Nuestro segundo objetivo es el refactorizar el procedimiento para obtener probabilidades condicionales. Sabemos que para obtener la probabilidad condicional de A en B necesitamos dos cifras:

- $\Pr(A \cup B)$: La probabilidad de ambos eventos sucedan en el espacio muestral. Eso lo obteníamos con el siguiente flujo:

```
muerte_baratheon = 0
for i, r in df.iterrows():
    if (r['attacker_king'] == 'Joffrey/Tommen Baratheon' and r['major_death'] == 1):
    muerte_baratheon += 1
```

- $\Pr(B)$: La probabilidad de ocurrencia específica en B:

```
batallas_baratheon = df['attacker_king'].value_counts()[0]
```

- $\Pr(A|B) = \frac{\Pr(A \cup B)}{\Pr(B)}$: La razón entre ambos eventos:

```
print("Pr(Muerte|Baratheon):", muerte_baratheon / batallas_baratheon)
```

Para generar nuestra función debemos establecer todos los parámetros necesarios. Cabe destacar que *todos los eventos son ocurrencias* ***específicas*** *dentro de las variables*. De esta manera, nuestra función necesita de cinco parámetros:

1. El evento específico A. `A_var`
2. La variable donde A pertenece. `A_condition`
3. El evento específico B. `B_var`
4. La variable donde B pertenece. `B_condition`
5. La tabla de datos donde todos los datos se encuentran localizados. `df`

```
def conditional_pr(df, A_var, A_condition, B_var, B_condition):

    get_a_and_b = 0

    for i, r in df.iterrows():
        if (r[A_var] == A_condition and r[B_var] == B_condition):
            get_a_and_b += 1

    get_b_pr = df[B_var].value_counts().loc[B_condition]

    print("Pr(", A_condition, "|", B_condition, ")", (get_a_and_b / get_b_pr))
```

La prueba de fuego de la función es que estime la misma probabilidad que en nuestro código a mano:

```
refactor_pr = conditional_pr(df, A_condition=1,
                A_var='major_death',
                B_condition='Joffrey/Tommen Baratheon',
                B_var='attacker_king')
original_pr = muerte_baratheon / batallas_baratheon
print("Hardcoded: ",original_pr)
```

```
Pr( 1 | Joffrey/Tommen Baratheon ) 0.35714285714285715
Hardcoded:  0.35714285714285715
```

Una buena práctica es el agregar `docstring` a nuestras funciones. Esto permite insertar documentación dentro de la misma para explicar y definir los parámetros y su uso.

Los `docstring` se incluyen *después de la declaración de función y antes del cuerpo de la función*. Se deliminan mediante `"""` y se cierran con otros `"""`

```
def conditional_pr(df, A_var, A_condition, B_var, B_condition):
    """
    conditional_pr: retrieve conditional probability given the following formula:

                        Pr(A and B)
                Pr(A|B) = ----------
                            Pr(B)

    parameters:
        `df`: dataframe.
        `A_var`: event to be conditioned by B.
        `A_condition`: specific outcome on A.
        `B_var`: conditioning event on A.
        `B_condition`: specific conditioning event on B.

    """

    get_a_and_b = 0
```

```
    for i, r in df.iterrows():
        if (r[A_var] == A_condition and r[B_var] == B_condition):
            get_a_and_b += 1

    get_b_pr = df[B_var].value_counts().loc[B_condition]

    print("Pr(", A_condition, "|", B_condition, ")", (get_a_and_b / get_b_pr))
```

```
print(conditional_pr.__doc__)
```

```
    conditional_pr: retrieve conditional probability given the following formula:

                          Pr(A and B)
                Pr(A|B) = ----------
                             Pr(B)

    parameters:
        `df`: dataframe.
        `A_var`: event to be conditioned by B.
        `A_condition`: specific outcome on A.
        `B_var`: conditioning event on A.
        `B_condition`: specific conditioning event on B.
```

Esto es análogo a nuestro método `Shift + Tab` con Jupyter

```
refactor_pr = conditional_pr(df, A_condition=1,
                A_var='major_death',
                B_condition='Joffrey/Tommen Baratheon',
                B_var='attacker_king')
```

```
Pr( 1 | Joffrey/Tommen Baratheon ) 0.35714285714285715
```