Estadística univariada y control de flujo

Alcance de la lectura

- Conocer los principales modos de trabajo con Jupyter notebook.
- Importar archivos de datos utilizando pandas.
- Utilizar las estructuras de datos de `pd.Series` y `pd.DataFrame`.
- Analizar datos de forma univariada con pandas.
- Hacer uso de control de flujos para obtener medidas estadísticas.

pandas: Python for Data Analysis

`pandas` (acrónimo para Panel Data) es un módulo orientado a la manipulación y limpieza de estructuras de datos.

Se suele utilizar en conjunto a otros módulos como `numpy`, `scipy` y `matplotlib` para el análisis de datos, que veremos con detalle después.

¿Por qué pandas?

1. Es una colección de funciones y algoritmos que implementan las principales convenciones sobre el análisis de datos.

2. Permite importar distintos archivos de datos con procedimientos robustos que permiten centrar al investigador en lo substancial y no en procesar archivos.

3. Genera un objeto **DataFrame** con una estructura de matriz (Filas y Columnas) que resulta intuitiva para desarrollar el análisis orientado en variables y segmentado por casos.

4. Presenta una amplia gama de funciones y procedimientos comunes generadas en los objetos.

Importando archivos con pandas

Usualmente trabajaremos con datos en un formato de texto plano, llámese `csv` o `xlsx`.

Ahora generamos nuestra primera interacción con `pandas`:

- La convención indica que pandas se abrevia de forma pd.

```
import pandas as pd
```

- Dentro de nuestro directorio de trabajo encontraremos el archivo `alumnos.csv`.
- Un archivo `csv` (comma separated value) presenta el nombre de los atributos en la primera fila. Después de ella, le siguen las observaciones ingresadas.
- Cada observación está separada mediante comas (de ahí el nombre *archivo separado por comas*).

```
# solicitemos las primeras filas del archivo alumnos (desde bash)
!head alumnos.csv
```

```
nombre,altura,peso,edad,sexo
Hugo,1.67,60,23,h
Paco,1.73,83,25,h
Luis,1.62,70,28,h
Diana,1.58,58,21,m
Francisco,1.86,98,28,h
Felipe,1.79,100,26,h
Jacinta,1.69,62,20,m
Bernardo,1.6,83,31,h
Marisol,1.6,56,30,m
```

Para ingresar este archivo a nuestro notebook, utilizaremos la función `read_csv` de `pandas`.

Digresión: ¿Cómo llamar funciones dentro de un módulo?

Para llamar una función y/u objeto específico dentro de nuestro módulo importado, utilizamos la siguiente sintaxis:

```
modulo.funcion
```

Vamos a generar un objeto llamado `df` (contracción de dataframe) mediante `read_csv`.

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

Si todo resulta bien, podemos ver las primeras observaciones de nuestro nuevo objeto con `df.head()`.

```
df.head()
```

	nombre	altura	peso	edad	sexo
0	Hugo	1.67	60	23	h
1	Paco	1.73	83	25	h
2	Luis	1.62	70	28	h
3	Diana	1.58	58	21	m
4	Francisco	1.86	98	28	h

Imagen 1. Observando objeto con df.head().

Los resultados son idénticos que el archivo alumnos.csv, con la salvedad que están mejor presentados.

El resultado de `head` es la representación en filas y columnas.

Hay algunas salvedades a destacar:

1. En `Python`, los índices comienzan en 0.
2. La primera columna de nuestra tabla corresponde a la posición de la fila respecto al `DataFrame`. Esta información nos facilitará segmentación de archivos.

DataFrame

El objeto `df` que creamos recientemente es un objeto `DataFrame`, una de las estructuras elementales de `pandas`.

Un objeto `DataFrame` representa una tabla rectangular de datos compuestas por filas (observaciones registradas en el archivo) y una serie de columnas (atributos medibles que pueden ser integer, float, string, boolean, etc...).

Las observaciones registradas son insertadas en bloques bidimensionales que responden a la notación de matrices ***N x M***.

Podemos inspeccionar las dimensiones de la tabla mediante `.shape.` Éste nos informará de la cantidad de filas y columnas.

```
df.shape
```

```
(21, 5)
```

Esta tabla es manipulable y segmentable. Imaginemos que ahora deseamos extraer sólo las primeras 5 observaciones de nuestra tabla. La operación se realiza de la siguiente manera:

```
df[:5]
```

	nombre	altura	peso	edad	sexo
0	Hugo	1.67	60	23	h
1	Paco	1.73	83	25	h
2	Luis	1.62	70	28	h
3	Diana	1.58	58	21	m
4	Francisco	1.86	98	28	h

Imagen 2. Mostrando las primeras 5 observaciones.

Python interpretó esta instrucción como *"dentro de la tabla `df`, muéstrame las observaciones **hasta** la 5"*.

Eso se generó dentro de los brackets `[]`, donde pasamos un operador `:` que se llama *slice* y permite instruir hasta dónde se puede cortar un elemento.

En el caso anterior utilizamos *slice* para generar una submuestra hasta cierta condición (que se evalúa por el índice de la tabla; la primera columna).

¿Qué pasa si deseamos generar una segmentación *desde* un valor en específico?
Utilizamos la siguiente sintaxis:

```
df[15:]
```

	nombre	altura	peso	edad	sexo
15	Alejandra	1.86	74	21	m
16	Fernando	1.79	93	27	h
17	Carolina	1.60	63	28	m
18	Vicente	1.98	102	31	h
19	Benjamín	1.72	78	36	h
20	Gloria	1.58	65	23	m

Imagen 3. Mostrando desde un valor específico.

Acá instruimos a la tabla que entregue los resultados **desde** la fila 15 hasta el final de las observaciones.

¿Y si queremos seleccionar *entre* dos valores? Utilizamos la siguiente sintaxis:

```
df[5:10]
```

	nombre	altura	peso	edad	sexo
5	Felipe	1.79	100	26	h
6	Jacinta	1.69	62	20	m
7	Bernardo	1.60	83	31	h
8	Marisol	1.60	56	30	m
9	Facundo	1.98	112	36	h

Imagen 4. Mostrando observaciones entre dos valores.

Series

Las segmentaciones realizadas anteriormente fueron orientadas a las *filas* de una tabla. Esto también se puede realizar a las *columnas* de la tabla.

Para ello utilizamos una forma similar. Lo que vamos a separar la columna `altura`.

```
df['altura']
```

```
0     1.67
1     1.73
2     1.62
3     1.58
4     1.86
5     1.79
6     1.69
7     1.60
8     1.60
9     1.98
10    1.72
11    1.63
12    1.73
13    1.62
14    1.58
15    1.86
16    1.79
17    1.60
18    1.98
19    1.72
20    1.58
Name: altura, dtype: float64
```

Entre los brackets pasamos el nombre **exacto** de la columna que deseamos analizar.

Ya que trabajaremos con ésta columna, guardémosla en un nuevo objeto.

```
altura = df['altura']
type(altura)
```

```
pandas.core.series.Series
```

Cuando separamos esta columna y preguntamos por su tipo, Python nos entrega que es un objeto `pandas.core.series.Series`, este elemento que separamos se conoce como `Series`.

En `pandas`, las series son listas unidimensionales que contienen una secuencia de valores.

Todo objeto `pd.Series` tiene asociado una lista de etiquetas de datos denominada index. De manera similar a su comportamiento en DataFrame, nos permite realizar segmentaciones.

```
altura[15:]
```

```
15    1.86
16    1.79
17    1.60
18    1.98
19    1.72
20    1.58
Name: altura, dtype: float64
```

```
altura[:16]
```

```
0     1.67
1     1.73
2     1.62
3     1.58
4     1.86
5     1.79
6     1.69
7     1.60
8     1.60
9     1.98
10    1.72
11    1.63
12    1.73
13    1.62
14    1.58
15    1.86
Name: altura, dtype: float64
```

Estadística Univariada

Ahora que estamos familiarizados con los elementos básicos de un `DataFrame`, estamos en condiciones de generar nuestro primer análisis univariado.

Antes de entrar a la estadística univariada, es necesario hacer una digresión sobre cómo entender los datos desde múltiples visiones.

Digresión: ¿Cómo entender los datos?

Dada la naturaleza interdisciplinaria de Data Science, las distinciones entre tipos de datos resultan algo confusas. Existen tres grandes formas de entender los datos:

1. **Desde la programación** el enfoque es saber sobre la estructura específica de los datos. Nuestras preguntas están asociadas a saber si son `integer, float, string, boolean, etc...` Entender esto facilita la posterior construcción de funciones y pipelines dentro de Python.

2. **Desde la estadística** el enfoque es saber qué leyes estadísticas responden a nuestras variables. Entender que una variable que mide cantidad de eventos sigue un proceso Poisson o que la altura de una clase sigue una distribución normal permite preparar de mejor manera la estrategia analítica.

3. **Desde nuestro conocimiento específico** buscamos extraer la información relevante a lo que deberían medir nuestras observaciones y generar posibles alternativas para mejorar su desempeño.

4. La definición y naturaleza de los datos se determinan en gran medida por cómo se conceptualizaron y recolectaron.

5. Para delimitar y operacionalizar de mejor manera el proceso, separamos la definición de cada atributo a observar como variable.

6. Desde el sentido de la investigación, una **variable** es una *característica observable que varía entre distintas unidades de medición de una población*.

7. Existen dos grandes familias de variables en el proceso de originación de datos:
    - **Cualitativas**: Hacen referencia a características que son operacionalizadas por medio del número. Así, el número que representa cierto atributo no tiene sentido en sí. Existen dos familias de variables:
        - *Nominales*: Cada número representa un atributo no jerarquizado. Si estamos interesados en medir fenómenos como la religión o preferencia alimenticia de una persona, cada categoría se asociará a un número.
        - *Ordinal*: Cada número representa un atributo jerarquizado dentro de la variable. La clasificación de personas dentro de grupos etarios o niveles socioeconómicos responden a una estructura ordinal, donde el número hace referencia a un nivel dentro de la variable.
    - **Cuantitativas**: Hacen referencia a características operacionalizadas por medio del número. En este caso, el número representa una cantidad dentro del rango total de nuestra variable.
        - *Intervalar*: Cada número representa atributos medibles y la distancia entre dos atributos medibles tiene un valor intrínseco.
        - *Razón*: De forma adicional a la intervalar, en las variables de razón el cero absoluto representa algo.

El primer paso dentro de nuestro flujo de análisis es el resumen descriptivo de los datos observados.

Este resumen descriptivo permite describir las variables mediante los **momentos**:

1. **Medidas de Tendencia Central:** Buscan resumir aquél número que describa de mejor manera la variable.
2. **Medidas de Dispersión:** Resume qué tan dispersos están los datos *alrededor de la media*.
3. **Medidas de Sesgo:** Caracteriza qué tan desviada está la distribución de una variable respecto a su centro.
4. **Medidas de Kurtosis:** Caracteriza la altura del centro.

Media

Probablemente, la media es una de las medidas de tendencia central más conocidas. Ésta busca generar una cifra que represente de mejor manera la muestra que estudiamos.

Si tenemos un vector de datos *x* con tamaño $n(x_1, x_2, ..., x_n)$ la opción más intuitiva es calcular el promedio de todos los datos.

Para calcular un promedio, sumamos todos los datos del vector y posteriormente lo dividimos por la cantidad de datos.

$$\text{Media} \equiv \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Su implementación en Python nativo es simple. Si deseamos obtener la media de la altura de la clase, dividimos la suma de los valores por la cantidad de observaciones:

```
sum(altura) / len(altura)
```

```
1.7109523809523806
```

De forma alternativa podemos utilizar la función `mean` presente en `numpy`. Para ello, importemos el módulo de manera similar a como lo hicimos con `pandas`.

```
import numpy as np
np.mean(altura)
```

```
1.7109523809523806
```

Observamos que la altura promedio de nuestra media es de 1.71 metros. Cabe destacar que ambas soluciones entregan el mismo resultado.

Digresión: numpy

Numpy es un módulo orientado a la computación científica. Mientras que el fuerte de pandas es la manipulación de datos, la fortaleza de numpy está en las prestación de álgebra lineal y funciones matemáticas que entrega.

Moda

La moda es el valor observado con mayor frecuencia en la muestra. Dado que todos los datos observables se pueden contar, la moda sirve tanto para casos discretos o continuos.

$$Moda = x_i : n(x_i) > n(x_i) \quad \forall \quad j \neq i$$

Implementar la media sin módulos es un poco más complicado, así que lo dejaremos para adelante. La librería `scipy` tiene una función que calcula la moda de una muestra.

```
import scipy.stats as stats
stats.mode(altura)
```

```
ModeResult(mode=array([1.58]), count=array([3]))
```

`stats.mode` nos entrega la siguiente información: el valor que más se repite es 1.58 metros, y este se repite 3 veces.

Mediana

La mediana es el punto donde hay equidistancia en el vector X.

Para $X : x_1, x_2, \ldots, x_n$, donde x_i corresponden a los valores del vector X ordenados de manera ascendente, la mediana se obtiene a partir de la siguiente fórmula:

$$Mediana = \frac{1}{2}\left(x_{floor(\frac{n+1}{2})} + x_{ceiling(\frac{n+1}{2})}\right)$$

Donde floor y ceiling corresponden a funciones de aproximación al entero inferior y superior respectivamente y n corresponde al número de elementos del vector X.

`numpy` facilita la obtención de este cálculo mediante la función `median`:

```
np.median(altura)
```

```
1.69
```

El valor equidistante de la muestra es 1.69.

Con estas medidas logramos caracterizar cuál es la tendencia generalizada de nuestra muestra. Siguiendo con la estructura de los momentos, es necesario analizar qué tan dispersos se encuentran los datos en torno a estas medidas de centralidad.

Para ello utilizaremos las medidas de dispersión.

Medidas de Dispersión

Una vez que ya localizamos el punto donde se concentra la mayoría de los casos, el segundo momento es ver qué tan dispersos están alrededor de la tendencia central. Para ello nos valemos de las medidas de dispersión.

Con estos dos primeros momentos podemos resumir de buena manera la distribución empírica de las variables.

Percentiles

Los percentiles permiten identificar el valor donde un porcentaje en específico cae. Si tomamos el percentil 15%, el valor de este nos dirá el límite del vector.

Una manera de calcularlo es mediante el método del ranking más cercano que resume el punto de una lista ordenada de menor a mayor:

$$Percentil = [\frac{p}{100} \times N]$$

`numpy` permite calcular los percentiles, ingresando el objeto a analizar y el porcentaje que deseamos calcular. Si deseamos calcular los valores empíricos en nuestra muestra *asociados* al 50%, 75% y 90% de la muestra, ejecutamos lo siguiente:

```
print(np.percentile(altura,50))
print(np.percentile(altura,75))
print(np.percentile(altura,90))
```

```
1.69
1.79
1.86
```

El 50% de la muestra equivale a una altura de 1.69 metros. Cabe destacar que éste valor es exactamente el mismo que se calculó con `np.median()`. El 75% y el 90% de la muestra se representa por 1.79 y 1.86.

Rango

El rango de un vector resume la diferencia entre el menor y mayor valor.

$$Rango = máximo(x) - mínimo(x)$$

`numpy` provee de la función `ptp` para calcularlo:

```
np.ptp(altura)
```

```
/Users/veterok/anaconda3/lib/python3.6/site-packages/numpy/core/fromnume
ric.py:2223: FutureWarning: Method .ptp is deprecated and will be
removed in a future version. Use numpy.ptp instead.
  return ptp(axis=axis, out=out, **kwargs)

0.3999999999999999
```

El resultado es análogo a la siguiente implementación:

```
altura.max() - altura.min()
```

```
0.3999999999999999
```

El rango señala que la diferencia entre el menor y mayor valor de la altura de la clase es de .39.

Varianza

La varianza es la medida más común para cuantificar la dispersión de los datos.

Si tenemos un vector $n \in X : x1, x2, \ldots, x_n$, la varianza se puede obtener a partir de la siguiente fórmula:

$$\sigma^2 = \frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N-1}$$

Los pasos de la varianza conllevan a calcular la diferencia entre cada observación y la media del vector, elevarla al cuadrado (para evitar sumas que se cancelen entre sí) y sumarlas.

Posteriormente, se debe dividir por ***N -1. A N*** se le resta 1 para penalizar por el hecho que estamos estimando el segundo momento.

`numpy` provee de `var` para estimar qué tan dispersos se encuentran los datos **alrededor de la media**.

```
np.var(df['altura'])
```

```
0.015056235827664391
```

Con la media y la varianza de la altura a la mano, sabemos que la altura tiene un valor promedio de 1.71 con una varianza de 0.015.

Más adelante aprenderemos a aplicar estos números en leyes estadísticas que nos facilitarán la inferencia.

Desviación Estándar

La desviación estándar está fuertemente asociada con la varianza. Para obtenerla, es necesario sacar la raíz cuadrada de la varianza.

$$\text{DesvEst}(x) = \sqrt{\text{Var}(x)} = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N-1}}$$

Dado que tanto varianza como desviación estándar entregan información similar (qué tan disperso están los datos alrededor de la media), no existen razones de peso para preferir una por sobre la otra.

```
np.std(altura)
```

```
0.12270385416792902
```

Control de Flujo con pandas

Hasta el momento tenemos conocimiento sobre cómo funcionan las instrucciones de control de flujo en Python nativo. Teniendo en cuenta que el control de flujo permite agilizar los procedimientos que instruiremos en nuestro código, es una herramienta obligatoria a tener en cuenta.

Resulta que la implementación de control de flujo en el contexto de `pandas` es un poco más complejo. Esto considerando que ambas estructuras de datos que conocemos (`pd.Series` y `pd.DataFrame`) presentan elementos como índices, valores, filas y columnas que debemos tener en cuenta al momento de implementar control de flujo.

Loops `for` en objetos `pd.Series`

Para motivar esta sección, consideremos el siguiente caso: Vamos a generar un loop `for` en el `pd.DataFrame` que creamos anteriormente.

```
# para cada i en nuestra base de datos
for i in df:
    # imprime la representación de i
    print(i)

print("\nColumnas: ", df.columns)
```

```
nombre
altura
peso
edad
sexo

Columnas:  Index(['nombre', 'altura', 'peso', 'edad', 'sexo'],
dtype='object')
```

Por defecto, el loop nos entrega el nombre de las columnas. Este es el comportamiento por defecto que tiene `for` a nivel de `pd.DataFrame`. Más adelante aprenderemos sobre cómo acceder a los datos alojados a nivel de registro y de columna. Para profundizar el comportamiento de `for` en el contexto de `pandas`, veamos cómo funciona en interacción con un `pd.Series`. Para este caso iteraremos en la variable `df['peso']`, y la elevaremos al cuadrado sólo si es par.

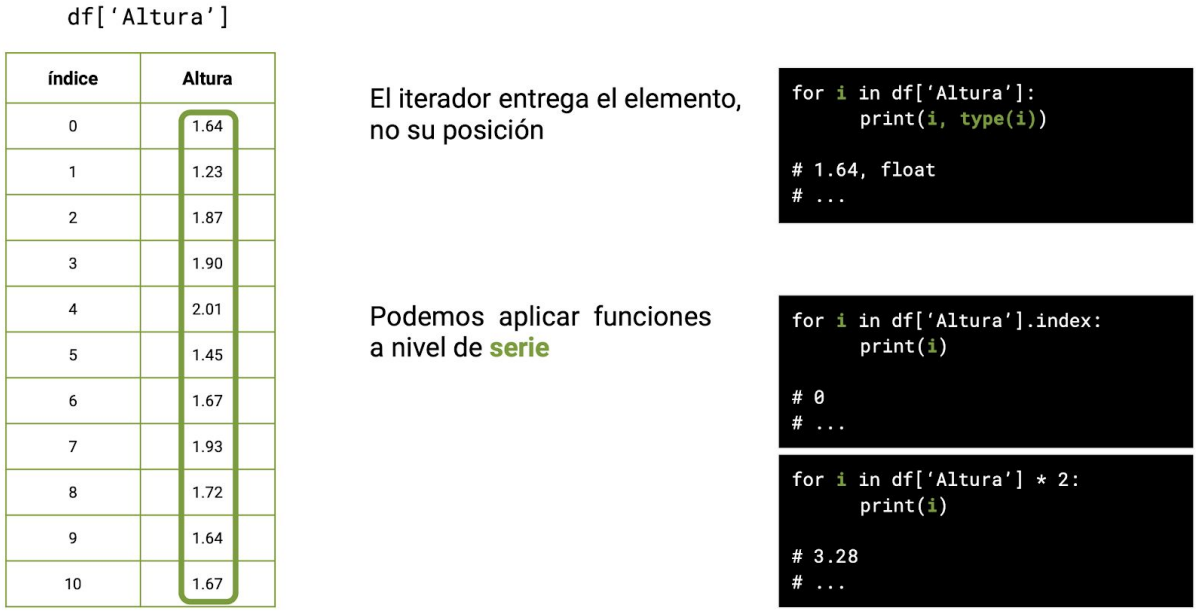

Imagen 5. Loops for en objetos.

```
for i in df['peso']:
    if i % 2 == 0:
        print(i ** 2)
```

```
3600
4900
3364
9604
10000
3844
3136
12544
5184
4624
6084
5476
10404
6084
```

Por defecto, el iterador infiere que el elemento a entregar es el valor en sí. Este punto es relevante a tomar en cuenta cuando buscamos implementar flujos. ¿Qué pasa si deseamos extraer la posición de cada valor?.
Podríamos estar tentados a realizar lo siguiente:

```
for i in df['peso']:
    print(i.index)
```

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

AttributeError                            Traceback (most recent call
last)

<ipython-input-24-e2901b39ae99> in <module>
      1 for i in df['peso']:
----> 2     print(i.index)
```

```
AttributeError: 'int' object has no attribute 'index'
```

Digresión: Lectura de errores

Recuerden que la parte más importante del error se encuentra en la última línea. Esta es la que informará sobre qué tipo de error se encontró en la expresión.

El problema es que estamos buscando un método del tipo index dentro de nuestro objeto iterable, el cual es un número entero. Una solución adecuada sería:

```
for i in df['peso'].index:
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Un problema recurrente con los loops es el hecho que las expresiones no persisten. Para guardar procedimientos en un `pd.Series`, podemos realizar lo siguiente:

```
# generamos una lista vacía
peso_cuadrado = []

# para cada uno de los valores en df['peso']
for i in df['peso']:
    # lo elevamos al cuadrado y posteriormente concatenamos al final de la lista
    peso_cuadrado.append(i ** 2)

# una vez terminado el proceso, convertimos la lista a una serie
peso_cuadrado = pd.Series(peso_cuadrado)
```

Resulta que una de las virtudes de la vectorización en el contexto de `pandas/numpy` es la simpleza de los procedimientos. El código anterior se puede reexpresar de la siguiente manera:

```
# esta nueva variable tendrá cada valor elevado al cuadrado
peso_cuadrado_vect = df['peso'] ** 2

# evaluamos si es que cada elemento es igual
peso_cuadrado == peso_cuadrado_vect
```

```
0     True
1     True
2     True
3     True
4     True
5     True
6     True
7     True
8     True
9     True
10    True
11    True
12    True
13    True
14    True
15    True
```

```
16    True
17    True
18    True
19    True
20    True
dtype: bool
```

Digresión: Sobre el uso y abuso de loops

- Más adelante aprenderemos sobre funciones vectorizadas en numpy, que sirven para aplicar procedimientos matemáticos *a una columna*, por sobre *cada elemento de la columna*.

- No sólo es más conveniente para nosotros, también es más rápido para el computador.

- Aún así, existirán casos donde nuestras rutinas alcanzarán mayores grados de complejidad y saber ejecutar un loop nos puede salvar mucho tiempo y capacidad de escalabilidad del código.

Iteradores en pd.DataFrame.iterrows()

Cuando comenzamos a hablar de los iteradores en el contexto de `pandas`, nuestro primer ejemplo fue realizar un `for i in df`, que devolvía el nombre de cada una de las columnas.

Resulta que para acceder a los valores alojados en las celdas, nuestro `pd.DataFrame` contiene una serie de métodos para facilitar nuestro flujo.

Comencemos por explorar el flujo a nivel de registros. Para ello haremos uso de `pd.DataFrame.iterrows()`.

Imagen 6. Uso de .iterrows().

Al utilizar `iterrows`, debemos tomar en cuenta **dos** iterables que se generan. El primero es el "nombre" del registro, el cual en este caso es el índice asociado a cada registro.

El segundo iterable corresponde a la serie del registro. En este caso, cada iterable tendrá las características de la unidad de análisis a lo largo de las columnas. Para ello, solicitamos el tipo de dato de cada serie iterable mediante `df.iterrows()`.

```
for rowname, rowserie in df.iterrows():
    print(type(rowserie))
```

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

- Otro de los elementos a considerar cuando implementamos los iteradores generados por `pandas` es el hecho que no es obligatorio ocupar ambos. Tomemos el caso de arriba como ejemplo: en este sólo utilizábamos el iterador correspondiente a la serie, ignorando el nombre de la fila.

Iteradores en pd.DataFrame.iteritems()

El segundo iterador a nivel de `pd.DataFrame` que podemos implementar es `df.iteritems`. De similar manera a como funciona `df.iterrows`, este entrega dos objetos iterables dentro del loop. La principal diferencia es cómo se recorren los elementos.

Cuando implementamos `df.iteritems`, estamos recorriendo registros **a lo largo de las columnas**. A diferencia de cuando implementamos `df.iterrows`, que recorremos registros **a lo largo de las filas**.

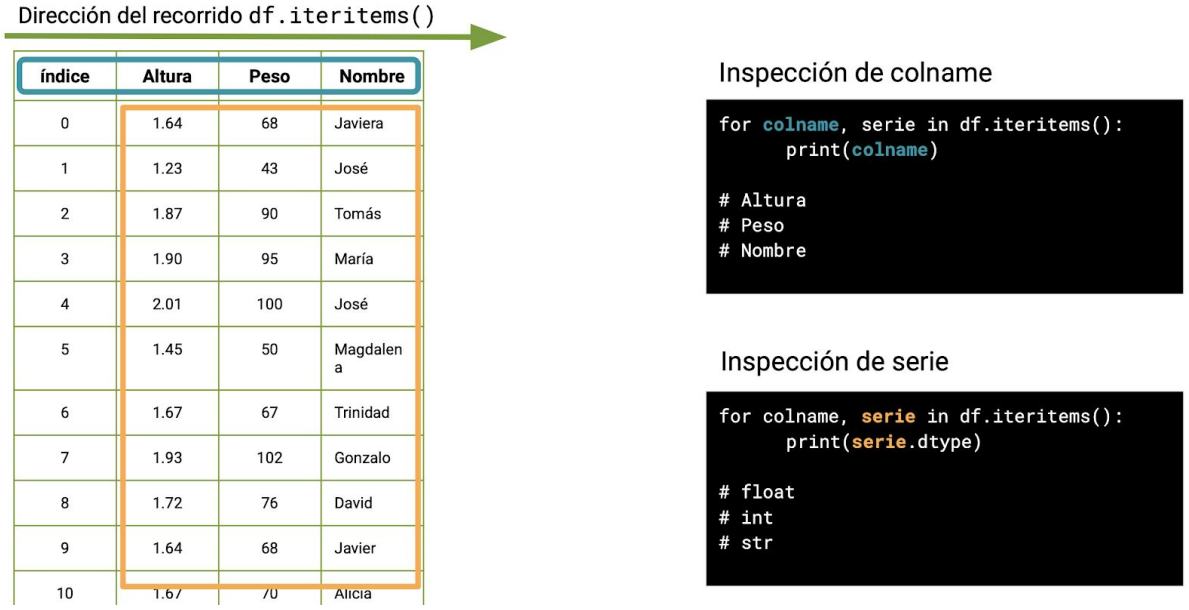

Imagen 7. Uso de .iteritems().

El primer iterable en `df.iteritems` hace referencia al nombre de las columnas, y el segundo iterable hace referencia a los valores de cada columna.

Un elemento importante a rescatar es el hecho que las series retornadas siempre tendrán el mismo tipo de dato, por lo que la cantidad de operaciones a realizar es mucho más amplia, en oposición a las series entregadas en `df.iterrows`, donde cada elemento dentro de la serie puede ser de un tipo distinto.

```
# evaluamos el primer iterable, ignorando el segundo con _
for colname, _ in df.iteritems():
    print(colname)
```

```
nombre
altura
peso
edad
sexo
```

```
# consultamos por la
for colname, colserie in df.iteritems():
    print(f"Tipo de dato de {colname}: {colserie.dtype}")
```

```
Tipo de dato de nombre: object
Tipo de dato de altura: float64
Tipo de dato de peso: int64
Tipo de dato de edad: int64
Tipo de dato de sexo: object
```

Segmentación booleana

Otra de las operaciones a realizar a nivel de `pd.DataFrame` es un subset booleano. El objetivo es separar una serie de registros en función a una operación booleana.

índice	Altura	Peso	Nombre
0	1.64	68	Javiera
1	1.23	43	José
2	1.87	90	Tomás
3	1.90	95	María
4	2.01	100	José
5	1.45	50	Magdalena
6	1.67	67	Trinidad
7	1.93	102	Gonzalo
8	1.72	76	David
9	1.64	68	Javier
10	1.67	70	Alicia

Selección en base a atributos

```
df[df['Peso'] < 70]
```

índice	Altura	Peso	Nombre
0	1.64	68	Javiera
1	1.23	43	José
5	1.45	50	Magdalena
6	1.67	67	Trinidad
9	1.64	68	Javier

Imagen 8. Segmentación booleana.

El punto de partida es generar una evaluación de un objeto `pd.Series` específico. En este caso deseamos evaluar todos los registros asociados a hombres en la columna sexo.

```
df['sexo'] == 'h'
```

```
0      True
1      True
2      True
3     False
4      True
5      True
6     False
7      True
8     False
9      True
10    False
11    False
12    False
13     True
14     True
15    False
16     True
17    False
18     True
19     True
20    False
Name: sexo, dtype: bool
```

Resulta que este retorno booleano podemos ingresarlos a nivel de matriz en nuestro `pd.DataFrame` para preservar solo las entradas donde el booleano equivale a `True`.

En este ejemplo, generaremos dos subconjuntos de datos en función a si corresponde a hombre o no.

```
edad_hombres = df[df['sexo'] == 'h']
edad_hombres
```

	nombre	altura	peso	edad	sexo
0	Hugo	1.67	60	23	h
1	Paco	1.73	83	25	h
2	Luis	1.62	70	28	h
4	Francisco	1.86	98	28	h
5	Felipe	1.79	100	26	h
7	Bernardo	1.60	83	31	h
9	Facundo	1.98	112	36	h
13	Diego	1.62	78	23	h
14	Gonzalo	1.58	67	22	h
16	Fernando	1.79	93	27	h
18	Vicente	1.98	102	31	h
19	Benjamín	1.72	78	36	h

Imagen 9. Mostrando datos si corresponde a hombres.

```
edad_mujeres = df[df['sexo'] != 'h']
edad_mujeres
```

	nombre	altura	peso	edad	sexo
3	Diana	1.58	58	21	m
6	Jacinta	1.69	62	20	m
8	Marisol	1.60	56	30	m
10	Trinidad	1.72	72	21	m
11	Camila	1.63	57	26	m
12	Macarena	1.73	68	27	m
15	Alejandra	1.86	74	21	m
17	Carolina	1.60	63	28	m
20	Gloria	1.58	65	23	m

Imagen 10. Mostrando datos si corresponde a mujeres.

Con ambos elementos, podemos implementar funciones clásicas `pd.DataFrame`.

```
print("La edad promedio para Hombres es: ",
np.mean(edad_hombres['edad']))
print("La edad promedio para Mujeres es: ",
np.mean(edad_mujeres['edad']))
```

```
La edad promedio para Hombres es:  28.0
La edad promedio para Mujeres es:  24.11111111111111
```

Iteradores en pd.DataFrame.iterrows()

Por último, también podemos expandir la funcionalidad de `pd.DataFrame().iterrows()` al extraer celdas específicas.

Esto se puede lograr mediante la incorporación de control de flujo dentro del iterable generado.

Índice	Sexo	Altura	Peso	Nombre
0	F	1.64	68	Javiera
1	M	1.53	43	José
2	M	1.87	90	Tomás
3	F	1.67	95	María
4	M	2.01	100	José
5	F	1.45	50	Magdalena
6	F	1.67	67	Trinidad
7	M	1.93	102	Gonzalo
8	M	1.72	76	David
9	M	1.64	68	Javier

```
mean_male = 0
for index, row in df.iterrows():
    if row['Sexo'] == 'M':
        mean_male += row['Peso']

print(mean_male/6)
```

Imagen 11. Mostrando celdas específicas.

Por otro lado, si deseamos contar la cantidad de mujeres mediante un for, la solución sería aproximada a:

```
# iniciamos el contador
count_female = 0
# para cada elemento de df['sexo']
for i in df['Sexo']:
    # si es de Sexo Femenino
    if i == 'F':
        # sumamos 1 al contador
        count_female += 1

count_female
```

```
4
```

El mismo resultado se puede lograr de manera equivalente

```
len(df[df['Sexo'] == 'F'])
```

```
4
```

Finalmente, en el caso que se busque evaluar más de una condición (cada condición evaluada de manera secuencial) se puede aplicar lo siguiente:

```
contador_hombres_bajo_promedio_altura = 0

# para cada posición y registro
for i, row in df.iterrows():
    # si el registro cumple con la condición menor a la media
    if (row['Altura'] < df['Altura'].mean()):
        # Y si éste es de Sexo Masculino
        if row['Sexo'] == 'M':
            # sumamos 1
            contador_hombres_bajo_promedio_altura += 1

contador_hombres_bajo_promedio_altura
```

```
3
```

Referencias

- McKinney, W. 2014. Python for Data Analysis. O'Reilley