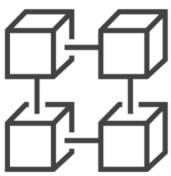
PYTHON DATA SCIENCE
MASTERY

MANUAL DEL CURSO

VOLUMEN I



DS4B

01_Python_Acelerado_Aprendizaje

September 16, 2021

1 CURSO ACELERADO DE PYTHON DS4B

1.1 OPERADORES

[1]:
```
2 + 2
```

[1]: 4

[2]:
```
2 - 1
```

[2]: 1

[3]:
```
2 * 2
```

[3]: 4

[4]:
```
4 / 2
```

[4]: 2.0

[5]:
```
2 ** 3
```

[5]: 8

[6]:
```
True
```

[6]: True

[7]:
```
False
```

[7]: False

[8]:
```
5 < 8
```

[8]: True

[9]:
```
8 > 5
```

[9]: True

[10]:
```
5 <= 8
```

[10]: True

[11]:
```
8 >= 5
```

[11]: True

[15]:
```
5 == 5
```

[15]: True

[16]:
```
5 != 5
```

[16]: False

[17]:
```
('a' == 'a') and (5 != 5)
```

[17]: False

[15]:
```
('a' == 'a') or (5 != 5)
```

[15]: True

[16]:
```
('a' == 'a') & (5 != 5)
```

[16]: False

[17]:
```
('a' == 'a') | (5 != 5)
```

[17]: True

[18]:
```
not True
```

[18]: False

[19]:
```
a = [1,2,3,4]
```

[25]:
```
a[1:4]
```

[25]: [2, 3, 4]

[19]:
```
#Operador slice. Se usa mucho para indexar por rango. [inicio:final:salto]
#Tener en cuenta que en Python los índices empiezan por cero y que el intervalo␣
↪de la derecha es abierto
a = [1,2,3,4]
a[1:3]
```

[19]: [2, 3]

[20]:
```
#Ejemplo del salto
a[0:3:2]
```

[20]:
```
[1, 3]
```

1.2 ASIGNACIONES E IMPRESIÓN DINÁMICA

[29]:
```
ventas_marzo = 5
```

[ ]:
```

```

Reglas para los nombres de las variables: * Pueden ser arbitrariamente largos. * Pueden contener tanto letras como números. * Deben empezar con letras. * Pueden aparecer subrayados para unir múltiples palabras. * No pueden ser palabras reservadas de Python.

[30]:
```
#Esta es la lista de palabras reservadas
import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async',
'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except',
'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda',
'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with',
'yield']
```

[2]:
```
nombre = 'Isaac'
edad = 41
```

[32]:
```
nombre
```

[32]:
```
'Isaac'
```

[33]:
```
print(nombre)
```

```
Isaac
```

[ ]:
```
a = 1
b = 2
```

[34]:
```
#Se pueden hacer asignaciones directas a varias variables a la vez
a,b,c,d = 1,2,3,4
c
```

[34]:
```
3
```

[35]:
```
"comillas dobles"
```

[35]:
```
'comillas dobles'
```

[36]:
```
'o comillas simples'
```

[36]:
```
'o comillas simples'
```

[37]:
```
"o simples 'dentro de dobles'"
```

[37]:
```
"o simples 'dentro de dobles'"
```

[38]:
```
'o simples "dentro" de dobles'
```

[38]:
```
'o simples "dentro" de dobles'
```

[39]:
```
'pero del 'mismo' tipo no funcionan'
```

```
  File "<ipython-input-39-905769bd42b0>", line 1
    'pero del 'mismo' tipo no funcionan'
               ^
SyntaxError: invalid syntax
```

Se pueden meter variables dentro de textos de forma dinámica

[40]:
```
print('Soy {} y tengo {} años'.format(nombre,edad))
```

```
Soy Isaac y tengo 41 años
```

Otra manera de introducir datos en texto de forma dinámica es con los indicadores de formato:

%d para enteros

%f para reales

%s para texto

Al terminar el texto se pondrá de nuevo % con el valor a incluir

[41]:
```
print('Hola me llamo %s' %'Isaac')
print('Tengo %d años y %d hijos' %(41,2))
```

```
Hola me llamo Isaac
Tengo 41 años y 2 hijos
```

[42]:
```
#En los reales podemos decirle el número de decimales que queremos
print('Con 2 decimales: %.2f' % 3.1416)
```

```
Con 2 decimales: 3.14
```

[43]:
```
#Si sumamos cadenas lo que hace es concatenarlas
print('Soy ' + 'Isaac y tengo ' + '41 años')
```

```
Soy Isaac y tengo 41 años
```

TIPOS DE DATOS

Individuales

Números

[44]:
```
#Tipo entero (int)
a = 14
print(type(a))

#Tipo real (float)
b = 14.34
print(type(b))
```

```
<class 'int'>
<class 'float'>
```

Nulos

En Python estandar se identifican con None. Pero cuidado porque en otros paquetes como Numpy, Pandas, ... tienen otras notaciones.

[45]:
```
nulo = None
type(nulo)
```

[45]:
```
NoneType
```

Fechas

Python estandar no tiene un tipo de datos como tal para las fechas. Hay que usar un módulo específico, por ejemplo datetime. No lo vamos a ver ya que tiene cierta complejidad y para la gestión de fechas usaremos principalmente Pandas.

Cadenas (texto) Un tipo especial de datos es cuando trabajamos con textos. Python tiene métodos específicos para este tipo de variables. Vamos a conocer los más frecuentes.

[53]:
```
cadena = 'Me llamo Isaac González y tengo 41 años'
```

[47]:
```
type(cadena)
```

[47]:
```
str
```

[48]:
```
cadena.lower()
```

[48]:
```
'me llamo isaac gonzález y tengo 41 años'
```

[49]:
```
cadena.upper()
```

[49]:
```
'ME LLAMO ISAAC GONZÁLEZ Y TENGO 41 AÑOS'
```

[50]:
```
cadena.split()
```

[50]:
```
['Me', 'llamo', 'Isaac', 'González', 'y', 'tengo', '41', 'años']
```

[51]:
```
cadena.split('y')
```

[51]:
```
['Me llamo Isaac González ', ' tengo 41 años']
```

[52]:
```
cadena.replace('Isaac','Mario')
```

[52]:
```
'Me llamo Mario González y tengo 41 años'
```

[54]:
```
cadena.count('a')
```

[54]:
```
4
```

[55]:
```
cadena.find('Isaac')
```

[55]:
```
9
```

Lista de todos los métodos de cadenas: https://www.w3schools.com/python/python_ref_string.asp

Conversiones

Se pueden convertir de un tipo a otro (siempre que tenga sentido, si no dará un error)

[56]:
```
float(14)
```

[56]:
```
14.0
```

[58]:
```
int(14.34)
```

[58]:
```
14
```

[59]:
```
str(14.34)
```

[59]:
```
'14.34'
```

[60]:
```
int('soy isaac')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-60-105eb27649f9> in <module>
----> 1 int('soy isaac')

ValueError: invalid literal for int() with base 10: 'soy isaac'
```

1.2.1 Secuencias

Rango

[62]:
```
#El rango en Python es tratado como un tipo de datos
rango = range(10)
type(rango)
```

[62]:
```
range
```

[64]:
```
#La sintaxis es range(inicio,final,paso)
list(range(0,10,2))
```

[64]:
```
[0, 2, 4, 6, 8]
```

Listas

Las listas se crean con corchetes. Pueden tener cualquier tipo de elemento (números, cadenas, otras listas, …), e incluso mezclados. Son mutables (se pueden añadir, eliminar o modificar elementos).

[71]:
```
lista1 = [1, 2, 3]
lista2 = ['rojo','amarillo','verde','rosa']
print(lista1,lista2)
```

```
[1, 2, 3] ['rojo', 'amarillo', 'verde', 'rosa']
```

[72]:
```
lista_mezclada = ['rojo',1,['Isaac','41']]
lista_mezclada
```

[72]:
```
['rojo', 1, ['Isaac', '41']]
```

[74]:
```
# también se pueden crear con la función list
lista3 = list((1,2,3))
lista3
```

[74]:
```
[1, 2, 3]
```

[75]:
```
#O simplemente crear una lista vacía para luego ir añadiendo elementos
vacia = []
type(vacia)
```

[75]:
```
list
```

[76]:
```
lista2
```

[76]:
```
['rojo', 'amarillo', 'verde', 'rosa']
```

[77]:
```
# Añadir un elemento a la lista
lista2.append('azul')
print(lista2)
```

```
['rojo', 'amarillo', 'verde', 'rosa', 'azul']
```

[78]:
```
# Para añadir varios elementos a la lista usamos extend
lista2.extend(['gris','negro'])
print(lista2)
```

```
['rojo', 'amarillo', 'verde', 'rosa', 'azul', 'gris', 'negro']
```

[79]:
```
lista2[0]
```

[79]:
```
'rojo'
```

[80]:
```
# Reemplazar elementos
lista2[0] = 'bermellon'
print(lista2)
```

```
['bermellon', 'amarillo', 'verde', 'rosa', 'azul', 'gris', 'negro']
```

[81]:
```
# Eliminar elementos por el índice
del lista2[1]
lista2
```

[81]:
```
['bermellon', 'verde', 'rosa', 'azul', 'gris', 'negro']
```

[82]:
```
# Eliminar elementos por su valor. Notar que ahora es un método
lista2.remove('rosa')
lista2
```

[82]:
```
['bermellon', 'verde', 'azul', 'gris', 'negro']
```

[83]:
```
# Sacar un elemento para meterlos en otra variable
cielo = lista2.pop(2)
cielo
```

[83]:
```
'azul'
```

[84]:
```
# Si no le pasamos argumento a pop saca el último elemento que haya
ultimo = lista2.pop()
ultimo
```

[84]:
```
'negro'
```

[85]:
```
lista2
```

[85]:
```
['bermellon', 'verde', 'gris']
```

[86]:
```
# Comprobar si un elemento está en una lista (parte 1)
'azul' in lista2
```

[86]:
```
False
```

[87]:
```
# Comprobar si un elemento está en una lista (parte 2)
'bermellon' in lista2
```

[87]:
```
True
```

[88]:
```
# Ver la longitud de una lista
len(lista2)
```

[88]:
```
3
```

[89]:
```
# Ordenar una lista. Fijarse que lo hace al vuelo (inplace)
sin_orden = ['f','h','a','g']
sin_orden.sort()
sin_orden
```

[89]:
```
['a', 'f', 'g', 'h']
```

[90]:
```
# Pero cuidado, sort sigue un orden donde las mayúsculas van antes que las␣
↪minúsculas
sin_orden = ['Isaac','alba']
sin_orden.sort()
sin_orden
```

[90]:
```
['Isaac', 'alba']
```

[91]:
```
# Si quieres ordenar listas mezcladas de mayúsculas y minúsculas hay que usar␣
↪el parámetro key = str.lower
sin_orden = ['Isaac','alba']
sin_orden.sort(key = str.lower)
sin_orden
```

[91]:
```
['alba', 'Isaac']
```

Referencia con todos los métodos de las listas: https://www.w3schools.com/python/python_ref_list.asp

Tuplas Se crean con paréntesis.

Pueden tener cualquier tipo de elemento (números, cadenas, otras listas, ...) incluso mezclados.

Son inmutables (tienen una longitud fija y no pueden cambiarse sus elementos).

[1]:
```
tupla1 = (1,'rojo',['Isaac',41])
tupla1
```

[1]:
```
(1, 'rojo', ['Isaac', 41])
```

[2]:
```
#Pero si intentamos cambiar un elemento no nos deja
tupla1[0] = 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-cc40538288d5> in <module>
      1 #Pero si intentamos cambiar un elemento no nos deja
----> 2 tupla1[0] = 2

TypeError: 'tuple' object does not support item assignment
```

[3]:
```
#También se puede crear con la función tuple sobre una lista
tupla1 = tuple([1,2,3])
tupla1
```

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

[4]:
```
#Si quisieras crear una tupla de un solo elemento hay que poner una coma al␣
 ↪final
print(type((5)))
print(type((5,)))
```

```
<class 'int'>
<class 'tuple'>
```

[5]:
```
#Las tuplas son el tipo de secuencia por defecto en Python.
#Si ponemos una secuencia sin especificar el formato Python entiende que son␣
 ↪tuplas
sin_formato = 1,2,3,4
type(sin_formato)
```

[5]:
```
tuple
```

Diccionarios Los diccionarios se crean con llaves.

Son pares no ordenados y mutables de clave-valor.

Pueden contener cualquier tipo de información, incluso mezclada.

[6]:
```
dicc1 = {'nombre':'Isaac', 'edad':41}
dicc1
```

[6]:
```
{'nombre': 'Isaac', 'edad': 41}
```

[7]:
```
#También se pueden crear con la función dict sobre una lista de tuplas con los␣
 ↪pares clave-valor
dicc1 = dict([('nombre','Isaac'),('edad',41)])
dicc1
```

[7]:
```
{'nombre': 'Isaac', 'edad': 41}
```

[8]:
```
#Crear un diccionario vacío
vacio = {}
vacio
```

[8]:
```
{}
```

[9]:
```
#Modificar un elemento
dicc1['edad'] = 43
dicc1
```

[9]:
```
{'nombre': 'Isaac', 'edad': 43}
```

[10]:
```
#Eliminar un elemento
del dicc1['edad']
dicc1
```

[10]:
```
{'nombre': 'Isaac'}
```

Referencia con todos los métodos de los diccionarios: https://www.w3schools.com/python/python_ref_dictionary.

Conjuntos (sets) Los conjuntos se crean también con llaves, pero no son pares.

Son colecciones de elementos UNICOS.

Pueden contener cualquier tipo de información, incluso mezclada. Son mutables.

[11]:
```
conj1 = {1,1,1,2,2,2,'rojo','rojo'}
conj1
```

[11]:
```
{1, 2, 'rojo'}
```

[ ]:
```
#También se pueden crear con la función set sobre una lista
lista = [1,1,1,2,2,2,'rojo','rojo']
conj2 = set(lista)
conj2
```

[15]:
```
conj2 = set(lista)
```

[16]:
```
conj2
```

[16]:
```
{1, 2, 'rojo'}
```

[17]:
```
#Podemos añadir un nuevo elemento con add
conj2.add('amarillo')
conj2
```

[17]:
```
{1, 2, 'amarillo', 'rojo'}
```

[18]:
```
#O añadir otro conjunto o lista con update
conj3 = {'negro','rosa'}
conj2.update(conj3)
conj2
```

[18]: {1, 2, 'amarillo', 'negro', 'rojo', 'rosa'}

[19]:
```
#Eliminar por valor con discard
conj2.discard('rosa')
conj2
```

[19]: {1, 2, 'amarillo', 'negro', 'rojo'}

[20]:
```
#Lógica de conjuntos: unión
conj1 = {'negro','rosa'}
conj2 = {'rosa','gris'}
conj1.union(conj2)
```

[20]: {'gris', 'negro', 'rosa'}

[21]:
```
#Lógica de conjuntos: intersección
conj1 = {'negro','rosa'}
conj2 = {'rosa','gris'}
conj1.intersection(conj2)
```

[21]: {'rosa'}

[22]:
```
#Lógica de conjuntos: diferencia del conjunto 1
conj1 = {'negro','rosa'}
conj2 = {'rosa','gris'}
conj1.difference(conj2)
```

[22]: {'negro'}

[23]:
```
#Lógica de conjuntos: diferencia del conjunto 2
conj1 = {'negro','rosa'}
conj2 = {'rosa','gris'}
conj2.difference(conj1)
```

[23]: {'gris'}

Referencia con todos los métodos de los conjuntos: https://www.w3schools.com/python/python_ref_set.asp

1.2.2 Repaso y a recordar:

- Podemos tener datos individuales o secuencias
- Dentro de los individuales los tipos más comunes son: int, float, none, str
- Dentro de las secuencias:
  - Rangos: son un tipo en sí mismos, se crean con range

- Listas: se crean con corchetes o con la función list
- Tuplas: se crean con paréntesis o con la función tuple
- Diccionarios: se crean con llaves o con la función dict. Son pares
- Sets: se crean con llaves o con la función set pero no son pares

1.3 INDEXACIÓN

Lo más importante es recordar siempre que en Python los índices empiezan en cero

[24]:
```
lista = [0,1,2,3,4]
lista[0]
```

[24]:
```
0
```

1.3.1 Indexar cadenas

[26]:
```
nombre = 'Isaac'
nombre[0]
```

[26]:
```
'I'
```

[27]:
```
#Se pueden pasar rangos con slice, pero el intervalo derecho es abierto (no␣
↪cuenta):
nombre = 'Isaac'
nombre[0:3]
```

[27]:
```
'Isa'
```

[29]:
```
#La indexación con números negativos lo que hace es empezar por el final
nombre[-1]
```

[29]:
```
'a'
```

1.3.2 Indexar listas

[30]:
```
lista = ['rojo','verde','azul']
lista[2]
```

[30]:
```
'azul'
```

[31]:
```
#Funciona slice
lista[0:2]
```

[31]:
```
['rojo', 'verde']
```

[32]:
```
#Las listas no se pueden indexar por valor
lista['rojo']
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-71d0f7577386> in <module>
      1 #Las listas no se pueden indexar por valor
----> 2 lista['rojo']

TypeError: list indices must be integers or slices, not str
```

[33]:
```
#Si queremos indexar por valor hay que usar index, pero cuidado, porque solo␣
↪devuelve lo primero que encuentra
lista = ['rojo','verde','rojo']
posicion = lista.index('rojo')
lista[posicion]
```

[33]:
```
'rojo'
```

[35]:
```
#La indexación con números negativos lo que hace es empezar por el final
lista = ['rojo','verde','azul']
lista[-1]
```

[35]:
```
'azul'
```

[ ]:
```
# Indexar listas dentro de listas
lista_anidada = ['Isaac','Maria',['Jose Antonio','Jose Manuel'],'Pedro']
print(lista_anidada[2])
print(lista_anidada[2][1])
```

Indexar tuplas

[40]:
```
#Se indexan por posición como las listas
tupla = (0,1,2,3,4)
tupla[0]
```

[40]:
```
0
```

[43]:
```
#Se pueden indexar varios elementos con slice
tupla[0:2]
```

[43]:
```
(0, 1)
```

[44]:
```
#La indexación con números negativos lo que hace es empezar por el final
tupla[-1]
```

[44]:
```
4
```

Indexar diccionarios

[45]:
```
#Si intentamos indexar por posición no funciona
dicc1 = {'nombre':'Isaac', 'edad':41}
dicc1[0]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-45-c252f0cf1533> in <module>
      1 #Si intentamos indexar por posición no funciona
      2 dicc1 = {'nombre':'Isaac', 'edad':41}
----> 3 dicc1[0]

KeyError: 0
```

[46]:
```
#Tenemos que indexar por la clave, y se indexa con corchetes, no con llaves!!
dicc1['nombre']
```

[46]:
```
'Isaac'
```

[47]:
```
#Indexar diccionarios o listas dentro de diccionarios
dicc_complejo = {'nombre':'Isaac',
                 'edad':41,
                 'familia':[{'mujer':'Gema'},{'hijos':['Carla','Mario']}]}
dicc_complejo
```

[47]:
```
{'nombre': 'Isaac',
 'edad': 41,
 'familia': [{'mujer': 'Gema'}, {'hijos': ['Carla', 'Mario']}]}
```

[48]:
```
dicc_complejo['familia']
```

[48]:
```
[{'mujer': 'Gema'}, {'hijos': ['Carla', 'Mario']}]
```

[49]:
```
dicc_complejo['familia'][1]
```

[49]:
```
{'hijos': ['Carla', 'Mario']}
```

[50]:
```
dicc_complejo['familia'][1]['hijos']
```

[50]:
```
['Carla', 'Mario']
```

[51]:
```
dicc_complejo['familia'][1]['hijos'][0]
```

[51]:
```
'Carla'
```

Nota: esta forma de indexación múltiple se puede usar en Python, pero después en Pandas veremos que no es buena práctica y tenemos alternativas mejores.

Indexar conjuntos

Los conjuntos no se pueden indexar

Repaso y a recordar:

- Los índices empiezan en cero
- Se puede utilizar slice para indexar
- Los índices con números negativos empiezan por atrás
- Las listas se indexan por posición. Si queremos por valor hay que usar index
- Las tuplas se indexan por posición
- Los diccionarios se indexan por su clave
- Los conjuntos no se pueden indexar
- En estructuras complejas de datos simplemente debemos identificar de qué tipo es el siguiente elemento anidado y utilzar su forma de indexación para ir avanzando

1.4 CONTROL DE FLUJO

1.4.1 If

Se utiliza para comprobar condiciones si ... entonces ... Detrás del si va un criterio, que si se se cumple se aplica el proceso del entonces.

Para mantener una sintaxis limpia Python utiliza (en esto y en todo) la indentación: cada línea se sabe a qué bloque pertenece por su nivel de indentación.

[54]:
```
x = 5
y = 3
if x > y:
    print('x es mayor que y')
```

```
x es mayor que y
```

También se puede definir un proceso para el caso de que no se cumpla la condición.

[57]:
```
x = 3
y = 5
if x > y:
    print('x es mayor que y')
else:
    print('y es mayor que x')
```

```
y es mayor que x
```

Notar la importancia de la indentación del código en Python. Esto no funciona:

[56]:
```
x = 3
y = 5
if x > y:
    print('x es mayor que y')
    else:
        print('y es mayor que x')
```

```
  File "<ipython-input-56-59eda1ce1ae3>", line 5
    else:
    ^
SyntaxError: invalid syntax
```

Se pueden incluir condiciones intermedias con elif

[59]:
```
x = 3
y = 6
if x > y:
    print('x es mayor que y')
elif x == y:
    print('x es igual que y')
else:
    print('y es mayor que x')
```

```
y es mayor que x
```

For

Sirve para crear bucles que recorran de forma iterativa cualquier tipo de elemento que sea iterable.

[60]:
```
for cada in range(5):
    print(cada)
```

```
0
1
2
3
4
```

[61]:
```
#Iterar una cadena
cadena = 'Isaac'
for cada in cadena:
    print(cada)
```

```
I
s
a
a
c
```

[62]:
```
#Iterar una lista
lista = ['Isaac','Pedro','Juan Carlos']
for nombre in lista:
    print(len(nombre))
```

```
5
```

```
5
11
```

[63]:
```
#Iterar una tupla
tupla = (0,1,2,3)
salida = list()
for cada in tupla:
    salida.append(cada * 2)
salida
```

[63]: [0, 2, 4, 6]

[65]:
```
res = [(1,2,3),(4,5,6)]
res
```

[65]: [(1, 2, 3), (4, 5, 6)]

[66]:
```
for cada in res:
    print(cada)
```

```
(1, 2, 3)
(4, 5, 6)
```

[67]:
```
#Si la tupla tiene varios valores podemos asignarlos directamente a diferentes␣
 ↪variables
#Es lo que se llama "tuple unpacking" y muchas funciones devuelven el resultado␣
 ↪como lista de tuplas, así que es útil
res = [(1,2,3),(4,5,6)]
for a,b,c in res:
    print(a,b,c)
```

```
1 2 3
4 5 6
```

[68]:
```
a
```

[68]: 4

[69]:
```
#Iterar un conjunto
conj = {'rojo','verde','azul'}
for cada in conj:
    print(cada)
```

```
rojo
verde
azul
```

[70]:
```
#Iterar un diccionario. Fijarse que nos devuelve solo las claves
dicc = {'uno': 1, 'dos': 2, 'tres': 3}
```

```
for cada in dicc:
    print(cada)
```

```
uno
dos
tres
```

[71]:
```
#Si queremos que nos devuelva el elemento compuesto por clave-valor tenemos que␣
 ↪usar items
dicc2 = {'uno': 1, 'dos': 2, 'tres': 3}
for cada in dicc2.items():
    print(cada)
```

```
('uno', 1)
('dos', 2)
('tres', 3)
```

[72]:
```
#Y como vemos nos ha devuelto una tupla, que ya podríamos desempaquetar con lo␣
 ↪que sabemos de tuple unpacking
dicc2 = {'uno': 1, 'dos': 2, 'tres': 3}
for letra,numero in dicc2.items():
    print(letra,numero)
```

```
uno 1
dos 2
tres 3
```

[73]:
```
#Si queremos solo las claves usaremos keys (es el comportamiento por defecto␣
 ↪que ya hemos visto)
dicc2 = {'uno': 1, 'dos': 2, 'tres': 3}
for cada in dicc2.keys():
    print(cada)
```

```
uno
dos
tres
```

[74]:
```
#Si queremos solo los valores usaremos values
dicc2 = {'uno': 1, 'dos': 2, 'tres': 3}
for cada in dicc2.values():
    print(cada)
```

```
1
2
3
```

[76]:
```
#Si queremos la posición y la clave separados usaremos enumerate
dicc2 = {'uno': 1, 'dos': 2, 'tres': 3}
for clave,valor in enumerate(dicc2):
```

```
    print('la posición es {} y el clave es {}'.format(clave,valor))
```

```
la posición es 0 y el clave es uno
la posición es 1 y el clave es dos
la posición es 2 y el clave es tres
```

La iteración de estructuras de datos mediante FOR será algo muy común, así que vamos a recapitular lo más importante que tenemos que recordar:

- Podemos iterar cadenas, tuplas, conjuntos, diccionarios o listas
- Con las tuplas es importante entender el tuple unpacking
- Si iteramos un diccionario, por defecto nos devolverá la clave
- Si queremos explícitamente las claves usamos nombrediccionario.keys()
- Si queremos los valores usamos nombrediccionario.values()
- Si queremos las claves y los valores usamos nombrediccionario.items()
- Si queremos la posición y las claves usamos nombrediccionario.enumerate()

1.4.2 List comprehension

Es una forma más compacta e incluso más rápida para iterar de forma similar a lo que hacemos con For. Su sintaxis es entre corchetes [haz esto por cada elemento en un iterable].

Aunque se llame list comprehension también funciona en conjuntos y diccionarios.

[77]:
```
[print(x) for x in range(5)]
```

```
0
1
2
3
4
```

[77]:
```
[None, None, None, None, None]
```

[81]:
```
#Se pueden incluir condiciones para hacerlo más flexible
[print(x) for x in range(0,5) if x>=3]
```

```
3
4
```

[81]:
```
[None, None]
```

[82]:
```
#Ejemplo en un diccionario
dicc = {'a':1,'b':2}
[clave.upper() for clave in dicc.keys()]
```

[82]:
```
['A', 'B']
```

1.4.3 Contadores

No son un elemento de sintaxis en sí mismos, pero es un recurso que se usa a veces en la programación, por lo que merece la pena conocerlo. Básicamente es definir una variable que de forma

iterativa se irá incrementando.

[83]:
```
#Un uso típico es para romper un bucle cuando el contador llegue a un límite
limite = 5
cont = 1
while cont < limite:
        print(cont)
        cont = cont + 1
print('Hasta aquí he llegado')
```

```
1
2
3
4
Hasta aquí he llegado
```

[84]:
```
#Hay una sintaxis reducida para los contadores cont += 1
limite = 5
cont = 1
while cont < limite:
        print(cont)
        cont += 1
print('Hasta aquí he llegado')
```

```
1
2
3
4
Hasta aquí he llegado
```

[88]:
```
#Otro uso típico es para ir "rellenando" por ejemplo un diccionario en función␣
 ↪de su índice
amigos = dict()
cont = 0

entrada = 'gsdfgdf'
while entrada != '':
    entrada = input('Dime un amigo: ')
    amigos[cont] = entrada
    cont = cont + 1
print(amigos)
```

```
Dime un amigo: Maria
Dime un amigo: Juan
Dime un amigo: Marcos
Dime un amigo: Teresa
Dime un amigo:
```

```
{0: 'Maria', 1: 'Juan', 2: 'Marcos', 3: 'Teresa', 4: ''}
```

1.4.4 Repaso y a recordar:

- If sirve para crear reglas si entonces. Sintaxis: if - elif - else
- For recorre cada elemento de cualquier objeto iterable
- Al iterar diccionarios acordarse de .items(), .keys(), .values(), .enumerate() y el concepto tuple unpacking
- Sintaxis de list comprehension: [haz esto por cada elemento en iterable si se cumple condición]
- Los contadores son un recurso frecuente en programación que debemos recordar

FUNCIONES PERSONALIZADAS

1.4.5 Funciones definidas por el usuario

Quizá sin saberlo, hemos estado usando un montón de funciones. Sin ir más lejos print es una.

En general una función es algo a lo que le pasas unos parámetros (aunque no necesariamente) y te devuelve una salida (aunque no necesariamente).

Pero además de las funciones predefinidas de Python nosotros podemos crear nuestras propias funciones.

Especificamos lo que queremos que devuelva con return.

[89]:
```
def suma_dos(num1,num2):
    resultado = num1 + num2
    return(resultado)
```

[90]:
```
#Para llamar a una función usaremos su nombre más paréntesis y dentro de los␣
↪mismos los parámetros que queramos pasarle
suma_dos(2,4)
```

[90]: 6

[93]:
```
#Se pueden poner valores por defecto a los parámetros para el caso de que no se␣
↪le pasen a la función
def suma_dos(num1,num2 = 4):
    resultado = num1 + num2
    return(resultado)
suma_dos(num1 = 2)
```

[93]: 25

[94]:
```
#Las funciones de Python pueden devolver varios valores
def suma_dos(num1,num2 = 4):
    resultado1 = num1 + num2
    resultado2 = num1 * num2
    return(resultado1,resultado2)
```

```
res1,res2 = suma_dos(num1 = 2)
print(res1)
print(res2)
```

```
6
8
```

1.4.6 Funciones lambda

Hay veces que no necesitaremos crear una función como tal ya que únicamente querremos algo puntual y "al vuelo".

Es lo que se llaman funciones lambda, que normalmente irán dentro de otros procesos para hacer su trabajo y luego no persistirán.

También se llaman anónimas, ya que no tienen nombre.

Otro punto importante es que sólo pueden tener una única expresión, no un bloque de acciones (que sí pueden tener las funciones normales).

Las usaremos mucho dentro de la función map (que veremos más adelante)

[95]:
```
#Ejemplo de cómo tendríamos que hacer para aplicar una función dentro de map␣
↪definiéndola tradicionalmente (con nombre)

numeros = [1,2,3,4]

def doble(num):
    return(num * 2)

list(map(doble,numeros))
```

[95]:
```
[2, 4, 6, 8]
```

[96]:
```
#Mismo ejemplo pero ahora con una función lambda (anónima)

list(map(lambda num: num * 2,numeros))
```

[96]:
```
[2, 4, 6, 8]
```

OTRAS FUNCIONES ÚTILES

Map

Sirve para aplicar una función a cada elemento de un objeto iterable. Pero devuelve un objeto map, que normalmente hay que convertir para visualizarlo, por ej a una lista, tupla, etc

[1]:
```
colores = ['rojo','amarillo','verde']
list(map(len,colores))
```

[1]:
```
[4, 8, 5]
```

[2]:
```
#También se puede usar con funciones personalizadas que hayamos construido␣
↪nosotros
def num_caracteres(texto):
    return(len(texto))

list(map(num_caracteres,colores))
```

[2]:
```
[4, 8, 5]
```

Es equivalente a un list comprehension, y es más sencillo de hacer.

[3]:
```
#El ejercicio superior con list comprehension
colores = ['rojo','amarillo','verde']

[len(color) for color in colores]
```

[3]:
```
[4, 8, 5]
```

Filter

Devuelve los elementos de una secuencia que cumplen una condición. Pero devuelve un objeto filter, que normalmente hay que convertir para visualizarlo, por ej a una lista, tupla, etc

[8]:
```
colores = ['rojo','amarillo','verde']
list(filter(lambda color: len(color) == 4, colores))
```

[8]:
```
['rojo']
```

Es equivalente a un list comprehension que incluya un if.

[7]:
```
#El ejercicio superior con list comprehension
colores = ['rojo','amarillo','verde']

[color for color in colores if len(color) == 4]
```

[7]:
```
['rojo']
```

Input

Sirve para pedir al usuario una entrada manual de datos

[9]:
```
pregunta_nombre = input("Cómo te llamas?\n")
print("Hola {}, encantado de saludarte".format(pregunta_nombre))
```

```
Cómo te llamas?
Isaac
Hola Isaac, encantado de saludarte
```

[10]:
```
#Cuidado, input siempre devuelve una cadena
pregunta_numero = input("Dime un número que le sumaré 5: ")
print(pregunta_numero + 5)
```

```
Dime un número que le sumaré 5: 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-b6270dcf9e41> in <module>
      1 #Cuidado, input siempre devuelve una cadena
      2 pregunta_numero = input("Dime un número que le sumaré 5: ")
----> 3 print(pregunta_numero + 5)

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

[11]:
```
#Si lo que queremos como entrada es un número tenemos que convertirlo con int()
pregunta_numero = int(input("Dime un número que le sumaré 5: "))
print(pregunta_numero + 5)
```

```
Dime un número que le sumaré 5: 3
8
```

Zip

Zip es una función que construye un objeto iterable a partir de otros y los empareja primero con primero, segundo con segundo y así.

Por ejemplo vamos a usarlo para unificar en un diccionario dos listas separadas de nombres y apellidos.

[12]:
```
nombres = ['Pedro','Manuel','Maria']
apellidos = ['García','González','Martinez']

dict(zip(nombres, apellidos))
```

[12]:
```
{'Pedro': 'García', 'Manuel': 'González', 'Maria': 'Martinez'}
```

Funciones estadísticas básicas

Notar que hay que pasarles una lista cuando hay varios valores

[13]:
```
sum([1,5,4])
```

[13]:
```
10
```

[14]:
```
max([1,5,4])
```

[14]:
```
5
```

[15]:
```
min([1,5,4])
```

[15]:
```
1
```

[16]:
```
abs(-5)
```

[16]: 5

[17]:
```
round(3.1416, ndigits=2)
```

[17]: 3.14

Referencia con todas las funciones internas de Python: https://www.w3schools.com/python/python_ref_functions

1.4.7 Repaso y a recordar:

- Podemos crear funciones personalizadas con def nombre(parámetros):
- Puede haber funciones sin parámetros, y también podemos definirlos por defecto
- Las funciones lambda no tienen nombre y se aplican al vuelo
- Map aplica una función a cada elemento de un iterable
- Filter permite filtrar elementos de una secuencia según un criterio
- Input devuelve una cadena, si queremos un entero hay que usar int()
- Zip va recorriendo los elementos de 2 iterables y emparejándolos

1.5 IMPORTACIÓN DE MÓDULOS

Lo que hemos visto hasta ahora ha sido un recorrido por lo más importante y lo que debes conocer de Python "base".

No obstante, la funcionalidad de Python no termina aquí, si no que de hecho empieza, ya que a partir de aquí hay infinidad de nuevas "ampliaciones" que permiten incorporar nuevas funciones y por tanto nuevas funcionalidades.

Es lo que se llaman módulos, o también hay gente que lo llama paquetes o librerías. Aunque realmente un paquete es una agregación de módulos.

Al final, conociendo la sintaxis base de Python que ya has estudiado, incorporar nuevas funcionalidades mediante módulos suele ser simplemente investigar cómo importarlos y la documentación de sus funciones.

Aquí puedes ver la lista de módulos "oficiales" que ya vienen con Python.

https://docs.python.org/3/py-modindex.html

Hay tres formas principales de importar módulos:

1. Usando la forma "import modulo as alias"
2. Usando la forma "from modulo import *"
3. O la forma "from modulo import funcion"

[19]:
```
#Con la 1 importamos todas las funciones, pero cada vez que las queramos usar␣
↪deberemos poner alias.funcion
#Por ejemplo vamos a importar el módulo random y generar un aleatorio
import random as rd
rd.random()
```

[19]: 0.3251103784044713

[20]:
```
#Con la 2 también importamos todas las funciones del módulo, pero no es␣
 ↪necesario poner el módulo cuando la llamemos
#Por ejemplo vamos a importar el módulo random y generar un aleatorio
from random import *
random()
```

[20]:
```
0.9744048711178545
```

[21]:
```
#Con la 3 importamos solo lo que le digamos, y no es necesario poner modulo.
 ↪funcion si no simplemente la función
#Por ejemplo vamos a importar el módulo random pero sólo con la función random␣
 ↪y generar un aleatorio
from random import random
random()
```

[21]:
```
0.6792994493939553
```

Cuando estemos trabajando en real seguramente estaremos usando funciones de varios módulos: numpy, pandas, seaborn, etc.

Por tanto se considera buena práctica usar la notación de módulo.función por legibilidad y claridad del código

03_Python_Acelerado_Ejercicios_Soluciones

September 16, 2021

1 CURSO ACELERADO DE PYTHON DS4B: SOLUCIÓN A EJERCICIOS

1.0.1 Ejercicio:

Calcula ocho menos 5

[102]:
```
8 - 5
```

[102]:
```
3
```

1.0.2 Ejercicio:

Calcula 4 al cuadrado

[103]:
```
4 ** 2
```

[103]:
```
16
```

1.0.3 Ejercicio:

Comprueba si 8 es diferente de 9

[104]:
```
8 != 9
```

[104]:
```
True
```

1.0.4 Ejercicio:

En una sola línea asigna 'Francisco' a la variable nombre y 'García' a la variable apellido, y en otras 2 líneas imprime cada uno

[105]:
```
nombre, apellido = ['Francisco','García']
print(nombre)
print(apellido)
```

```
Francisco
García
```

1.0.5 Ejercicio:

Usando las variables anteriores imprime de forma dinámica (usando format) la frase 'Me llamo Francisco y me apellido García'

[106]:
```
print('Me llamo {} y me apellido {}'.format(nombre, apellido))
```

```
Me llamo Francisco y me apellido García
```

1.0.6 Ejercicio:

Crea una variable frase que contenga 'Me llamo Francisco y me apellido García'. Y después pásalo todo a mayúsculas

[107]:
```
frase = 'Me llamo Francisco y me apellido García'
frase.upper()
```

[107]:
```
'ME LLAMO FRANCISCO Y ME APELLIDO GARCÍA'
```

1.0.7 Ejercicio:

Crea una lista (sin picarla a mano) que se llame lista_palabras y que contenga todas las palabras de la frase anterior. Después sácala por consola.

[108]:
```
lista_palabras = frase.split()
lista_palabras
```

[108]:
```
['Me', 'llamo', 'Francisco', 'y', 'me', 'apellido', 'García']
```

1.0.8 Ejercicio:

Cuenta cuantas palabras hay en lista_palabras

[109]:
```
len(lista_palabras)
```

[109]:
```
7
```

1.0.9 Ejercicio:

Convierte 13.24 a entero

[110]:
```
int(13.24)
```

[110]:
```
13
```

1.0.10 Ejercicio:

Convierte 13 a real

[111]:
```
float(13)
```

[111]: 13.0

1.0.11 Ejercicio:

Convierte 13 a texto

[112]:
```
str(13)
```

[112]: '13'

1.0.12 Ejercicio:

Crea una lista de los números impares entre y 30

[113]:
```
list(range(1,30,2))
```

[113]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]

1.0.13 Ejercicio:

Crea una lista llamada amigos que contenga a Juan, Pedro, María y Marta. Y sácala por consola.

[114]:
```
amigos = ['Juan', 'Pedro', 'María', 'Marta']
amigos
```

[114]: ['Juan', 'Pedro', 'María', 'Marta']

1.0.14 Ejercicio:

Acaban de conocer a Marcos y Sofía y se han hecho amigos. Añádelos al grupo. Y saca la nueva lista por consola.

[115]:
```
amigos.extend(['Marcos','Sofia'])
amigos
```

[115]: ['Juan', 'Pedro', 'María', 'Marta', 'Marcos', 'Sofia']

1.0.15 Ejercicio:

También han conocido a Paula. Añádela al grupo. Y saca la nueva lista por consola.

[116]:
```
amigos.append('Paula')
amigos
```

[116]: ['Juan', 'Pedro', 'María', 'Marta', 'Marcos', 'Sofia', 'Paula']

1.0.16 Ejercicio:

Al final Paula salió rara. Sácala de la lista e imprime por consola como ha quedado el grupo

[117]:
```
amigos.remove('Paula')
amigos
```

[117]:
```
['Juan', 'Pedro', 'María', 'Marta', 'Marcos', 'Sofia']
```

1.0.17 Ejercicio:

¿Hay algún amigo que se llame Alfredo? (Verdadero/Falso)

[118]:
```
'Alfredo' in amigos
```

[118]:
```
False
```

1.0.18 Ejercicio:

Ordena alfabéticamente la lista de amigos y saca el resultado por consola

[119]:
```
amigos.sort(key=str.lower)
amigos
```

[119]:
```
['Juan', 'Marcos', 'Marta', 'María', 'Pedro', 'Sofia']
```

1.0.19 Ejercicio:

Saca por consola solo los 3 primeros amigos de la lista.

[120]:
```
amigos[0:3]
```

[120]:
```
['Juan', 'Marcos', 'Marta']
```

1.0.20 Ejercicio:

Localiza en qué posición de la lista está Pedro (empezando por cero)

[121]:
```
amigos.index('Pedro')
```

[121]:
```
4
```

1.0.21 Ejercicio:

Usa list comprehension para sacar por consola sólo los nombres de amigos que empiecen por M

[122]:
```
[print(n) for n in amigos if n.startswith('M')]
```

```
Marcos
Marta
María
```

[122]:
```
[None, None, None]
```

1.0.22 Ejercicio:

Ahora haz lo mismo pero sin usar list comprehension (con for e if "clásicos")

[123]:
```
for n in amigos:
    if n.startswith('M'):
        print(n)
```

```
Marcos
Marta
María
```

1.0.23 Ejercicio:

Crea un contador de 'nombres que empiecen por M', recorre toda la lista con un for, suma 1 cada vez que encuentres uno e imprime el resultado final.

[124]:
```
cont = 0
for n in amigos:
    if n.startswith('M'):
        cont += 1
print(cont)
```

```
3
```

1.0.24 Ejercicio:

A continuación te paso listas con los chicos y las chicas. Están en orden de quien es pareja con quien. Tienes que crear el diccionario pareja_de donde las claves sean los chicos y los valores las chicas. Finalmente imprímelo por consola.

[125]:
```
chicos = ['Juan','Marcos','Pedro']
chicas = ['Marta','María','Sofía']
```

[126]:
```
pareja_de = dict(zip(chicos,chicas))
print(pareja_de)
```

```
{'Juan': 'Marta', 'Marcos': 'María', 'Pedro': 'Sofía'}
```

1.0.25 Ejercicio:

Saca por consola la pareja de Marcos

[127]:
```
pareja_de['Marcos']
```

[127]: 'María'

1.0.26 Ejercicio:

Al final Marcos y María han roto, y ahora la nueva pareja de Marcos es Paula. Actualízadlo en el diccionario e imprime como queda ahora el diccionario.

[128]:
```
pareja_de['Marcos'] = 'Paula'
pareja_de
```

[128]:
```
{'Juan': 'Marta', 'Marcos': 'Paula', 'Pedro': 'Sofía'}
```

1.0.27 Ejercicio:

El grupo se ha puesto de parte de María, así que han echado a Marcos y a su nueva pareja. Sácalos del grupo e imprime cómo queda.

[129]:
```
del pareja_de['Marcos']
pareja_de
```

[129]:
```
{'Juan': 'Marta', 'Pedro': 'Sofía'}
```

1.0.28 Ejercicio:

Hay que actualizar la lista de chicos y chicas. Hazlo siguiendo estas instrucciones:

1. Crea nuevas listas vacías de chicos y chicas
2. Recorre el diccinario pareja_de con un for y mediante tuple unpacking separa cada item del diccionario en chico y chica
3. Añade chico y chica a sus respectivas listas
4. Saca por pantalla chicos y chicas

[130]:
```
chicos = []
chicas = []
for chico,chica in pareja_de.items():
    chicos.append(chico)
    chicas.append(chica)
print(chicos)
print(chicas)
```

```
['Juan', 'Pedro']
['Marta', 'Sofía']
```

1.0.29 Ejercicio:

Extrae la palabra 'perro' del siguiente objeto.

[131]:
```
obj = [1,2,{'nombre':'Manuel','ciudad':'Madrid','vive_con':[{'mujer':␣
↪'Julia','hijo':'Andres','mascotas':['gato','perro']},'ah y un hamster']},4,5]
```

[132]:
```
obj[2]['vive_con'][0]['mascotas'][1]
```

[132]:
```
'perro'
```

1.0.30 Ejercicio:

Crea una función que se llame imc para calcular el índice de masa corporal a partir de la estatura (en metros) y el peso (en kilos).

La fórmula es IMC = peso (kg) / estatura (m) al cuadrado.

Pruébala pasándole por ejemplo tus datos.

[133]:
```
def imc(peso,estatura):
    imc = peso / (estatura **2)
    return(imc)

imc(100,1.8)
```

[133]:
```
30.864197530864196
```

1.0.31 Ejercicio:

Crea la misma función pero como una lambda que meta el dato en la variable resultado. Pruébala de nuevo.

[134]:
```
resultado = lambda peso,estatura: peso / (estatura **2)

resultado(100,1.8)
```

[134]:
```
30.864197530864196
```

1.0.32 Ejercicio:

Crea un pequeño programa que:

1. Pregunte al usuario por su peso en kilos y su estatura en centímetros
2. Llame a la función imc (acuérdate de transformar la estatura a metros)
3. Si imc >25 aconseje al usuario un poco de dieta y ejercicio

[137]:
```
peso = int(input('\n Introduce tu peso en kilos: '))
estatura = int(input('\n Introduce tu estatura en centímetros: '))
estatura = estatura / 100

resultado = imc(peso,estatura)
if resultado > 25:
    print('\n Umm, tu imc es de %.2f quizá sería recomendable un poco de dieta␣
 ↪y ejercicio'%resultado)
else:
    print('\n Estás estupend@, sigue así')
```

```
 Introduce tu peso en kilos: 100

 Introduce tu estatura en centímetros: 180
```

```
Umm, tu imc es de 30.86 quizá sería recomendable un poco de dieta y ejercicio
```

1.0.33 Ejercicio:

Importa los módulos random y statistics y crea un programa que:

Cree una lista vacía llamada imcs

Repita 100 veces el siguiente proceso:

```
* Calcular aletoriamente un peso entre 50 y 100
* Calcular aletoriamente una estatura entre 150 y 200 y convertirla a metros
* Llamar a la función imc y añadir el resultado a la lista imcs
```

Usando funciones del paquete statistics imprima por pantalla la media y la desviación típica de la distribución final obtenida.

Pista: puedes buscar las funciones que te ayuden a hacerlo en:

https://www.w3schools.com/python/module_random.asp

https://www.w3schools.com/python/ref_stat_mean.asp

[136]:
```python
import random
import statistics

imcs = []

for cada in range(100):
    peso = random.randint(50,100)
    estatura = random.randint(150,200)
    estatura = estatura / 100
    resultado = imc(peso,estatura)
    imcs.append(resultado)

print(statistics.mean(imcs))
print(statistics.stdev(imcs))
```

```
27.78832633918493
9.122130266530384
```

04_PythonAcelerado_Programa1_PiedraPapelTijera

September 16, 2021

0.1 EJERCICIO DE PIEDRA-PAPEL-TIJERA

En este ejercicio vamos a crear este popular juego simplemente como forma de practicar sobre algunos de los principales conceptos que aprendimos en el curso acelerado de Python.

El primer paso siempre es hacer un diseño de lo que queremos que haga el programa. En nuestro caso:

1. Preguntar el nombre al jugador
2. Explicar las reglas del juego
3. Preguntar al jugador qué quiere sacar
4. Hacer una tirada aleatoria por el ordenador
5. Resolver quien ha ganado y sumarle el punto
6. Repetir del 3 al 5 diez veces
7. Evaluar el ganador final
8. Mostrar el resultado

Suele ser buena práctica intentar sacar en funciones todas aquellas operaciones que puedan ser aisladas o que vayan a repetirse varias veces.

En nuestro caso vamos a sacar como funciones los siguiente procesos: * Preguntar al jugador qué quiere sacar * Hacer una tirada aleatoria por el ordenador * Resolver quien ha ganado y sumarle el punto

Y el resto, que es secuencial, vamos a dejarlo como la parte principal del programa, que irá llamando a esas funciones cuando las necesite.

Cuando hacemos esto hay que definir primero las funciones y dejar la parte principal del programa para el final.

Vamos a por ello!

PASO 1:

Define una función, llamada jugador, que pregunte al jugador qué jugada quiere hacer usando los códigos de abajo, y devuelva el resultado como un entero.

1: Piedra

2: Papel

3: Tijera

[1]:
```
def jugador():
    respuesta = input('\n ¿Qué jugada eliges (1: Piedra, 2: Papel, 3: Tijera)?:␣
 ↪')
    return int(respuesta)
```

PASO 2:

Define una función, llamada ordenador, que seleccione aleatoriamente entre piedra, papel o tijera y devuelva el resultado.

Pista: necesitarás el módulo random, y puedes buscar una función que te ayude en:

https://www.w3schools.com/python/module_random.asp

[2]:
```
import random

def ordenador():
    opciones = ['piedra','papel','tijera']
    opcion = random.choice(opciones)
    return(opcion)
```

PASO 3:

Define una función, llamada compara, que tome como parámetros el resultado de la jugada del jugador y del ordenador, implemente la lógica del juego y devuelva el resultado, que podrá ser quien ha ganado o si ha habido empate.

[3]:
```
def compara(jugador,ordenador):
    ganador = None
    if (jugador == 1 and ordenador == 'tijera'): ganador = 'jugador'
    elif (jugador == 1 and ordenador == 'papel'): ganador = 'ordenador'
    elif (jugador == 1 and ordenador == 'piedra'): ganador = 'empate'
    elif (jugador == 2 and ordenador == 'tijera'): ganador = 'ordenador'
    elif (jugador == 2 and ordenador == 'papel'): ganador = 'empate'
    elif (jugador == 2 and ordenador == 'piedra'): ganador = 'jugador'
    elif (jugador == 3 and ordenador == 'tijera'): ganador = 'empate'
    elif (jugador == 3 and ordenador == 'papel'): ganador = 'jugador'
    elif (jugador == 3 and ordenador == 'piedra'): ganador = 'ordenador'
    else: print('Ha habido algún error')
    return(ganador)
```

PASO 4:

Define la parte principal del programa que vaya siguiendo lo definido en el diseño y llamando a las funciones que hemos creado cuando sea necesario.

Cuando lo hayas terminado ejecuta todas las celdas y diviértete!!

[ ]:
```
nombre = input('Cómo te llamas?: ')
print('Hola %s vamos a echar 10 jugadas. Y el que consiga más puntos gana!'␣
 ↪%nombre)
```

```
puntos_j = 0
puntos_o = 0

for cada in range(10):
    tirada_jugador = jugador()
    tirada_ordenador = ordenador()
    ganador = compara(tirada_jugador,tirada_ordenador)
    if ganador == 'jugador':
        print('\n Yo he sacado %s'%tirada_ordenador)
        print('\n Así que esta la has ganado tú!!')
        puntos_j += 1
    elif ganador == 'ordenador':
        print('\n Yo he sacado %s'%tirada_ordenador)
        print('\n Así que esta la he ganado yo!!')
        puntos_o += 1
    elif ganador == 'empate':
        print('\n Vaya, yo también he sacado %s'%tirada_ordenador)
        print('\n Así que hemos empatado, venga otra mano')
    else: print('Ha habido algún problema')

if puntos_j > puntos_o:
    puntos_finales = puntos_j
    print('\n HEMOS TERMINADO!!')
    print('\n El resultado final es que has ganado tú con {} puntos'.
 ↪format(puntos_finales))
elif puntos_j > puntos_o:
    puntos_finales = puntos_o
    print('\n El resultado final es que he ganado yo con {} puntos'.
 ↪format(puntos_finales))
else:
    print('\n Pues parece que hemos empatado, así que todos ganamos!')
```

[ ]:

05_PythonAcelerado_Programa2_Agenda

September 16, 2021

0.1 EJERCICIO DE CREAR UNA MINI-AGENDA

En este ejercicio vamos a hacer uno de los casos más típicos cuando uno está aprendiendo a programar, crear una mini-agenda de teléfonos.

Además de practicar y consolidar lo aprendido te permitirá darte cuenta de realmente el tipo diferente de programas que ya puedes hacer combinando de forma diferente todas las piezas del "puzzle" (funciones y sintaxis de Python) que sabes.

Si ya te sientes medianamente confiado te animo a que en este caso hagas un primer intento por tu cuenta de resolver este caso sin ver primero la solución.

Este es el diseño de lo que queremos que haga el programa en este caso:

1. Crear un diccionario vacío al iniciarse
2. Mostrar un menú al usuario que le permita elegir entre: crear, modificar o eliminar un registro, o salir del programa
3. Resolver esa acción que ha eligido el usuario y después volver al menú principal (salvo que haya elegido salir en cuyo caso el programa termina)
4. Si ha elegido crear, el programa preguntará el nombre y el teléfono y lo añadirá al diccionario siendo el nombre la clave y el teléfono el valor
5. Si ha elegido borrar, el programa pedirá un nombre y borrará ese registro de la agenda
6. Si ha elegido modificar, el programa preguntará el nombre y el teléfono, buscará ese nombre en la agenda y modificará su valor (el teléfono)

En este caso te recomiendo sacar como funciones los procesos de crear, modificar y borrar usuarios, y dejar el resto de la lógica en la parte principal del programa.

Vamos a por ello!

PASO 1:

Crea un diccionario vacío llamado agenda y define, siguiendo el diseño indicado, funciones para:

- Crear un registro
- Borrar un registro
- Modificar un registro

[ ]:
```
agenda = {}
```

[ ]:
```
def crear_registro():
    nombre = input('Nombre: ')
```

```
    tlf = input('Teléfono: ')
    agenda[nombre] = tlf
```

[ ]:
```
def borrar_registro():
    a_borrar = input('Nombre a borrar: ')
    del agenda[a_borrar]
    print('Borrado con éxito!')
```

[ ]:
```
def modificar_registro():
    nombre_modificar = input('Nombre a modificar: ')
    tlf_modificar = input('Nuevo teléfono: ')
    agenda[nombre_modificar] = tlf_modificar
```

PASO 2:

Crea la lógica principal del programa donde se muestre un menú con las siguientes opciones que estará activo y funcionando hasta que el usuario elija la opción de Salir:

1. Crear un nuevo registro
2. Modificar un registro existente
3. Eliminar un registro
4. Salir

[ ]:
```
opcion = 0
while opcion != 4:
    print('1. Crear un nuevo registro')
    print('2. Modificar un registro existente')
    print('3. Eliminar un registro')
    print('4. Salir')
    opcion = int(input('Qué quieres hacer: '))
    if opcion == 1: crear_registro()
    elif opcion == 2: modificar_registro()
    elif opcion == 3: borrar_registro()
    print('\n\n')
```

PASO 3:

Crea, modifica y borra unos cuantos registros, y cuando hayas terminado y elegido salir imprime la agenda por consola y comprueba el resultado.

[ ]:
```
agenda
```

01_IntensivoNumpy

September 16, 2021

1 CURSO INTENSIVO DE NUMPY

1.1 INSTALACIÓN

Se instala con conda install numpy

1.2 INTRODUCCIÓN E IMPORTACIÓN

Es un paquete de computación eficiente muy enfocado al álgebra lineal.

Muy rápido porque usa muchas librerías de C.

El resto de paquetes de ML están basados en él.

Dos estructuras básicas vectores (array de una dimensión) y matrices (array de 2 dimensiones).

Como término que unifica ambos usaremos array.

[1]:
```
import numpy as np
```

¿Por qué Numpy?

Ahora que ya conocemos las estructuras de datos de Python vemos que no tiene realmente ninguna realmente orientada a trabajar con cálculos de manera masiva.

Están más orientadas a almacenar y recuperar información, pero no a procesarla en grandes cantidades y de forma eficiente.

Por otro lado casi todos los métodos de análisis y de machine learning se basan en el concepto de vector.

Lo más parecido que tiene Python a un vector podría ser una lista. Pero esta carece de lo que se llaman "operaciones vectorizadas", es decir, aplicar el mismo cálculo a todos los componentes del vector, y que es clave en data science.

[2]:
```
# Por ejemplo si creamos una lista y la multiplicamos por dos
# No conseguimos el efecto que seguramente estábamos esperando
lista = [1,2,3]
lista * 2
```

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

[3]:
```
# Necesitaríamos usar list comprehension (o un for) para hacerlo
[elemento * 2 for elemento in lista]
```

[3]:
```
[2, 4, 6]
```

[4]:
```
# Sin embargo si la pasamos a un vector de Numpy sí
lista_vec = np.array([1,2,3])
lista_vec * 2
```

[4]:
```
array([2, 4, 6])
```

A nivel de estructura de datos Numpy utiliza el array multidimensional (array de n dimensiones)

[5]:
```
v = np.array([1,2,3])
type(v)
```

[5]:
```
numpy.ndarray
```

A nivel de tipo de dato es importante ver cómo lo define: con el tipo de dato (por ejemplo entero) más el número de bits que usa (16, 32, 64, etc.).

Es importante porque en Pandas veremos que usa la misma notación.

[6]:
```
v.dtype
```

[6]:
```
dtype('int32')
```

Que no nos confunda que Numpy sea una librería numérica. También puede haber arrays no numéricos.

Pero eso sí, **todos los componentes del array tienen que ser del mismo tipo** (que será una de las cosas que nos llevará a Pandas).

[7]:
```
v = np.array(['a','b','c'])
type(v)
```

[7]:
```
numpy.ndarray
```

[8]:
```
#Si ponemos de diferentes tipos vemos que los convierte para que sean del mismo␣
↪tipo
v = np.array(['a',2,'c'])
v
```

[8]:
```
array(['a', '2', 'c'], dtype='<U11')
```

1.3 COMO CREAR UN ARRAY

1.3.1 A PARTIR DE UNA LISTA

[9]:
```
#Crear un vector a partir una lista
v = np.array([1,2,3])
v
```

[9]:
```
array([1, 2, 3])
```

[10]:
```
#Crear una matriz a partir de una lista de listas
m = np.array([[1,2,3],[4,5,6]])
m
```

[10]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

1.3.2 A PARTIR DE UN RANGO

[11]:
```
#Crear un vector a partir de un rango
np.arange(1,10,2)
```

[11]:
```
array([1, 3, 5, 7, 9])
```

[12]:
```
#Crear un vector interpolando entre dos números
np.linspace(1,10,20)
```

[12]:
```
array([ 1.        ,  1.47368421,  1.94736842,  2.42105263,  2.89473684,
        3.36842105,  3.84210526,  4.31578947,  4.78947368,  5.26315789,
        5.73684211,  6.21052632,  6.68421053,  7.15789474,  7.63157895,
        8.10526316,  8.57894737,  9.05263158,  9.52631579, 10.        ])
```

1.3.3 CREAR UN ARRAY PREDEFINIDO

[13]:
```
#Crear un array de ceros (el parámetro es una tupla con filas, columnas)
np.zeros((4,3))
```

[13]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

[14]:
```
#Crear una matriz de identidad
np.eye(5)
```

[14]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
```

```
       [0., 0., 0., 0., 1.]])
```

1.3.4 CREAR UN ARRAY ALEATORIO

Con los aletorios de Python que hemos visto solamente podíamos extraer uno de cada vez.

Teniendo que hacer list comprehension para obtener toda una lista.

[15]:
```
#Esto nos devuelve solo un valor
import random as rd
rd.randint(1,10)
```

[15]:
```
2
```

[16]:
```
#Si quierámos por ejemplo 10
[rd.randint(1,10) for cada in range(10)]
```

[16]:
```
[4, 4, 10, 8, 3, 7, 3, 6, 3, 6]
```

Sin embargo vamos a ver cómo con Numpy generamos directamente vectores (que podrán ser variables) que es normalmente lo que queremos.

[17]:
```
#Aleatorios de 0-1 con distribución uniforme
np.random.rand(5,10)
```

[17]:
```
array([[0.62600308, 0.63367351, 0.69365151, 0.67164287, 0.04587912,
        0.37832065, 0.99182729, 0.20674233, 0.55148943, 0.75270849],
       [0.94257081, 0.80801143, 0.84402915, 0.92103646, 0.41981399,
        0.85389998, 0.51312722, 0.27997811, 0.22653748, 0.81842454],
       [0.15548147, 0.84396866, 0.96774083, 0.57106997, 0.04774802,
        0.18454685, 0.24756512, 0.98110341, 0.48467907, 0.85443132],
       [0.34640233, 0.65298814, 0.94220599, 0.15015809, 0.83373935,
        0.4816276 , 0.52536268, 0.28132931, 0.15752363, 0.15153455],
       [0.3035823 , 0.24025296, 0.13960793, 0.95222501, 0.82242182,
        0.52460572, 0.37117484, 0.013054  , 0.4656086 , 0.44216443]])
```

[18]:
```
#Aleatorios de distribución normal con media cero
np.random.randn(20)
```

[18]:
```
array([-0.47902404, -0.61166857, -0.47379687, -2.37645838, -0.08846991,
       -1.6336327 ,  0.22404281,  1.34548819, -1.16407169,  0.09169541,
       -2.05781498,  0.38985184,  0.99627439, -0.48137949,  0.64084541,
       -0.04327927, -2.68133667, -0.93883853, -1.09113516,  1.70829157])
```

[19]:
```
#Aleatorios entre dos enteros (inferior, superior, tamaño)
np.random.randint(1,11,20)
```

[19]:
```
array([ 3,  3,  2,  1,  6,  5,  8,  5,  9,  4,  1,  2,  9,  9,  7,  4,  5,
        4,  4, 10])
```

[20]:
```
#También es muy útil establecer una semilla
np.random.seed(1234)
```

1.4 COMO TRANSFORMAR UN ARRAY

1.4.1 CAMBIAR LA FORMA Y ORDENAR

[21]:
```
#Ver la forma de una array
v = np.array([1,2,3])
v.shape
```

[21]: (3,)

[22]:
```
#Cambiar la forma de un array
v = np.arange(20)
print(v)
```

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

[23]:
```
v.shape
```

[23]: (20,)

[24]:
```
v2= v.reshape(2,10)
v2.shape
```

[24]: (2, 10)

[25]:
```
#Ordenar
v = np.random.rand(10)
print(v)
```

```
[0.19151945 0.62210877 0.43772774 0.78535858 0.77997581 0.27259261
 0.27646426 0.80187218 0.95813935 0.87593263]
```

[26]:
```
v.sort()
v
```

[26]:
```
array([0.19151945, 0.27259261, 0.27646426, 0.43772774, 0.62210877,
       0.77997581, 0.78535858, 0.80187218, 0.87593263, 0.95813935])
```

1.4.2 ESTADÍSTICOS BÁSICOS SOBRE UN ARRAY

[27]:
```
#Estadísticos
v = np.arange(20)
print(v.mean())
print(np.median(v))
print(np.std(v))
print(v.var())
```

```
print(v.max())
print(v.min())
print(np.corrcoef(v,v))
```

```
9.5
9.5
5.766281297335398
33.25
19
0
[[1. 1.]
 [1. 1.]]
```

[28]:
```
#Localizar los índices de los estadísticos
v = np.random.randint(1,11,10)
print(v)
print(v.argmax())
v[v.argmax()]
```

```
[ 9  1  6  1 10  7  3  1  6  3]
4
```

[28]:
```
10
```

1.4.3 CREAR COPIAS

[29]:
```
v1 = np.arange(5)
print(v1)
```

```
[0 1 2 3 4]
```

[30]:
```
v2 = v1
v2
```

[30]:
```
array([0, 1, 2, 3, 4])
```

[31]:
```
v2[:] = 5
```

[32]:
```
v2
```

[32]:
```
array([5, 5, 5, 5, 5])
```

[33]:
```
v1
```

[33]:
```
array([5, 5, 5, 5, 5])
```

[34]:
```
#Cuidado con las copias porque modificar una copia puede modificar también el␣
↪original
v1 = np.arange(5)
```

```
print(v1)
v2 = v1
v2[:] = 5
print(v2)
print(v1)
```

```
[0 1 2 3 4]
[5 5 5 5 5]
[5 5 5 5 5]
```

[35]:
```
#Para que no modifique el original hay que decir explícitamente que quieres una␣
 ↪copia
#Si no lo entenderá como un puntero al original (parar ahorrar memoria) y lo␣
 ↪podrá modificar
v1 = np.arange(5)
print(v1)
v2 = v1.copy()
v2[:] = 5
print(v2)
print(v1)
```

```
[0 1 2 3 4]
[5 5 5 5 5]
[0 1 2 3 4]
```

1.5 COMO INDEXAR UN ARRAY

[57]:
```
#Indexar vectores
v = np.arange(11)
print(v[2])
print(v[2:5])
```

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

[58]:
```
#Indexar matrices
#Creamos la matriz
m = np.random.rand(3,4)
print(m)
```

```
[[0.6087238  0.44409687 0.42970558 0.90596004]
 [0.97118351 0.21025827 0.52772137 0.1417399 ]
 [0.00734634 0.71959339 0.14999705 0.97912112]]
```

[59]:
```
#Indexar matrices
#Indexar celdas

#Opción 1: con dos corchetes, el primero para las filas y el segundo para las␣
 ↪columnas
```

```
print(m[0][0])

#Opción 2: con un solo corchete pero filas y columnas separadas por comas
print(m[0,0])
```

```
0.6087238025645857
0.6087238025645857
```

[60]:
```
#Indexar matrices
#Indexar filas o columnas

#Filas
print(m[0,:])

#Columnas
print(m[:,0])
```

```
[0.6087238  0.44409687 0.42970558 0.90596004]
[0.6087238  0.97118351 0.00734634]
```

[61]:
```
#Indexar matrices
#Indexar parcialmente filas o columnas

#Filas
print(m[0,0:2])

#Columnas
print(m[0:1,0])
```

```
[0.6087238  0.44409687]
[0.6087238]
```

[ ]:
```
#Indexar mediante vectores booleanos
#Las condiciones generan vectores booleanos
v = np.random.rand(10)
print(v)
print(v < 0.5)
```

[ ]:
```
#Indexar mediante vectores booleanos
#Usar una condición como índice filtrará directamente el vector
v = np.random.rand(10)
print(v)
v[v < 0.5]
```

[62]:
```
v = np.random.rand(10)
v
```

[62]:
```
array([0.36757995, 0.88682325, 0.19638186, 0.16533226, 0.48168762,
       0.19329721, 0.48693366, 0.28590213, 0.3207565 , 0.9855776 ])
```

[64]:
```
v_f = v < 0.5
```

[66]:
```
v[v_f]
```

[66]:
```
array([0.36757995, 0.19638186, 0.16533226, 0.48168762, 0.19329721,
       0.48693366, 0.28590213, 0.3207565 ])
```

1.6 REPASO Y A RECORDAR

- Numpy es la base del análisis de datos masivo y eficiente con Python
- Y la base de paquetes posteriores como Pandas
- Sobre todo hay que conocer su estructura base: el array multidimensional
- Podemos crear arrays a partir de listas, rangos, estructuras predefinidas, distribuciones aleatorias, etc
- Los indexamos con los mismos recursos que ya conocemos, poniendo énfasis en el tema de las copias

Como lectura complementaria (y voluntaria no necesaria) para consolidar y complementar lo que hemos aprendido te recomiendo el siguiente post:

https://medium.com/better-programming/numpy-illustrated-the-visual-guide-to-numpy-3b1d4976de1d

En el cual se refuerzan estos conceptos de una forma muy gráfica que puede ayudar a su consolidación.

02_IntensivoAlgebra

September 16, 2021

1 CURSO INTENSIVO DE INTRODUCCIÓN AL ÁLGEBRA LINEAL

[2]:
```
import numpy as np
```

1.1 INTRODUCCIÓN AL ÁLGEBRA LINEAL

Como decíamos al principio muchos de los algoritmos de machine learning se construyen con operaciones avanzadas sobre estructuras de datos como vectores o matrices.

Realmente no es necesario un conocimiento profundo de álgebra para hacer machine learning. Pero si es interesante por lo menos tener una nociones de lo que está pasando por debajo.

Y de nuevo el paquete base para hacer todo esto es Numpy, que seguiremos utilizando durante esta explicación.

Un poco de teoría: profundizando en la comprensión de las estructuras de datos

Hasta ahora hemos visto 3 estructuras:

- Escalares: en la lección de Python los llamábamos "datos individuales"
- Vectores: secuencias de varios escalares
- Matrices: secuencias de varios vectores organizados en dos dimensiones

Geométricamente los escalares son simplemente un punto y se dice que tienen dimensión cero.

Los vectores son una recta y tienen dimensión 1. De hecho pueden ser representados como matrices de dimensión 1, por ej m x 1 si son verticales o 1 x m si son horizontales.

Las matrices son un plano, y tienen dimensión 2, que se denota por m x n.

Y podemos seguir extendiendo a los arrays multidimensionales, que son hiperplanos con dimensión n.

[10]:
```
#Ejemplo de escalar con Numpy
np.array(1)
```

[10]:
```
array(1)
```

[11]:
```
#Ejemplo de vector con Numpy
np.array([1,2,3])
```

[11]:
```
array([1, 2, 3])
```

[12]:
```
#Ejemplo de matriz con Numpy
np.array([[1,2,3],[4,5,6]])
```

[12]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

[ ]:
```
#Ejemplo de array multidimensional con Numpy
np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
```

1.1.1 OPERACIONES CON VECTORES

SUMAR VECTORES La condición es que tienen que tener la misma longitud

[13]:
```
v1 = np.array([1,2,3])
v2 = np.array([4,5,6])

v1 + v2
```

[13]:
```
array([5, 7, 9])
```

RESTAR VECTORES La condición es que tienen que tener la misma longitud

[14]:
```
v1 - v2
```

[14]:
```
array([-3, -3, -3])
```

MULTIPLICAR UN VECTOR POR UN ESCALAR

[15]:
```
v1 * 10
```

[15]:
```
array([10, 20, 30])
```

MULTIPLICAR DOS VECTORES La condición es que tienen que tener la misma longitud

El tipo de multiplicación más usado es el producto escalar o punto producto: multiplicar uno a uno los correspondientes y luego sumar los resultados.

[16]:
```
v1
```

[16]:
```
array([1, 2, 3])
```

[17]:
```
v2
```

[17]:
```
array([4, 5, 6])
```

[18]:
```
np.dot(v1,v2)
```

[18]:
```
32
```

1.1.2 OPERACIONES CON MATRICES

[7]:
```
v3d = np.array([[[ 3, -4,  6],[ 0, -8,  3]],
                [[ 6, 5,  -4],[ 9, -1,  -2]]])
v
```

[7]:
```
array([[[ 3, -4,  6],
        [ 0, -8,  3]],

       [[ 6,  5, -4],
        [ 9, -1, -2]]])
```

[12]:
```
v[0,1,1]
```

[12]:
```
-8
```

SUMAR MATRICES La condición es que tienen que tener la misma forma

[9]:
```
m1 = np.array([[ 3, -4,  6],[ 0, -8,  3]])
m2 = np.array([[ -1, -9,  7],[ 4, 8,  -7]])

m1 + m2
```

[9]:
```
array([[  2, -13,  13],
       [  4,   0,  -4]])
```

RESTAR MATRICES La condición es que tienen que tener la misma forma

[23]:
```
m1 - m2
```

[23]:
```
array([[  4,   5,  -1],
       [ -4, -16,  10]])
```

MULTIPLICAR UNA MATRIZ POR UN ESCALAR

[24]:
```
m1 * 10
```

[24]:
```
array([[ 30, -40,  60],
       [  0, -80,  30]])
```

MULTIPLICAR DOS MATRICES Usaremos la forma del punto producto.

La condición es que el número de columnas de la primera tiene que ser el mismo que el número de filas de la segunda.

Más formalmente es multiplicar una m x n por una n x k.

Por ejemplo multiplicar una 3x2 por una 2x4.

Es así porque lo que hace el punto producto es ir multiplicando cada fila de la primera por cada columna de la segunda.

El resultado será una nueva matriz de m x k, ej 3x4.

Donde el puntoproducto de fila1 de la primera por columna 1 de la segunda será el resultado de la fila1,columna1 de la tercera.

[25]:
```
# Veamos la dimensión de m1
m1.shape
```

[25]:
```
(2, 3)
```

[26]:
```
m1
```

[26]:
```
array([[ 3, -4,  6],
       [ 0, -8,  3]])
```

[27]:
```
# Y creemos por ejemplo otra matriz 3x3
m3 = np.array([
    [8,6,2],
    [6,3,4],
    [2,-4,6]
])
m3.shape
```

[27]:
```
(3, 3)
```

[28]:
```
#Hacemos la multiplicación punto producto igual que con los vectores
np.dot(m1,m3)
```

[28]:
```
array([[ 12, -18,  26],
       [-42, -36, -14]])
```

APLANAR UNA MATRIZ Habrá ocasiones en machine learning que aunque tengamos que usar una matriz el formato en el que tenemos que pasarle esa matriz a la clase que la procesará es un array unidimensional.

Aplanar una matriz (flatten en inglés) significa eso, pasarla de dimensión 2 a dimensión 1, simplemente concatenando todas sus filas en una sola.

Lo hacemos con el método flatten de Numpy.

[29]:
```
#Recordemos como es la matriz m3
m3
```

[29]:
```
array([[ 8,  6,  2],
       [ 6,  3,  4],
       [ 2, -4,  6]])
```

[30]:
```
#Así queda al aplanarla
m3.flatten()
```

[30]:
```
array([ 8,  6,  2,  6,  3,  4,  2, -4,  6])
```

1.1.3 PARA QUÉ SIRVE TODO ESTO

Muchas de las operaciones en ML se pueden hacer mediante álgebra matricial.

Y de hecho hacerlas asi es más eficiente computacionalmente.

Por lo que los algoritmos ya las hacen asi aunque nosotros no seamos conscientes ni tengamos que hacer nada al respecto.

Deep learning:

Deep learning es un subcampo de machine learning basado en la evolución de un algoritmo concreto llamado redes neuronales.

Es un campo en si mismo, bastante complejo y que solo se debe abordar una vez que se tengan muy trabajadas las bases generales de todo lo que vemos en este programa. Por tanto no entraremos en él.

Pero si más o menos estás en el ámbito seguro que has oído hablar por ejemplo de TensorFlow (el framework de deep learning de Google).

Pues bien, se llama así porque los tensores son la base de deep learning.

Pero realmente los tensores no son más que la generalización de lo que hemo visto:

- Un escalar es un tensor de rango 0
- Un vector es un tensor de rango 1
- Una matriz es un tensor de rango 2
- Puede haber tensores de muchas más dimensiones, por ej uno de rango 3 sería una colección de matrices

En Python los tensores se gestionan con nd.arrays.

[35]:
```
# Ejemplo de tensor de rango 3

m1 = np.array([[ 3, -4,  6],[ 0, -8,  3]])
m2 = np.array([[ -1, -9,  7],[ 4, 8,  -7]])

tensor = np.array([m1,m2])
tensor
```

[35]:
```
array([[[ 3, -4,  6],
        [ 0, -8,  3]],

       [[-1, -9,  7],
        [ 4,  8, -7]]])
```

[36]:
```
# También se podría haber creado directamente así

tensor = np.array([[[ 3, -4,  6],[ 0, -8,  3]],[[ -1, -9,  7],[ 4, 8,  -7]]])
tensor
```

[36]:
```
array([[[ 3, -4,  6],
        [ 0, -8,  3]],

       [[-1, -9,  7],
        [ 4,  8, -7]]])
```

[37]:
```
# Vemos que retorna una colección de 2 matrices cada una de 2x3
tensor.shape
```

[37]:
```
(2, 2, 3)
```

Pongamos el ejemplo de **Computer Vision:**

Especialmente en Deep Learning se trabaja mucho con imágenes. Pero un ordenador no ve las imágenes como nosotros.

Cada imagen es en realidad un conjunto de pixeles, por ej de 1000x1000, es decir una matriz, o un tensor de rango 2.

En una imagen en blanco y negro cada celda sería un número entre 0 y 255 que denota la escala de gris. Siendo 0 blanco puro y 255 negro puro.

Una forma de incluir color es usar las escalas RGB, donde cualquier color se puede representar como una combinación de esos 3 colores.

Por tanto tendríamos una matriz de 1000x1000 por cada uno de esos 3 canales, es decir 1000x1000x3 o un tensor de rango 3.

Y así es como podemos representar una imagen con álgebra matricial y aplicarle machine learning.

Por ejemplo vamos a poner un caso simplificado donde la imagen fuera solo de 5x5 pixeles.

Usando una función de numpy para generar aleatorios entre 0 y 255 así es como el ordendador "vería" esa imagen.

[41]:
```
R = np.random.randint(0,255,(5,5))
G = np.random.randint(0,255,(5,5))
B = np.random.randint(0,255,(5,5))

imagen = np.array([R,G,B])
imagen
```

[41]:
```
array([[[ 11,  83, 171, 159,  98],
        [ 93,  90,  58, 116,   7],
        [ 13, 225, 166, 162, 186],
        [ 23,  88,  19, 232, 163],
        [ 54, 129, 159, 165, 103]],

       [[ 27,  88, 224, 108,   1],
        [ 91,  30, 168, 192, 101],
        [130, 133, 165,  75,  41],
        [130,  63, 187, 208,  71],
```

```
         [195, 160, 153, 241, 140]],

        [[ 94,  42, 127, 229,  18],
         [ 70, 142, 120, 251,  16],
         [100, 199, 141, 163,  32],
         [214, 238, 173,  60,  49],
         [239, 253, 162, 109, 136]]])
```

[38]:
```
R = np.random.randint(0,255,(5,5))
R
```

[38]:
```
array([[106, 168, 138, 107, 151],
       [176, 131,   7, 163, 202],
       [243, 166,   3, 139, 195],
       [114, 175,   2,  77, 136],
       [ 91,  40, 254, 132,  87]])
```

[39]:
```
G = np.random.randint(0,255,(5,5))
G
```

[39]:
```
array([[  3, 146,  72, 179, 173],
       [172, 147,  85,  10, 216],
       [ 20, 115, 153, 211,  78],
       [ 37, 104, 130, 106, 152],
       [ 79, 236, 230, 160, 251]])
```

[40]:
```
B = np.random.randint(0,255,(5,5))
B
```

[40]:
```
array([[ 33, 140, 113,  84,  66],
       [250,  68, 245, 228, 151],
       [140,  37,  60,   7,  76],
       [226,  40,   9,  10,  61],
       [ 36, 163,  23,  38, 185]])
```

[ ]:

[ ]:

1.2 PARA SABER MÁS

Como lectura complementaria (y voluntaria no necesaria) para consolidar y complementar lo que hemos aprendido te recomiendo el siguiente post:

https://medium.com/better-programming/numpy-illustrated-the-visual-guide-to-numpy-3b1d4976de1d

En el cual se refuerzan estos conceptos de una forma muy gráfica que puede ayudar a su consolidación.

[ ]:

04_Numpy_Algebra_Ejercicios_Soluciones

September 16, 2021

1 INTENSIVOS DE NUMPY Y ALGEBRA: EJERCICIOS

Estos ejercicios te permitirán repasar y consolidar lo que has aprendido en los intensivos de Numpy y Álgebra.

No hagas los ejercicios hasta que hayas estudiado y memorizado el contenido de sus notebooks.

Instrucciones:

- Crea una copia de este notebook antes de escribir nada. De esa forma respetarás el original para volver a hacerlo
- Cada ejercicio tiene unas instrucciones de lo que debes hacer en él
- Bajo las instrucciones estará una celda vacía, aquí es donde tú debes escribir la solución
- Y bajo esta última celda verás que hay otras dos celdas. La primera está vacía y no la debes tocar. Es para que tengas siempre a mano la solución al ejercicio, ya que esta no va a cambiar

1.1 Ejercicio:

Importa numpy como np

[45]:
```
import numpy as np
```

1.2 Ejercicio:

Crea una lista de Python con los números del 1 a l0. Y después suma 5 a cada uno de sus elementos.

[46]:
```
lista = list(range(1,11))
[cada + 5 for cada in lista]
```

[46]:
```
[6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

1.3 Ejercicio:

Haz lo mismo pero ahora usando numpy (primero crea un vector a partir de lista).

[48]:
```
vector = np.array(lista)
vector + 5
```

[48]:
```
array([ 6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
```

1.4 Ejercicio:

Crea un array directamente desde un rango, desde el 100 al 200 de 10 en 10.

[49]:
```
np.arange(100,201,10)
```

[49]:
```
array([100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200])
```

1.5 Ejercicio:

Crea una array de 20 elementos interpolados desde el 100 al 200.

[50]:
```
np.linspace(100,200,20)
```

[50]:
```
array([100.        , 105.26315789, 110.52631579, 115.78947368,
       121.05263158, 126.31578947, 131.57894737, 136.84210526,
       142.10526316, 147.36842105, 152.63157895, 157.89473684,
       163.15789474, 168.42105263, 173.68421053, 178.94736842,
       184.21052632, 189.47368421, 194.73684211, 200.        ])
```

1.6 Ejercicio:

Crea una matriz de 4 filas y 6 columnas rellena con ceros.

[51]:
```
np.zeros((4,6))
```

[51]:
```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

1.7 Ejercicio:

Crea una matriz identidad de 4 filas y 4 columnas.

[67]:
```
np.eye(4,4)
```

[67]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

1.8 Ejercicio:

Crea una matriz de 6 filas y 3 columnas rellena con números aleatorios entre 0 y 1.

[53]:
```
np.random.rand(6,3)
```

[53]:
```
array([[0.09451382, 0.16584694, 0.2529427 ],
       [0.28996585, 0.46437665, 0.43470393],
```

```
       [0.46521979, 0.27674738, 0.07856732],
       [0.13936708, 0.82925581, 0.39132827],
       [0.43577272, 0.32297518, 0.06138335],
       [0.27166362, 0.54577411, 0.28245301]])
```

1.9 Ejercicio:

Crea un vector aleatorio de 100 elementos que tenga una distribución normal.

[54]:
```
np.random.randn(100)
```

[54]:
```
array([ 0.12834816,  0.42524468,  0.21095275,  0.7218104 ,  0.09394681,
        0.4642257 , -0.11891562,  0.61751569,  0.75489449, -2.30931096,
       -1.75704714,  0.89818773,  0.15026191, -1.06121246, -0.39216193,
       -1.26625334, -0.64986291, -1.48819816,  0.02671956,  0.09407852,
       -0.38231459,  0.20115651, -0.03886299,  0.6251126 ,  0.80682781,
       -1.19814968, -0.50837248,  0.68435471,  0.71324418,  1.74594028,
        0.78351515, -1.50249539, -0.55837808,  0.61603342, -0.79127258,
        0.30382751, -0.45963241, -0.76974453, -0.45983724, -0.93955482,
        0.84061529,  0.95448884, -0.83919377,  0.52608214,  3.31969621,
       -0.768641  , -0.13039133,  0.31945984,  0.98480205, -0.41329785,
       -0.86605547,  1.86406004, -1.1478997 ,  2.46526731, -1.49621034,
        0.01154948,  1.64698209,  1.29050885,  1.12787419,  2.14417859,
       -1.91519325,  0.60085336, -0.66306722, -0.26413746,  0.03949464,
        0.7090535 , -0.21414052,  0.55457442, -0.35212701, -0.04521234,
        1.10325213, -1.17100427,  0.24712849,  0.29264046, -0.81451885,
        2.35411216, -0.67790035, -1.31458784, -1.02067448,  0.08417335,
        2.10617362,  0.91360029, -0.36452606,  0.8948383 , -0.31088983,
       -1.3595482 , -1.7689971 , -0.04635292,  0.8215378 , -0.6086671 ,
        0.97958821, -0.45031162, -0.43203875,  1.57961267, -0.38254992,
       -0.39133306, -2.28696988,  1.86855813, -0.83737734, -0.02799622])
```

1.10 Ejercicio:

Crea un vector aleatorio de 100 elementos que vayan entre el 1 y el 100.

[55]:
```
np.random.randint(1,101,100)
```

[55]:
```
array([ 13,  90,  91,  42,   3,  61,  81,  65,  66,  81,  36,  81,  28,
        50,  46,  69,  35,  95,  96,  21,  48,  96,   7,  40,  18,  12,
        25,  12,  38,  72,  91,  34,  66,  51,  95,   7,   1,  92,   6,
        28,  92,  15,  48,  18,  58,  24,  15,  21,  93,  28,  70,  71,
         8,  22,  54,  35,  46,  49,  78,  28,  94,  46,  73,  28,  94,
        21,  17,  54,  77,  39,  66,  51,  40,  35, 100,  90,  60,  46,
        23,  59,  56,  26,  30,  72,  86,  99,  33,  92,  50, 100,  91,
        35,  58,  96,  37,   8,  35,  54,  24,  31])
```

1.11 Ejercicio:

Crea un vector llamado vector, como el anterior pero ordénalo.

[56]:
```
vector = np.random.randint(1,101,100)
vector.sort()
vector
```

[56]:
```
array([ 1,  1,  1,  2,  2,  3,  3,  5,  5,  7,  8,  9, 11, 12, 17, 17, 19,
       19, 20, 20, 23, 24, 26, 26, 26, 27, 27, 28, 29, 32, 32, 33, 34, 35,
       35, 36, 40, 40, 40, 40, 45, 46, 50, 52, 53, 53, 55, 55, 55, 56, 56,
       57, 57, 57, 58, 59, 60, 61, 62, 63, 65, 66, 66, 66, 67, 68, 70, 71,
       71, 72, 73, 74, 74, 78, 78, 79, 79, 81, 81, 82, 82, 85, 85, 85, 85,
       86, 86, 87, 87, 88, 89, 91, 92, 93, 94, 95, 95, 95, 96, 98])
```

1.12 Ejercicio:

Sobre vector cacula la media, mediana, desv típica, varianza, máximo y mínimo.

[57]:
```
print(vector.mean())
print(np.median(vector))
print(np.std(vector))
print(vector.var())
print(vector.max())
print(vector.min())
```

```
51.8
56.0
29.27422074112307
856.9800000000001
98
1
```

1.13 Ejercicio:

Extrae de vector los elementos que están entre el sexto y el octavo ambos incluidos.

[58]:
```
vector[5:8]
```

[58]:
```
array([3, 3, 5])
```

1.14 Ejercicio:

Extrae de vector los elementos que son pares. Pista: puedes apoyarte en el operador módulo %

[59]:
```
vector[vector % 2 == 0]
```

[59]:
```
array([ 2,  2,  8, 12, 20, 20, 24, 26, 26, 26, 28, 32, 32, 34, 36, 40, 40,
       40, 40, 46, 50, 52, 56, 56, 58, 60, 62, 66, 66, 66, 68, 70, 72, 74,
       74, 78, 78, 82, 82, 86, 86, 88, 92, 94, 96, 98])
```

1.15 Ejercicio:

Crea dos vectores de enteros llamados v1 y v2, cada uno de 5 elementos aleatorios entre 1 y 10 y súmalos.

[60]:
```
v1 = np.random.randint(1,11,5)
v2 = np.random.randint(1,11,5)

v1 + v2
```

[60]:
```
array([10, 17, 10, 16, 15])
```

1.16 Ejercicio:

Multiplícalos usando el producto escalar.

[61]:
```
np.dot(v1,v2)
```

[61]:
```
215
```

1.17 Ejercicio:

Transforma el vector que creaste antes llamado vector en una matriz de 10 x 10 y llámala matriz

[62]:
```
matriz = vector.reshape(10,10)
matriz
```

[62]:
```
array([[ 1,  1,  1,  2,  2,  3,  3,  5,  5,  7],
       [ 8,  9, 11, 12, 17, 17, 19, 19, 20, 20],
       [23, 24, 26, 26, 26, 27, 27, 28, 29, 32],
       [32, 33, 34, 35, 35, 36, 40, 40, 40, 40],
       [45, 46, 50, 52, 53, 53, 55, 55, 55, 56],
       [56, 57, 57, 57, 58, 59, 60, 61, 62, 63],
       [65, 66, 66, 66, 67, 68, 70, 71, 71, 72],
       [73, 74, 74, 78, 78, 79, 79, 81, 81, 82],
       [82, 85, 85, 85, 85, 86, 86, 87, 87, 88],
       [89, 91, 92, 93, 94, 95, 95, 95, 96, 98]])
```

1.18 Ejercicio:

Multiplica matriz por 8

[63]:
```
matriz * 8
```

[63]:
```
array([[  8,   8,   8,  16,  16,  24,  24,  40,  40,  56],
       [ 64,  72,  88,  96, 136, 136, 152, 152, 160, 160],
       [184, 192, 208, 208, 208, 216, 216, 224, 232, 256],
       [256, 264, 272, 280, 280, 288, 320, 320, 320, 320],
       [360, 368, 400, 416, 424, 424, 440, 440, 440, 448],
       [448, 456, 456, 456, 464, 472, 480, 488, 496, 504],
```

```
[520, 528, 528, 528, 536, 544, 560, 568, 568, 576],
[584, 592, 592, 624, 624, 632, 632, 648, 648, 656],
[656, 680, 680, 680, 680, 688, 688, 696, 696, 704],
[712, 728, 736, 744, 752, 760, 760, 760, 768, 784]])
```

01_Estadistica_I_Descriptiva

September 16, 2021

Contenido

1 ESTADÍSTICA I PARA DATA SCIENCE: DESCRIPTIVA

1.1 INSTALACIÓN

Vamos a instalar los paquetes que todavía no tenemos.

Este paso sólo tendrás que hacerlo la primera vez.

Después te dejo que lo dejes comentado.

[5]:
```
# conda install pandas
# conda install scipy
# conda install seaborn
# conda install -c conda-forge statsmodels
```

1.2 PREPARACION

[1]:
```
#Carga de paquetes y datos
import pandas as pd
import numpy as np
import statistics
import scipy as sp
import seaborn as sns
import random
import math
from statsmodels.stats.proportion import proportions_ztest

df = sns.load_dataset('tips')
```

[3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   total_bill  244 non-null    float64
 1   tip         244 non-null    float64
 2   sex         244 non-null    category
 3   smoker      244 non-null    category
 4   day         244 non-null    category
 5   time        244 non-null    category
 6   size        244 non-null    int64
dtypes: category(4), float64(2), int64(1)
memory usage: 7.4 KB
```

[7]:
```
df.head()
```

[7]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

1.3 CONCEPTOS BÁSICOS

1.3.1 Descriptiva Vs Inferencial

Son las 2 grandes ramas de la estadística.

Y por tanto así también vamos a organizar este módulo.

La **DESCRIPTIVA** se centra en describir los datos que estamos analizando. Resumirlos, visualizarlos y sacar conclusiones.

Es la que más se usa en la práctica, y son el tipo de operaciones más básicas e imprescindibles que vamos a estar haciendo en Data Science.

Simplemente usando las técnicas que vamos a aprender en este apartado ya podremos hacer el 80% de los análisis que normalmente se hacen en contextos empresariales.

La **INFERENCIAL** se pregunta "vale, pero ¿esas conclusiones que hemos detectado en la descriptiva realmente son robustas y estables y pueden ser extrapoladas?".

Parte del concepto teórico de que realmente siempre vamos a estar analizando muestras, y no toda la población (más sobre esto en el siguiente apartado).

Por lo que son necesarias técnicas que nos garanticen estadísticamente que esas conclusiones no son fruto del azar o de nuestra muestra en concreto.

Esta parte está muy infrautilizada en la práctica del día a día, ya que pocas veces se usan estas técnicas para contrastar si los resultados de un análisis, estudio o informe son realmente estadísticamente significativos.

Sin embargo como Data Scientist, conocerlas te va a servir tanto para garantizar que la calidad de tu trabajo es la correcta como para poder identificar falsas conclusiones que te intenten "colar".

Ahora bien, debes saber un hecho importante que afecta a todo lo que veamos en la parte inferencial.

Todas las técnicas y herramientas que veremos están basadas en que la muestra ha sido obtenida ALEATORIAMENTE y sin ningún tipo de sesgo.

Para garantizar esto la teoría nos dice que debemos hacer un correcto "diseño de experimento" que en la empresa lógicamente no se puede hacer casi nunca.

Así te corresponde como analista el evaluar el grado de "aleatoridad" y no sesgo que tienen tus datos y hasta qué punto puedes apoyarte en los recursos de la estadística inferencial.

1.3.2 Población Vs Muestra

Población:

Cuando trabajamos con todos los elementos de interés.

Se denota por N.

Los números que la describen se llaman parámetros.

Muestra:

Cuando trabajamos con algunos de los elementos de interés.

Se denota por n.

Los números que la describen se llaman estadísticos o estimadores.

1.3.3 Escalas de medida y tipos de variables

Este es uno de los puntos más importantes que debes conocer y sin embargo pocas veces es tratado en las formaciones de Data Science.

La estadística nos dice que hay 4 grandes escalas de medida:

1. **Nominal**: Tiene la propiedad de identidad (igual/diferente) pero NO la de orden. Ejemplos: sexo, colores, …
2. **Ordinal**: Añade la propiedad de orden, pero NO tiene una igual distancia entre elementos. Ejemplos: clase social, nivel formativo, …
3. **Intervalo (o discreta)**: Añade la propiedad de igualdad de distancias, pero no permite infinitos valores entre sus elementos. Ejemplos: número de hijos, número de siniestros, …
4. **Razón (o continua)**: Puede tener (teóricamente) infinitos valores entre sus elementos. Ejemplos: precio de un producto, distancias, …

A partir de las escalas se derivan los tipos de variables, que pueden ser diferente en cada implementación (por ej en cada lenguaje de programación), pero casi siempre podremos hacer una asignación aproximada a una escala.

Por ejemplo los enteros serían variables discretas, los reales continuas, los objetos categóricas, etc.

A un nivel más elevado podemos agruparlos en dos grupos que son super relevantes, ya que estos dos grupos van a determinar qué técnicas analíticas o qué gráficos podemos aplicar en cada uno de ellos:

- **Categóricas (o cualitativas)**: incluyen nominales y ordinales
- **Numéricas (o cuantitativas)**: incluyen discretas y razón

Como este curso tiene un caracter eminentemente aplicado vamos a usar esta división para ver qué tipo de análisis podemos hacer en cada una.

1.4 ESTADÍSTICA DESCRIPTIVA

Como ya habíamos anticipado la misión de la descriptiva es resumir, graficar y analizar los datos para intentar encontrar patrones de interés.

A grandes rasgos tenemos dos tipos de técnicas a nuestra disposición:

- **Cálculos**: aplicar algún tipo de estadístico o análisis y obtener uno o varios resultados numéricos (indicadores, tablas, porcentajes, etc.)
- **Gráficos**: resumir y mostrar la información de forma gráfica para detectar los patrones más fácilmente

Y también como decíamos más arriba el tipo de cálculos y gráficos que podemos hacer van a depender directamente del tipo de variable. Así que los vamos a dividir entre los de variables categóricas y los de variables cuantitativas.

1.4.1 Análisis de variables categóricas

Con cálculos

Conteos y frecuencias Se entiende por frecuencia el conteo del valor de cada variable.

[4]:
```
df.smoker.value_counts()
```

[4]:
```
No     151
Yes     93
Name: smoker, dtype: int64
```

Moda Es el valor más frecuente de una variable.

[6]:
```
#Moda
numeros = np.random.randint(0,11,1000)
print(pd.Series(numeros).value_counts())
statistics.mode(numeros)
```

```
7     104
0     101
5      97
8      95
9      94
3      88
4      87
10     87
2      84
1      83
6      80
dtype: int64
```

[6]:
```
7
```

Tablas cruzadas Consiste en cruzar dos o más variables para analizar la frecuencia.

Suelen usarse o en absoluto o en porcentaje.

[7]:
```
#En absoluto
pd.crosstab(df.sex,df.smoker,margins = True)
```

[7]:

smoker	Yes	No	All
sex			
Male	60	97	157
Female	33	54	87
All	93	151	244

[11]:
```
#En porcentaje
pd.crosstab(df.sex,df.smoker,margins = True,normalize = 'all')
```

[11]:

smoker	Yes	No	All
sex			
Male	0.245902	0.397541	0.643443
Female	0.135246	0.221311	0.356557
All	0.381148	0.618852	1.000000

Chi Cuadrado Al hacer la tabla cruzada hemos visto que hay diferencias en si es fumador o no en base a si es hombre o mujer.

Pero, ¿estas diferencias son suficientes para tomar el resultado como una conclusión?. O quizá están dentro del margen esperable por efecto del muestreo y por tanto no podemos concluir nada.

Es lo que se llama ver si una conclusión es estadísticamente significativa. Y veremos el concepto en profunidad en la parte de estadística inferencial.

Pero en un análisis de tablas cruzadas como este podemos hacerlo con el estadístico chi-cuadrado.

Que parte de una hipótesis nula de que no hay relación entre las variables, es decir que son independientes y por tanto fumar o no, no depende del sexo, y la contrasta contra los datos obtenidos.

Si el pvalor es menor o igual que 0.05 entonces rechaza esa hipótesis nula, y significa que los datos si apoyan que hay diferencias significativas y por tanto podemos concluir que sí hay diferencias en cuanto al fumar entre hombres y mujeres.

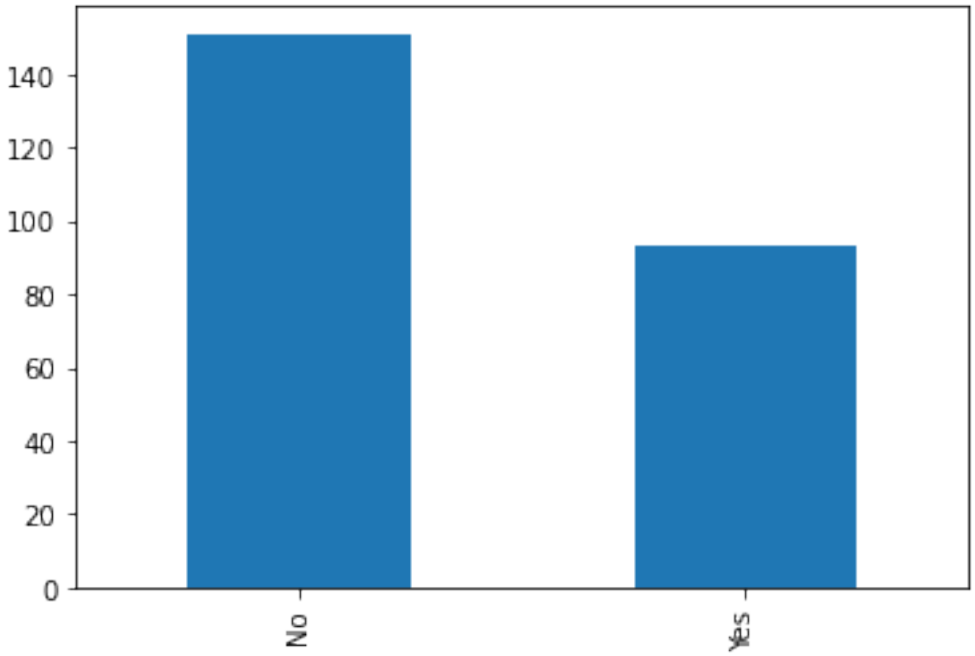
[8]:
```
#Chi-cuadrado
tabla = pd.crosstab(df.sex,df.smoker)
chi, pvalor, gl, experado = sp.stats.chi2_contingency(tabla)
print(chi)
print(pvalor)
#En este caso el valor es mayor que 0.05, por tanto no podemos rechazar la␣
 ↪hipótesis nula,
#y por tanto no hay diferencias significativas entre hombres y mujeres
```

```
0.008763290531773594
0.925417020494423
```

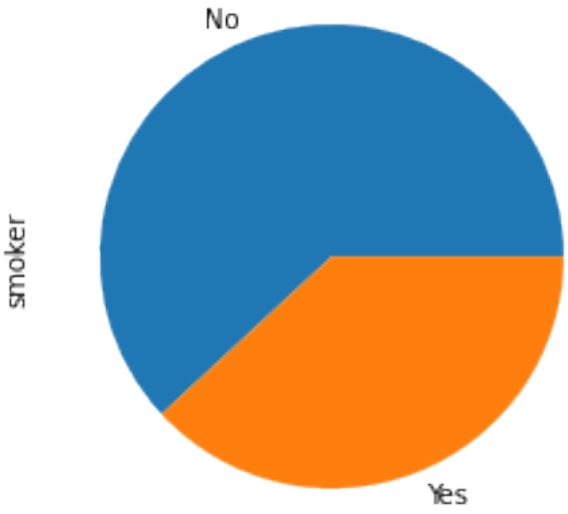
Con gráficos Los gráficos más frecuentes que podremos hacer con variables categóricas son:

- Gráficos de barras: en vertical, horizonal, apilados, etc.
- Gráficos de sectores (o de tarta)

[13]:
```
#Ejemplo de gráfico de barras
df.smoker.value_counts().plot(kind = 'bar');
```



[14]:
```
#Ejemplo de gráfico de sectores
df.smoker.value_counts().plot(kind = 'pie');
```



1.4.2 Análisis de variables cuantitativas

Con cálculos Se diferencian dos tipos de medida:

- De **centralización**: que intentan resumir la información de la variable en un sólo dato que pueda ser lo más representativo posible. Aquí están medidas como las medias, mediana o moda.
- De **dispersión**: que representan el grado de variabilidad existente en la variable. Normalmente cuanto más baja sea la medida de dispersión más representativa será la medida de centralización. Aquí entran métricas como la varianza, la desviación típica o el coeficiente de variación.

También veremos las métricas para cuantificar la relación existente entre dos variables cuantitativas: las correlaciones.

Medias La media normal se llama **media aritmétrica**, y será la que usaremos en la mayoría de los casos.

La **media winsorizada** se usa cuando hay valores atípicos que pueden sesgar el valor de la media.

Para hacer la media de porcentajes o tasas hay que usar la **media geométrica**.

Para hacer la media de medias hay que usar la **media armónica**.

[21]:
```
#Media arimética
statistics.mean([3,4,5])
```

[21]:
```
4
```

[17]:
```
#Media winsorizada
numeros = np.array([3,4,5,6,999997])
print(numeros)
print(numeros.mean())

#Winsorizar sustituye los valores fuera de los límites por el último valor
winsorizados = sp.stats.mstats.winsorize(numeros, limits = [0, 0.2])
print(winsorizados)
print(winsorizados.mean())
```

```
[     3      4      5      6 999997]
200003.0
[3 4 5 6 6]
4.8
```

[23]:
```
#Media geométrica (para porcentajes)
statistics.geometric_mean([0.3,0.4,0.5])
```

[23]:
```
0.3914867641168864
```

[24]:
```
#Comparamos con lo que hubiera salido en una media arimética
statistics.mean([0.3,0.4,0.5])
```

[24]: 0.4

[25]:
```
#Media armónica (para media de medias)
#Suponemos que los datos de esta lista son medias, ej velocidad media de coches␣
 ↪en 3 calles
statistics.harmonic_mean([30.3,40.8,50.1])
```

[25]: 38.72307605739473

[26]:
```
#Comparamos con lo que hubiera salido en una media arimética
statistics.mean([30.3,40.8,50.1])
```

[26]: 40.4

Mediana Si se ordenan todos los valores de la variable en orden ascendente o descendente la medianta es el valor de la variable correpondiente al elemento que ocupa la posición central, es decir, el que está en el 50%.

La mediana es una medida de centralización más recomendable que la media cuando tenemos distribuciones que no son normales (más sobre esto cuando veamos la inferencial), o cuando tenemos atípicos.

[29]:
```
#Mediana
numeros = np.random.randint(0,11,11)
numeros_ord = np.sort(numeros)
print(numeros_ord)
print(statistics.median(numeros_ord))
```

```
[ 1  2  2  4  4  5  6  6  7  7 10]
5
```

Varianza La varianza es el resultado de restar la media a cada valor de la variable, elevarlo al cuadrado (para evitar los negativos), sumarlo todo, y dividir el resultado por el número de datos.

Es una medida de dispersión, porque será mayor cuanto más lejos estén el global de los valores con respecto a la media.

[2]:
```
#Vamos a calcularla manualmente para entenderla
var1 = np.random.randint(0,11,20)
print(var1)
media = var1.mean()
print(media)
suma_cuadrados = sum((var1 - media) ** 2)
print(suma_cuadrados)
varianza = suma_cuadrados / (len(var1))
print(varianza)
```

```
[ 3  8  2  4  0 10  4  1  5 10  1  8  6  7  1  1  5  0 10  6]
4.6
```

```
224.79999999999998
11.239999999999998
```

[39]:
```
#En la práctica usaremos una función para calcularla directamente
var1.var()
```

[39]:
```
9.790000000000003
```

Desviación típica El problema con la varianza es que hemos tenido que elevar al cuadrado para quitarnos los negativos.

Entonces el dato obtenido ya no está en la misma escala que la media por ejemplo para poder compararlos.

Como solución se hace la raiz cuadrada a la varianza para volver a traerla a la escala, y es lo que se llama desviación típica.

[42]:
```
#Desviación típica
var1.std()
```

[42]:
```
3.1288975694324037
```

Coeficiente de variación Como decíamos más arriba cuando más baja sea la dispersión con respecto a la media será mejor porque nos indica que esa media es más representativa del total de los datos de la variable.

Ahora que ya tenemos una medida, la desviación típica, que sí está en la misma escala que la media, podemos compararlas.

Es lo que se llama el coeficiente de variación, y consiste en dividir la desviación típica por la media. Normalmente se pone en porcentaje.

Permite comparar entre variables de diferentes escalas, por ejemplo: ¿en qué somos más variables las personas: en la altura o en el peso?

[43]:
```
#Coeficiente de variación
var1.std() / var1.mean() * 100
```

[43]:
```
63.85505243739599
```

Correlación de Pearson Cuando la gente habla de correlación para medir la relación entre dos variables se está refiriendo a Pearson.

Sólo sería técnicamente correcto usar Pearson con variables contínuas y linealmente relacionadas, y si estas tienen una distribución normal.

Pero si las variables no son contínuas o normalmente distribuídas, o su relación es no lineal usar Pearson no es lo más correcto. Podríamos usar Spearman como veremos más abajo.

Además la correlación de Pearson es una medida lineal, y por tanto no válida para medir relaciones no lineales.

Ello significa que si dos variables aparecen como correlacionadas según Pearson efectivamente tendrán relación. Pero lo contrario no es siempre cierto, ya que podría haber relación pero ser no lineal.

La forma más fácil para identificar correlaciones no lineales es usar un gráfico de dispersión como veremos más adelante.

La correlación de Pearson se interpreta así:

- +1 o -1: es una correlación perfecta (siempre que sube una la otra también o siempre que sube una la otra baja)
- 0: no hay relación entre las variables
- Si el pvalor está por debajo de 0.05 la relación sí es significativa (se recomienda considerarlo sólo para más de 500 datos)

[44]:
```
#Correlación de Pearson
#Esta implementación de scipy devuelve la correlación como primer valor y el␣
 ↪pvalor como el segundo
var1 = np.random.randint(1,21,1000)
var2 = np.random.randint(1,21,1000)

sp.stats.pearsonr(var1,var2)
```

[44]:
```
(-0.021503831721390886, 0.4969865011347734)
```

R cuadrado R cuadrado es simplemente el cuadrado de la correlación de Pearson.

Pero sin embargo es una métrica muy útil en modelización, ya que se puede entender como el porcentaje de una variable que podemos explicar a partir de otra (o combinación de otras como en los modelos lineales).

Por tanto será frecuente verla en modelos como las regresiones, donde el software nos reportará por ejemplo un R cuadrado de 0.6 y eso significa que con nuestro modelo estamos siendo capaces de explicar el 60% de la variable objetivo, y por tanto nos queda un 40% que no sabemos explicar.

Al igual que la correlación de Pearson es una medida lineal, por tanto si cuando estamos modelizando con una técnica lineal nos sale un R cuadrado bajo puede ser interesante probar una técnica no lineal para ver si hay relaciones no lineales que no estaban siendo capturadas.

Veremos todo esto en la parte de modelización del programa.

[46]:
```
#R cuadrado
#Esta implementación de scipy devuelve la correlación como primer valor y el␣
 ↪pvalor como el segundo
(sp.stats.pearsonr(var1,var2)[0] ** 2) * 100
```

[46]:
```
0.04624147787018969
```

Correlación de Spearman La mayoría de la gente conoce la correlación de Pearson para medir la relación entre dos variables.

Pero realmente sólo es correcto usar Pearson con variables contínuas, linealmente relacionadas y si estas tienen una distribución normal.

Pero si las variables no son contínuas o no son normalmente distribuídas usar Pearson no es lo más correcto.

La **Rho de Spearman** proporciona una solución para hacer la correlación cuando:

- las variables son contínuas pero no se distribuyen según la normal
- las variables son rankings
- la relación es no lineal
- las variables son discretas pero con menos de 30 valores distintos

Se interpreta así:

- +1 o -1: es una relación perfectamente monotónica (siempre que sube una la otra también o al revés)
- 0: no hay relación entre las variables
- Si el pvalor está por debajo de 0.05% la relación sí es significativa (se recomienda considerarlo sólo para más de 500 datos)

[47]:
```
#Correlación de Spearman
var1 = np.random.randint(1,21,1000)
var2 = np.random.randint(1,21,1000)

sp.stats.spearmanr(var1,var2)
```

[47]:
```
SpearmanrResult(correlation=0.050494394339809144, pvalue=0.11053482196961902)
```

Correlación Vs Causalidad Este es uno de los errores más frecuentes en el uso cotidiano de la estadística.

Se tiende a pensar que si dos variables están correlacionadas entonces se puede establecer una relación causa - efecto entre ellas.

Realmente, para poder establacer una relación causa - efecto se tiene que cumplir que:

1. Exista correlación
2. La causa preceda en el tiempo a la consecuencia
3. Se puedan descartar explicaciones alternativas

Aquí entran los conceptos de correlación espúrea y correlación parcial.

Correlación espúrea es cuando dos variables parece que están relacionadas (pueden correlacionar de hecho), pero es realmente por el efecto de otras terceras variables no consideradas.

Correlación parcial es la correlación real que queda entre las 2 primeras variables una vez que se elimina el efecto de la tercera o terceras.

Por ejemplo, está demostrado que existe alta correlación entre el tamaño del pie y el cociente de inteligencia.

Sin embargo esa correlación está marcada realmente por una tercera variable, la edad. Si se controla la edad y se elimina su efecto entonces ya no existe correlación entre el tamaño del pie y el cociente

de inteligencia.

En contextos empresariales está plagado de este tipo de efectos y es conveniente conocer este concepto y aplicar siempre la visión crítica antes de obtener conclusiones.

Ejemplos:

- No suben las ventas cuando hay promociones: ¿se han controlado las promociones de la competencia?
- Los empleados que viajan dejan más la empresa: ¿se ha controlado la edad?
- Los clientes que más ganan sin embargo tienen menos ahorros: ¿se ha diferenciado entre los que tienen hipoteca y los que no?

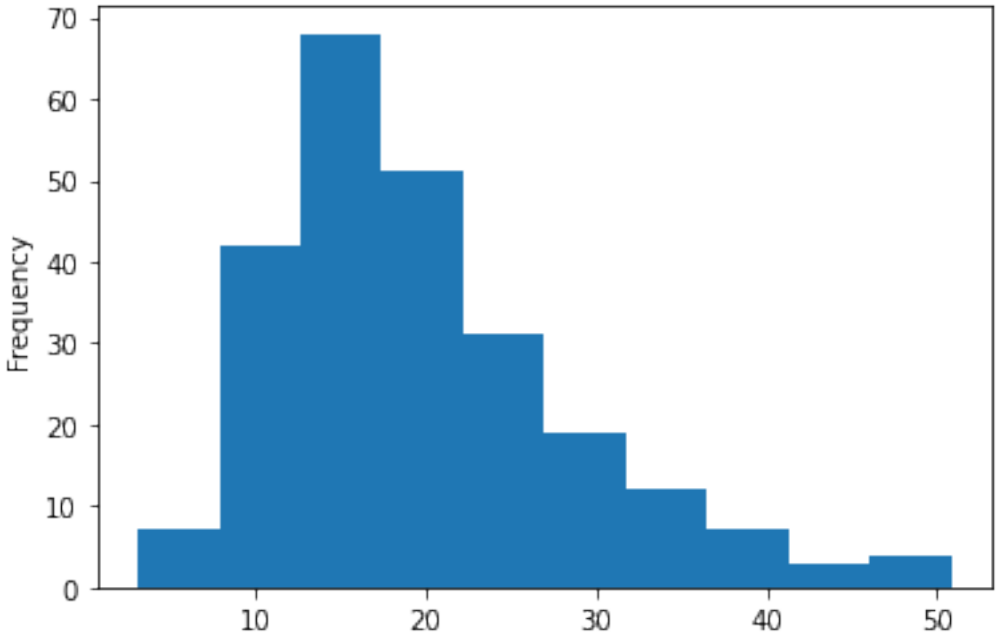
Con gráficos Los gráficos más frecuentes que podremos hacer con variables cuantitativas son:

- Histogramas
- Gráficos de densidad
- Gráficos de dispersión (2 variables)

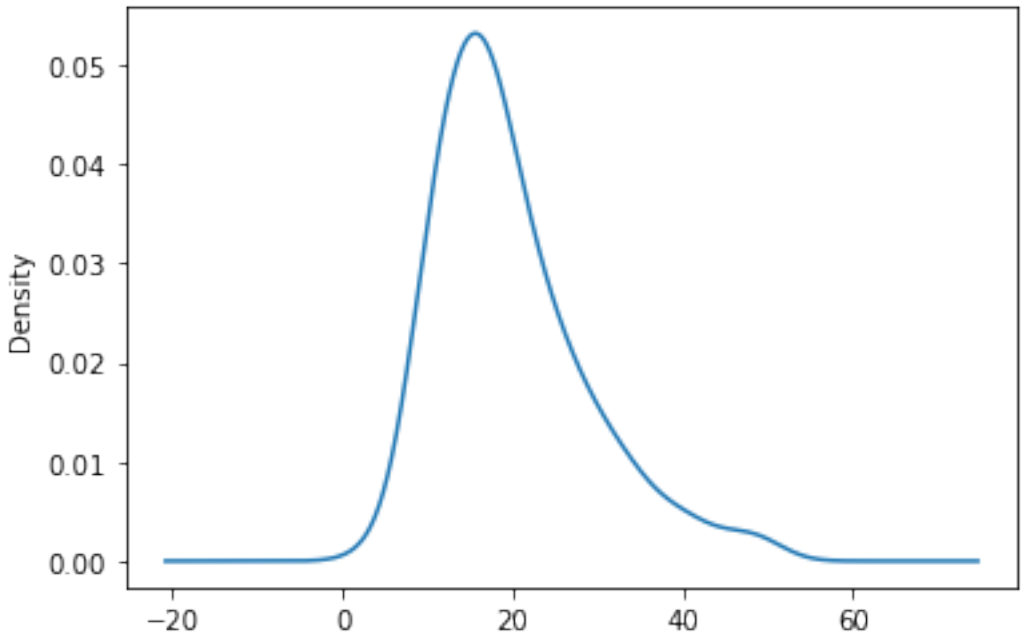
[48]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype   
---  ------      --------------  -----   
 0   total_bill  244 non-null    float64 
 1   tip         244 non-null    float64 
 2   sex         244 non-null    category
 3   smoker      244 non-null    category
 4   day         244 non-null    category
 5   time        244 non-null    category
 6   size        244 non-null    int64   
dtypes: category(4), float64(2), int64(1)
memory usage: 7.4 KB
```

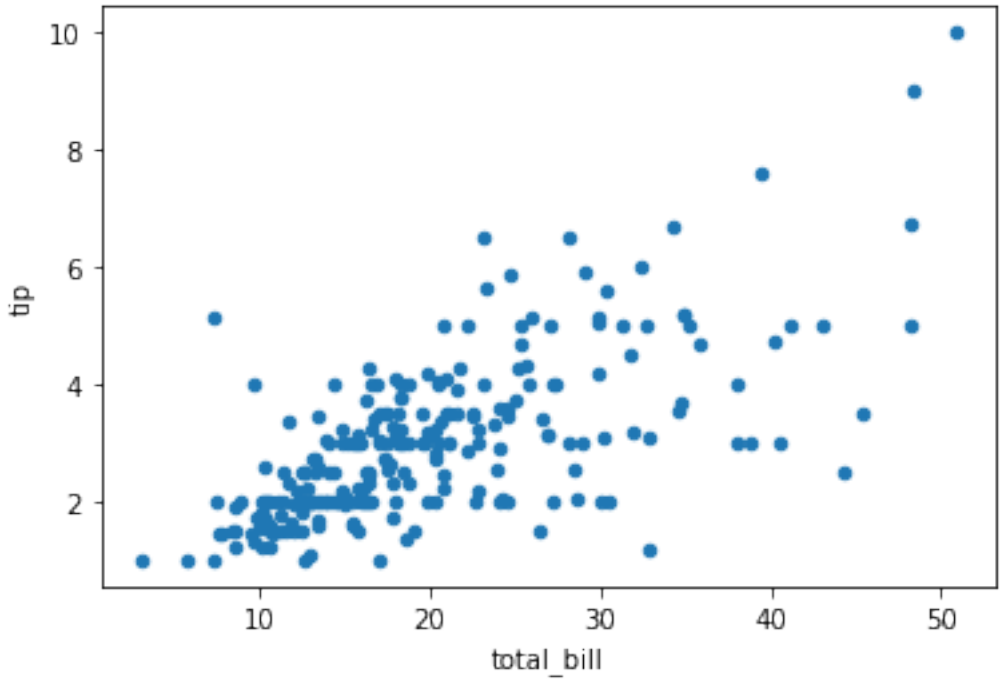
[49]:
```
#Ejemplo de histograma
df.total_bill.plot(kind = 'hist');
```



[31]:
```
#Ejemplo de densidad
df.total_bill.plot(kind = 'density');
```



[32]:
```
#Ejemplo de gráfico de dispersión
df.plot.scatter('total_bill','tip');
```



1.4.3 ¿Para qué usamos todo esto en Data Science?

Para todo :-)

Cuando estés haciendo análisis te pasarás gran parte del tiempo haciendo este tipo de análisis y gráficos para las fases de un proyecto de Data Science de:

- Calidad de datos
- Corrección de errores
- Análisis exploratorio
- Transformación y creación de variables

Además la correlación es una de las técnicas de preselección de variables en modelización predictiva, así que la usaremos potencialmente de dos formas:

- Identificar qué variables NO están correlacionadas con la variable a predecir, y por tanto no invertir tiempo en trabajar sobre ellas ni incluirlas en la modelización
- Identficiar qué variables están correlacionadas entre sí y por tanto no es conveniente usarlas simultáneamente en los modelos

Realmente esta parte descriptiva será mucho más usada en la práctica que la inferencial. Así que debes conocerla y entenderla bien.

02_Estadistica_II_Inferencial

September 16, 2021

Contenido

1 ESTADÍSTICA II PARA DATA SCIENCE: INFERENCIAL

1.1 PREPARACION

[1]:
```
#Carga de paquetes y datos
import pandas as pd
import numpy as np
import statistics
import scipy as sp
import seaborn as sns
import random
import math
from statsmodels.stats.proportion import proportions_ztest

df = sns.load_dataset('tips')
```

[2]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
```

```
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   total_bill  244 non-null    float64
 1   tip         244 non-null    float64
 2   sex         244 non-null    category
 3   smoker      244 non-null    category
 4   day         244 non-null    category
 5   time        244 non-null    category
 6   size        244 non-null    int64
dtypes: category(4), float64(2), int64(1)
memory usage: 7.4 KB
```

[3]:
```
df.head()
```

[3]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

1.2 ESTADÍSTICA INFERENCIAL

La estadística inferencial nos da herramientas para poder estimar **cómo de robustos y representativos del dato "real" en la población son las conclusiones que estamos obteniendo mediante nuestros análisis en la muestra**.

Ya sabemos que la población es el total de elementos sobre el cual estamos intentando obtener conclusiones. Y en la mayoría de los casos será un ideal ya que operativamente no podremos nunca acceder a él.

Por ejemplo si queremos estudiar los hábitos de compra de los españoles nunca podremos hacer una encuesta ni recopilar datos de TODOS los españoles.

O si estamos analizando los clientes de una empresa realmente nunca podremos analizar el total de clientes potenciales a los que esa empresa podría algún día llegar a tener como clientes.

Por ello en la práctica se entiende que siempre estaremos trabajando con una muestra de esa población.

(Tener en cuenta las notas que comentamos al inicio de este módulo sobre el grado en el que podemos considerar como realmente aleatoria una muestra que tenemos en un contexto empresarial)

Sobre esa muestra haremos nuestros análisis y obtendremos unas conclusiones, pero no sabemos hasta qué punto esas conclusiones o datos se acercan al dato "real" que obtendríamos si pudiéramos analizar toda la población.

En la práctica usaremos la **estadística inferencial** sobre todo para:

- Validar que el dato que hemos obtenido en la muestra es, bajo unas condiciones probabilísticas, un dato correcto, robusto y extrapolable

- Conocer los límites o intervalos superior e inferiores al dato que podríamos considerar válidos
- Aceptar o rechazar ciertas conclusiones (hipótesis) que queramos poner a prueba

En términos técnicos los puntos anteriores se cubren mediante lo que se llama:

1. Estimación de intervalos de confianza
2. Contraste de hipótesis

Además por el camino aprenderemos sobre distribuciones de probabilidad, y cómo hacer muestras (algo que aún en tiempos de Big Data sigue siendo de mucha utilidad).

1.2.1 Distribuciones

Una distribución es una función que muestra los **posibles valores de una variable y como de frecuente o probable es que ocurra cada uno**.

Es un concepto clave para todo lo que vamos ver después, especialmente la distribución que se llama normal o curva de Gauss.

La gran clave conceptual de la utilidad de las distribuciones es que pueden ser definidas por unos parámetros, y por tanto conocidas en todo su recorrido.

Por tanto, si cuando estamos analizando un evento, descubrimos que ese evento parece seguir una distribución conocida, es como tener un mapa, o como conocer las preguntas del examen.

Ya que podremos utilizar ese conocimiento de la distribución para hacer predicciones o contrastar hipótesis.

Por ejemplo se ha visto que lo que se llama "teoría de colas", como el número de llamadas que entran por unidad de tiempo en un call center, suele seguir la distribución de Poisson. Así que si queremos dimensionar adecuadamente un call center podemos usar esta distribución para predecir probabilísticamente el número de llamadas que entrarán y dimensionar en consecuencia.

Podemos hacer una gran clasificación de las distribuciones entre:

- Distribuciones de variables discretas: Bernouilli, Binomial, Poisson
- Distribuciones de variables continuas: Normal, T de Student, distribución F

Las discretas las vamos a repasar brevemente y poner ejemplos gráficos para que te suenen. Y en las contínuas realmente nos vamos a centrar en la distribución Normal que es la más imporante para todo lo que vamos a ver después.

Distribuciones discretas Bernouilli

Sigue esta distribución todo experimento aleatorio en que solo pueden ocurrir dos sucesos que además son mutuamente excluyentes.

El ejemplo más clásico es tirar una moneda.

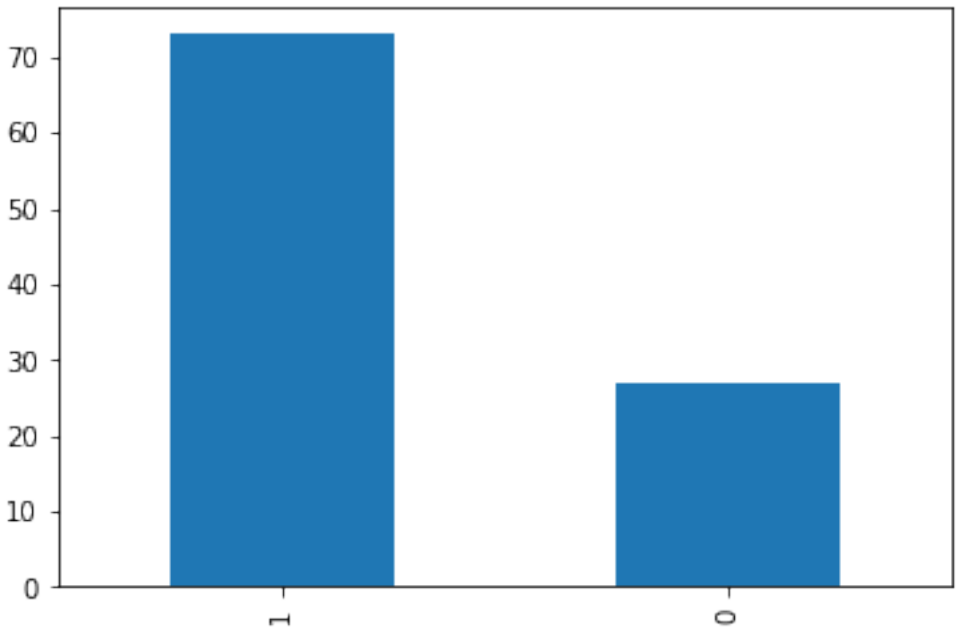
Los eventos no tienen por que ser 50%-50% pueden tener diferentes probabilidades siempre que suemen 100%.

[4]:
```
#Ejemplo de una moneda trucada para que el 70% de las veces salga cara
from scipy.stats import bernoulli
moneda = bernoulli.rvs(size=100,p=0.7)
```

```
moneda
```

[4]:
```
array([1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1,
       1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1,
       1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1,
       0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1,
       1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1])
```

[5]:
```
pd.Series(moneda).value_counts().plot(kind = 'bar');
```



Binomial

Sigue esta distribución la suma de los "éxitos" (los unos) que se obtienen en n repeticiones de un experimento de Bernouilli donde la probabilidad (p) de unos se mantiene constante.

Por ejemplo si tiro una moneda (no sesgada) 10 veces, y repito ese experimento 100 veces, ¿en cuantos de esos experimentos obtendré 0 caras, 1 cara, 2 caras, etc.

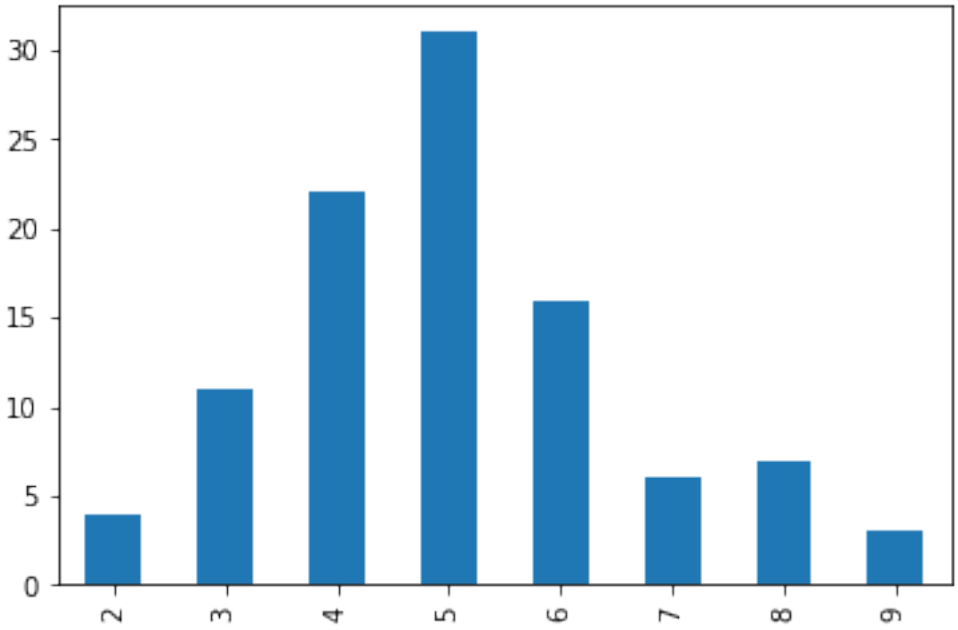
A medida que n aumenta, y siempre que p no tome valores muy extremos la binomial tiende hacia la normal. Así que en la práctica podremos usar la normal como un proxy en muchas ocasiones.

[6]:
```
from scipy.stats import binom
monedas = binom.rvs(n=10,p=0.5,size=100)
monedas
```

[6]:
```
array([5, 9, 3, 5, 3, 6, 4, 5, 5, 5, 7, 4, 2, 6, 5, 4, 4, 5, 6, 4, 5, 5,
       7, 6, 6, 5, 4, 4, 3, 6, 4, 8, 2, 5, 4, 5, 6, 5, 5, 4, 5, 5, 4, 6,
       6, 9, 5, 8, 4, 8, 5, 8, 8, 4, 5, 7, 3, 6, 6, 5, 5, 5, 5, 4, 7, 4,
```

```
       3, 5, 3, 5, 4, 5, 3, 9, 6, 4, 7, 5, 7, 4, 3, 5, 6, 6, 8, 3, 2, 4,
       4, 4, 5, 2, 5, 3, 8, 4, 6, 6, 5, 3])
```

[7]:
```
pd.Series(monedas).value_counts().sort_index().plot(kind = 'bar');
```



Poisson

Trata de predecir el número de veces que un evento ocurrirá en un intervalo de tiempo.

Por ejemplo lo que decíamos de teoría de colas en un call center. O el número de pacientes de urgencias que llegarán cada hora.

El parámetro que la define es Lambda: la media del número de eventos por intervalo de tiempo.

Conforme Lambda se aleja de valores muy bajos esta distribución tiende hacia la normal.

También es especialmente útil para "sucesos raros".

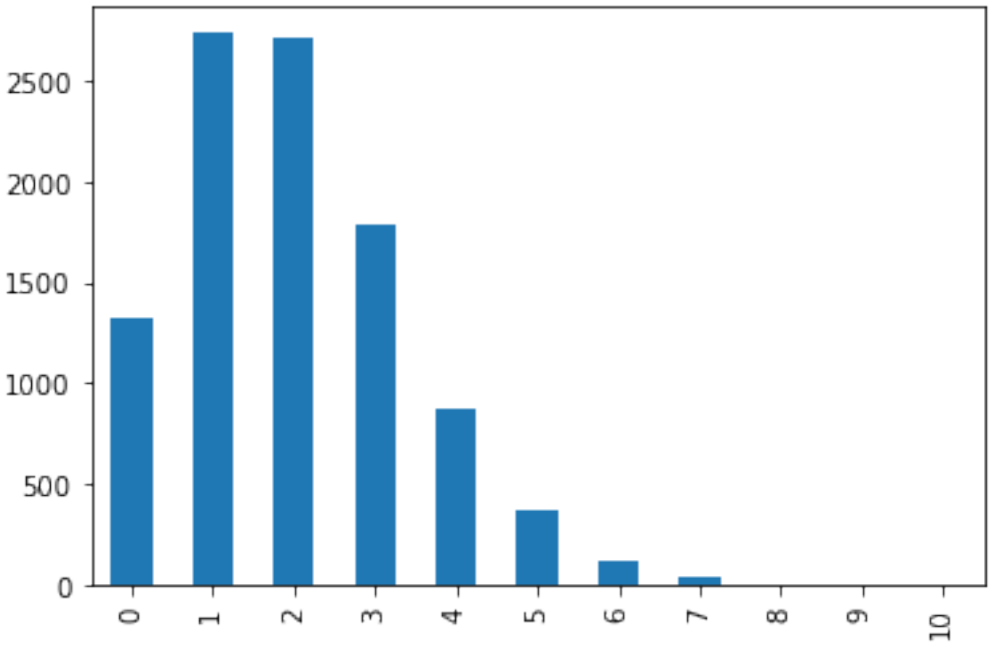
Por ejemplo para predecir el número de suicidios diarios en un país, cuantos trillizos nacerán en un día, número de visitantes concurrentes en una web pequeña, etc.

Ejemplo: si lo esperado es que lleguen 2 pacientes cada 15 minutos a urgencias, ¿cual sería la distribución más probable de llegadas?

[8]:
```
#El parámetro mu en esta implementación es Lambda
from scipy.stats import poisson
llegadas = poisson.rvs(mu=2,size=10000)
llegadas
```

[8]:
```
array([1, 0, 1, …, 2, 1, 0])
```

[9]:
```
pd.Series(llegadas).value_counts().sort_index().plot(kind = 'bar');
```



Distribuciones continuas Distribución normal

Es la distribución más importante y la más utilizada. También llamada curva o campana de Gauss.

Ya que es la que se produce cuando una variable está compuesta de otras muchas variables.

Y eso es lo más frecuente en casi cualquier ámbito.

Por ejemplo el peso de una persona depende de muchos factores: genética, alimentación, edad, etc. Y se demuestra que se distribuye según una normal.

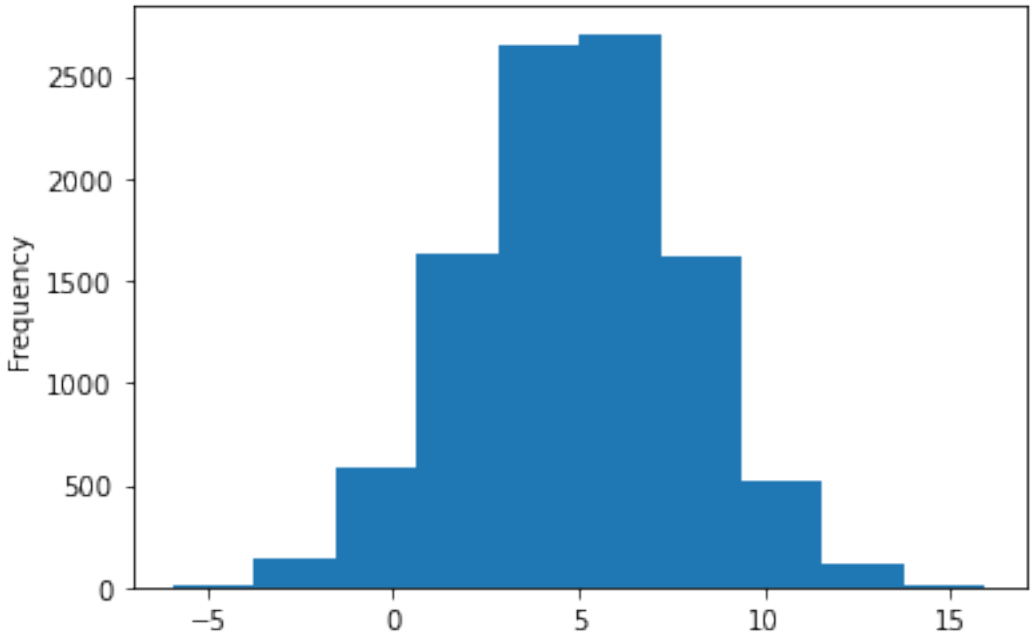
O la inteligencia que también depende de múltiples factores genéticos y culturales.

Etc.

Además la mayoría de distribuciones, cuando trabajamos con grandes números como es el caso de data science, convergen hacia la normal.

La normal se define por dos parámetros: la media y la desviación típica.

[10]:
```
#Ejemplo de una normal
#Loc es la media y scale es la desviación típica
from scipy.stats import norm
normal = norm.rvs(size=10000,loc = 5, scale = 3)
pd.Series(normal).plot(kind = 'hist');
```



Otras distribuciones

Existen otras distribuciones, como la F de Snedecor que se usa para el contraste de igualdad de varianzas, o la T de student que es una normal para cuando tenemos poca muestra o desconocemos la varianza poblacional.

Esta última tiene más relevancia, ya que la usaremos en ciertos casos del contraste de hipótesis.

Pero quédate simplemente con que es como una normal pero con mayor densidad en las colas.

1.2.2 Teorema del Límite Central

Es uno de los teoremas más importantes de la estadística, ya que es el que nos va a permitir aplicar con seguridad todo lo que veremos en los apartados siguientes.

Básicamente consiste en que se ha demostrado que si hacemos medias sobre muchas muestras aleatorias de una población, la distribución resultante de todas esas muestras se va a distribuir según una normal, **da igual cual fuera forma de la distribución original en la población**.

El conjunto de esas medias sobre sobre muchas muestras aleatorias de una población es a su vez una distribución, y se llama **distribución muestral**.

Pero además dice que:

1. La media de la distribución muestral tiende a converger a media de la población. Por tanto podemos usar el dato de la distribución muestral como válido en la población
2. La variabilidad de la distribución muestral, que se llama **error típico** (o también lo verás como error estandar), va a ser igual a la desviación típica de la población dividida por la raiz cuadrada del tamaño de la muestra

Osea que si sabemos la media de la distribución muestral estamos muy cerca de la media real de la población.

Pero **en cada una de las muestras** de la distribución muestral (que será lo que manejemos en la realidad, una sola de esas muestras) esa media va tener un valor parecido pero diferente a las demás muestras. Es decir, es un dato que tiene una variabilidad.

Si esa variabilidad (el error típico) es grande significa que esa media que estamos sacando en nuestra muestra puede no ser buena estimación de la media real en la población.

Pero si es pequeña significa que esa estimación sí está más cerca del valor real en la población.

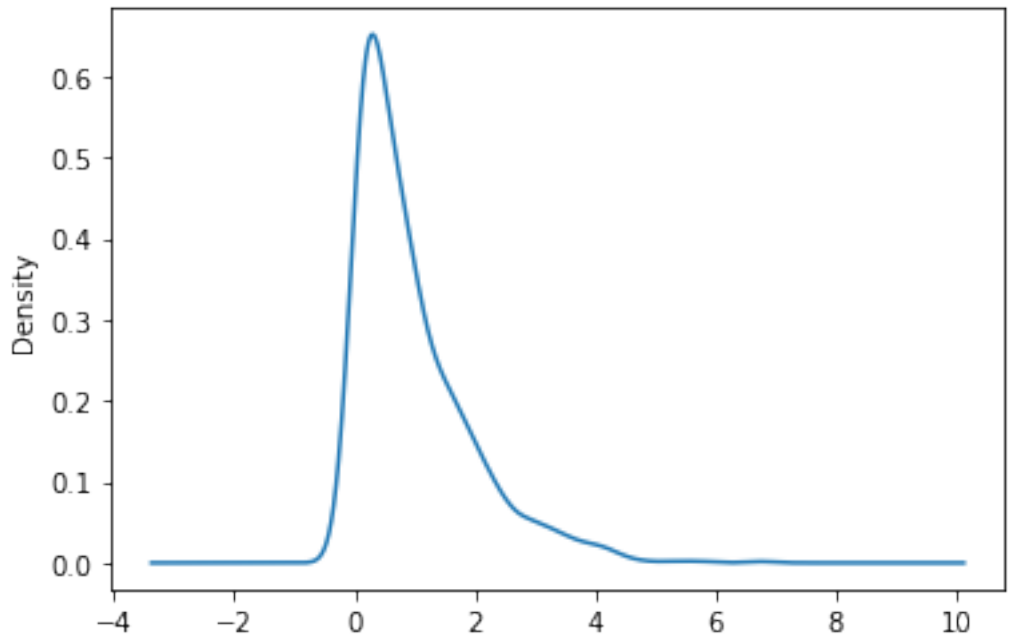
Y como el denominador del error típico incluye el tamaño muestral, pasará que cuanto mayor sea la muestra menor será la variabilidad, es decir, más precisa será la media muestral con respecto a la de la población.

En la práctica, como no tenemos la desviación típica de la población, se sustituye por la desviación típica de la muestra, de forma que usando la fórumula del punto 2) podremos estimar el error típico de la distribución muestral a partir de la desviación típica de la muestra.

Pero aquí hay mucha miga, así que vamos paso a paso.

Vamos a crear una distribución simulando una población con otra forma, por ejemplo una exponencial

[11]:
```
from scipy.stats import expon
poblacion = expon.rvs(loc=0, scale=1,size=1000)
pd.Series(poblacion).plot(kind = 'density');
```



Ahora sobre la población vamos a extraer 500 muestras aleatorias de tamaño 500 cada una de ellas. Y por tanto tendremos una **distribución muestral**.

[12]:
```
random.seed(1234)
muestras = [random.sample(list(poblacion), 500) for i in range(500)]
medias = [np.array(muestra).mean() for muestra in muestras]
#Visualizamos las 10 primeras
medias[0:10]
```

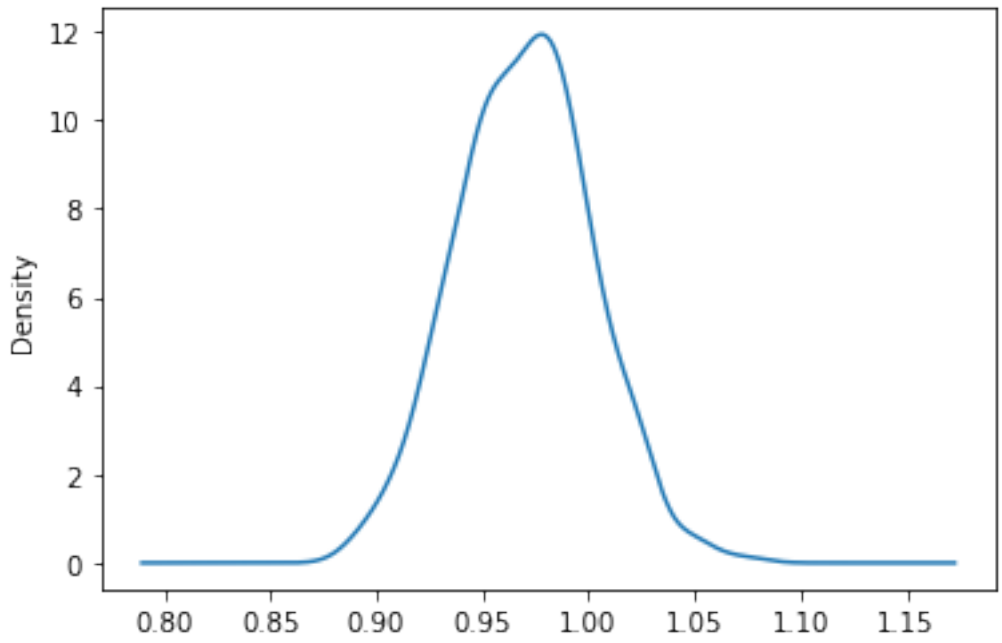
[12]:
```
[0.9997672640116994,
 0.9599506126016479,
 1.0238573887582647,
 0.9932353591815983,
 0.9618175641624933,
 0.909207361056239,
 0.963307051218361,
 0.9777230627175867,
 0.9959170336565685,
 0.9478127244442255]
```

Calculamos la media en la población y en la distribución muestral para ver que efectivamente tiende a converger.

Y también graficamos la distribución muestral para ver que es una normal.

[13]:
```
media_poblacion = poblacion.mean()
media_muestras = np.array(medias).mean()

print('La media de la población es: %f'%media_poblacion)
print('Y la media de la distribución muestral es: %f'%media_muestras)

pd.Series(medias).plot(kind = 'density');
```

```
La media de la población es: 0.970981
Y la media de la distribución muestral es: 0.969980
```



Ahora bien, en la práctica no tendremos 500 muestras, si no que tendremos un dataset que podemos entender como una sola de esas muestras, y sobre ese dataset haremos nuestros análisis y calcularemos nuestros datos.

Por tanto no tendremos esos datos de la distribución muestral de las 500, pero podemos volver a usar el teorema del límite central, que nos dice que el error típico de la distribución muestral será igual a la desviación típica encontrada en la muestra dividida por la raiz cuadrada del tamaño de la muestra.

Por ejemplo vamos a coger una de las muestras y calcular el error típico a partir de ella.

[14]:
```
muestra = muestras[0]

desv_tip_muestra = np.array(muestra).std()
error_tipico = desv_tip_muestra / math.sqrt(500)

print('El error típico según el teorema del límite central es: %f'%error_tipico)
```

```
El error típico según el teorema del límite central es: 0.044921
```

[15]:
```
error_tipico
```

[15]: 0.044921344657011766

[16]:
```
np.array(muestra).mean()
```

[16]: 0.9997672640116994

EN CONCLUSIÓN, el teorema del límite central nos va a permitir que cuando en la realidad sólo estemos ante un conjunto de datos que entenderemos como sólo una muestra de todo el espacio muestral disponible podamos:

- Asumir que la media de nuestra muestra converge a la de la distribución muestral y por tanto a la de la población
- Saber que nuestra media realmente es solo una de todas las posibles, y por tanto es un dato que tiene una variación
- Poder calcular esa variación, mediante el error típico

En nuestro ejemplo de arriba, sabemos que el valor de la media en la muestra va a estar en torno al 0.98.

Pero sabemos que ese dato tiene una variación, en concreto en nuestra muestra encontramos que el error típico es 0.04.

Es decir, el dato de la media de 0.98 variará en 0.04 **por cada desviación típica que nos alejemos de la media**

Pero entonces, ¿dónde ponemos los límites para decir si un nivel de variación es ya razonable o no?

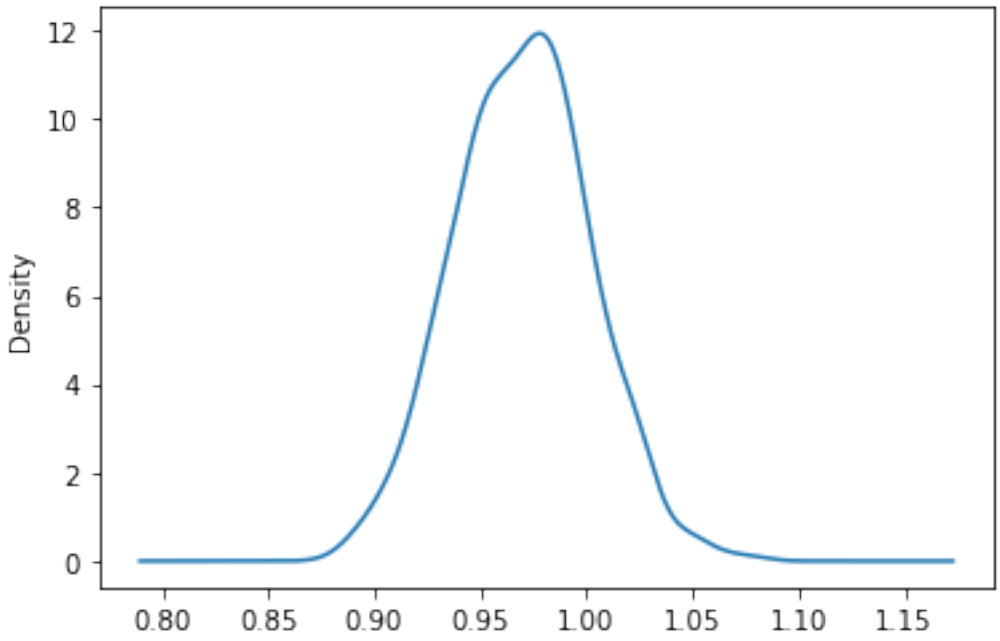
Por ejemplo, vamos a recuperar el gráfico de la distribución muestral.

Y sabemos que el dato real estará en torno a 0.98, aunque no estamos seguros de que sea exactamente 0.98.

Entonces ¿qué consideraríamos exactamente como un dato representativo de nuestra media? ¿La media también podriá ser 0.93, o 1.02, o 2.8?

Justamente resolver esa pregunta es lo que se llama calcular el "Intervalo de Confianza"

[17]:
```
pd.Series(medias).plot(kind = 'density');
```



1.2.3 Cálculo de intervalos de confianza

Nos habíamos quedado en que tenemos un dato (la media en nuestra muestra) que se aproxima al dato real en la población, pero que puede no ser exactament el dato real.

Es decir, usando ese dato de la muestra como aproximación estamos cometiendo un error: el **margen de error**.

Margen de error Parte de ese error viene explicado por la variabilidad, a menos variabilidad menos error, y esa variabilidad la calculábamos con el error típico.

Pero habíamos visto que el error típico es la desviación típica de la distribución muestral.

Es decir, que si nos alejamos una desviación típica de la media el error que podemos estar cometiendo era en nuestros datos de 0.04. (Esto habría que hacerlo tanto por la derecha como por la izquierda).

Pero entonces si nos alejamos 2 desviaciones típicas estaríamos cometiendo un posible error de 0.08.

Es decir, que otra parte del error viene dado por cuanto queramos nosotros alajarnos de la media.

En definitiva el **margen de error será igual al error típico multiplicado por el número de desviaciones típicas que queramos alejarnos**.

U operativamente, a partir de lo que tenemos disponible en una muestra:

margen de error = error típico * número desviaciones típicas

Si lo descomponemos tenemos que:

El margen de error depende de:

- la variabilidad de lo que estemos midiendo: esto no lo podemos controlar
- el tamaño muestral: a mayor tamaño menor error
- el número de desviaciones típicas que queramos alejarnos, que lo definimos nosotros en función del nivel de confianza que queramos aplicar

Niveles de confianza Pero si el número de desviaciones típicas que nos queremos alejar lo decidimos nosotros … ¿En qué nos podemos basar para decidirlo?.

Aquí entra el concepto de **niveles de confianza**.

Imagínate que pienso un número del 1 al 10 y tienes que adivinarlo.

Para ello te dejo que digas un intervalo de números.

Por ejemplo, si te dejara un intervalo de 5 números, ¿cómo de confiado estarías en acertar?. Pues un 50% no?

Y si te dejara un intervalo de 9 números, ¿cómo de confiado estarías en acertar?. Pues un 90%

Ahora bien, en el caso de que mi número estuviera en tu intervalo. ¿Qué error medio estarías cometiendo si hiciéramos el experimento infinitas veces?

- En el caso del intervalo de cinco números estarías cometiendo un error medio de 2.5

- En el caso del intervalo de nueve números estarías cometiendo un error medio de 4.5

Por tanto **cuanto mayor sea el intervalo más seguro vas a estar de que mi número va estar en ese intervalo, pero mayor va a ser también el error medio que estás cometiendo** y por tanto de menor valor será tu dato.

Pues eso es exactamente lo que pasa con los niveles de confianza.

Y dado que es sobre una distribución normal y por tanto conocida, podemos cuantificarlo en términos de probabilidad.

Por ejemplo sabemos que una normal el 68% de los datos están entre la media y una desviación típica por abajo y otra por arriba.

Por tanto podemos darle la vuelta y decir: si quiero trabajar con una confianza del 68% entonces tengo que marcar un intervalo entre la media +- una desviación típica.

Y así se suele hacer. Primero marcas el nivel de confianza al que quieres trabajar y después calculas cuantas desviaciones típicas supone.

Se suelen usar los siguientes estándares:

- 95.5% de nivel de confianza que equivale a 2 desviaciones típicas
- 99.7% de nivel de confianza que equivale a 3 desviaciones típicas

Calculemos los datos en nuestro ejemplo y vamos a ver que, al igual que el ejemplo de adivinar el número, si incrementamos la confianza también incrementamos el error.

[18]:
```
error_tipico
```

[18]:
```
0.044921344657011766
```

[19]:
```
#Error muestral con nuestros datos para un nivel de confianza del 95,5%
error_95 = 2 * error_tipico
print(error_95)

#Error muestral con nuestros datos para un nivel de confianza del 99,7%
error_99 = 3 * error_tipico
print(error_99)
```

```
0.08984268931402353
0.1347640339710353
```

Distribución normal tipificada En el punto anterior decíamos que vamos a usar el número de desviaciones típicas que un dato se aleja de la media para cuantificar el nivel de confianza al que estamos trabajando.

Pero sabemos que varias distribuciones, aunque todas sean normales, van a tener distintas medias y distintas desviaciones típicas.

Por ello se suele aplicar una operación que se llama tipificar, y que usaremos también en machine learning, para poder trabajar con un estandar. De hecho a las puntuaciones típicas también se les llama puntuaciones estandar.

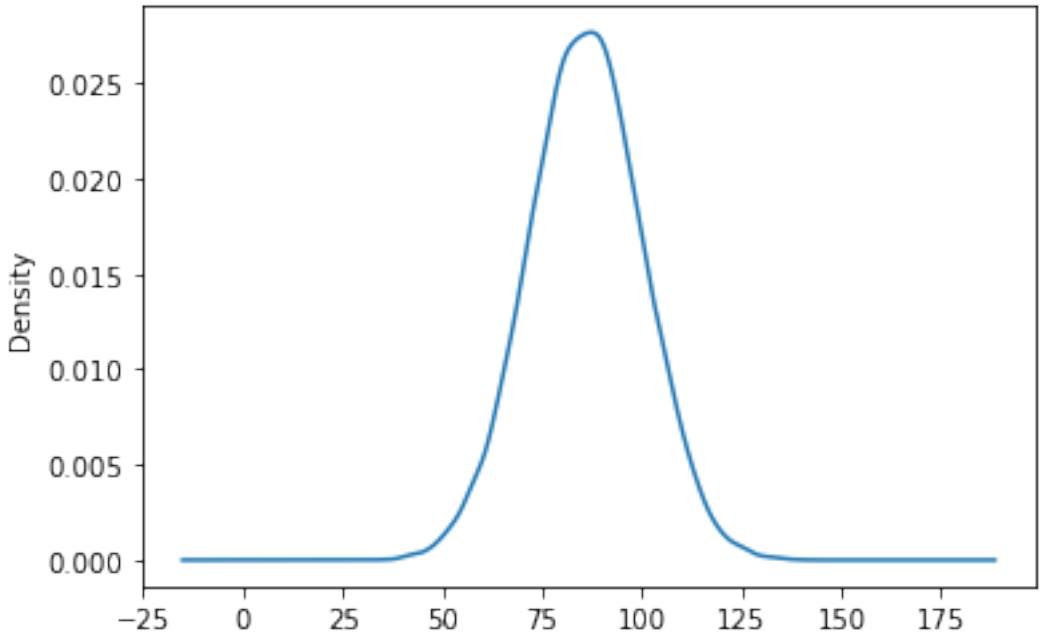
La fórmula consiste en tomar cada dato de la distribución, restarle la media, y dividir el resultado por la desviación típica.

El resultado es una métrica que se llama puntuaciones z, o típicas o estandar, y que va a tener una distribución con media cero y desviación típica 1.

Vamos a ver un ejemplo donde primero vamos a generar una distribución normal llamada sin_tipificar, con media 86 y desviacion típica 14.

Y después vamos a tipificarla.

[20]:
```
from scipy.stats import norm
sin_tipificar = norm.rvs(size=10000,loc = 86, scale = 14)
pd.Series(sin_tipificar).plot(kind = 'density');
```



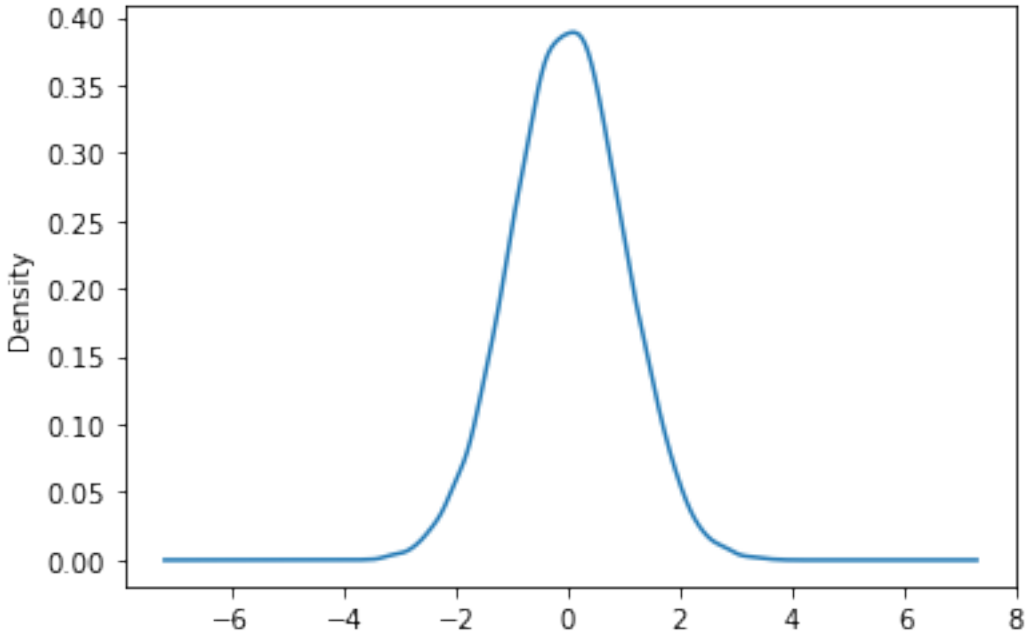
[21]:
```
#Vamos a calcular sus estadísticos
media_sin = sin_tipificar.mean()
desv_tip_sin = np.array(sin_tipificar).std()
print('Media: %.2f'%media_sin)
print('Desv Tip: %.2f'%desv_tip_sin)
```

```
Media: 85.90
Desv Tip: 14.06
```

[22]:
```
#Ahora vamos a tipificarla y ver que aunque cambien los estadísticos la␣
 ↪distribución es igual
tipificada = np.array([(dato - media_sin) / desv_tip_sin for dato in␣
 ↪sin_tipificar])
```

```
pd.Series(tipificada).plot(kind = 'density');

media_tip = tipificada.mean()
desv_tip_tip = np.array(tipificada).std()
print('Media: %.2f'%media_tip)
print('Desv Tip: %.2f'%desv_tip_tip)
```

```
Media: 0.00
Desv Tip: 1.00
```



Intervalos de confianza Ahora que ya sabemos estimar el margen de error, calcular el intervalo de confianza es muy sencillo.

El límite inferior vendrá dado por la media menos el margen de error.

Y el límite superior vendrá dado por la media más el margen de error.

Por tanto el intervalo entre el que estará el dato real, al nivel de confianza elegido vendrá dado por:

[media - margen de error, media + margen de error]

[23]:
```
#Intervalo de confianza en nuestro ejemplo a un NC del 95,5%
media_muestra = np.mean(muestra)

ic_95 = [media_muestra - error_95, media_muestra + error_95]
print(ic_95)
```

```
#Intervalo de confianza en nuestro ejemplo a un NC del 99,7%
ic_99 = [media_muestra - error_99, media_muestra + error_99]
print(ic_99)
```

```
[0.9099245746976758, 1.0896099533257229]
[0.865003230040664, 1.1345312979827347]
```

A la hora de intrepretarlo podríamos leerlo así:

- Estamos un 95% seguros de que la media real va a estar entre 0.88 y 1.05
- Estamos un 99% seguros de que la media real va a estar entre 0.84 y 1.09

Aunque técnicamente sería más correcto leerlo así:

- Si repitiéramos 100 veces el experimento, en 95 de ellas nos saldría un dato entre 0.88 y 1.05
- Si repitiéramos 100 veces el experimento, en 99 de ellas nos saldría un dato entre 0.84 y 1.09

Intervalos de confianza en proporciones Hemos visto como calcular los intervalos de confianza cuando estamos estimando medias.

Pero el otro gran estadístico que podemos estar usando son las proporciones.

Por ejemplo si una landing ha convertido en un piloto al 13% podemos querer estimar sus intervalos.

Conceptualmente todo es igual, pero cambia una cosa: la fórmula para calcular el error típico.

El error típico para distribuciones muestrales de proporciones se calcular como:

Raiz cuadrada de ((p * q) dividido por n), siendo:

p = la proporción obtenida

q = 1-p

n = tamaño de la muestra

Por ejemplo en el caso anterior habíamos obtenido la proporción de conversión del 0.13 en una muestra de 500 visitantes a la web.

Vamos a calcular su intervalo de confianza con un 95,5% de nivel de confianza.

[24]:
```
error_tipico_prop = math.sqrt((0.13 * 0.87)/500)
print(error_tipico_prop)
```

```
0.015039946808416579
```

[27]:
```
error_tipico_prop_95 = error_tipico_prop * 2
ic_prop_95 = [0.13 - error_tipico_prop_95, 0.13 + error_tipico_prop_95]
ic_prop_95
```

[27]:
```
[0.09992010638316684, 0.16007989361683317]
```

Es decir, podríamos esperar que si nada cambia esa landing estuviera convirtiendo entre el 9.9% y el 16% cada día (siempre que hubiera muestra suficiente)

¿Para qué nos sirve esto en data science?

En data science raras veces estaremos haciendo estimación de intervalos como tal, pero sí es algo muy frecuente en ciertas técnicas como por ejemplo el forecast, donde además de obtener la salida de la predicción del modelo podemos pedir al software que saque los intervalos de confianza de esa predicción.

Es decir, en vez de estimar que el mes que viene venderemos 180.000€ es más seguro estimar que venderemos entre 172.000€ y 188.000€.

Además cuando los intervalos son muy amplios sabemos que hay mucho margen en la predicción y por tanto hay que tratarla con precaución.

También es muy usado en casos como la planificación de la demanda, donde siempre deberemos trabajar con un "colchón de seguridad" para no quedarnos sin stock, y los intervalos nos ayudan a calcular ese colchón.

Y también es usado para calcular el tamaño correcto que debería tener una muestra para poder trabajar a unos niveles de confianza y de error determinados.

Por la importancia del muestreo vamos a verlo en su propio apartado.

1.2.4 Muestreo

Aún en tiempos de Big Data el muestreo sigue siendo muy útil en data science.

Aunque la capacidad de computación sea cada vez mayor, la cantidad de datos y de análisis y modelos que hay que hacer en contextos empresariales tampoco para de crecer.

El muestreo ha demostrado su efectividad para el tipo de análisis que hacemos en data science. Siendo que los modelos construídos sobre una muestra bien realizada son igual de válidos y potentes que los realizados sobre toda la población.

Por lo que, aunque tuviéramos la fuerza bruta necesaria para procesar miles de veces toda la información no tiene ningún sentido hacerlo. Y casi siempre se trabaja con muestras para el desarrollo, que después ya se implantará sobre toda la población.

Es cierto que precisamente debido a la abundancia de datos muchas veces el muestreo se hace "a ojo" sabiendo que vas a cumplir los grandes números suficientes para que, siempre que sea aleatorio, sea correcto.

Pero aquí te voy a enseñarlo a hacerlo correctamente, que siempre es lo ideal.

De todas formas, insisto, lo más importante siempre es que para que funcione la muestra tiene que ser aleatoria.

Fórmula para calcular el tamaño de la muestra Cuando vimos el error muestral ya teníamos el tamaño de la muestra incluído en esa ecuación. Recordamos:

margen de error = (desviación típica de la muestra / raiz del tamaño muestral) * número desviaciones típicas

Por tanto sólo necesitamos despejar el tamaño muestral en esa ecuación. Con un añadido, y es que lógicamente antes de hacer la muestra no conocemos su desviación típica, así que se cambia esa variable por la situación de variaza máxima, que en estadística se obtiene como p * q, es decir 0.5 * 0.5.

Hay que aclarar que de nuevo existen muchas fórmulas dependiendo de la situación. Pero en data science casi siempre trabajaremos con poblaciones infinitas (>100.000).

Por tanto la fórmula final para calcular el tamaño muestral es:

[ ]:
```
#muestra = ((z**2) * (50*50)) / (me ** 2)
```

Es decir, que el tamaño de la muestra va a depender de:

- El nivel de confianza al que queramos trabajar: que determinará el z
- El error muestral máximo que queramos cometer

Si queremos trabajar a más NC o cometer menor error deberemos incrementar la muestra.

Por ejemplo:

Calcula el tamaño de muestra necesario si queremos hacer un análisis donde como mucho nos podamos desviar un 3% y con la certeza de que de 100 veces que lo hiciéramos en 95 nos saldrían conclusiones similares

[ ]:
```
#Vamos a crear una función que calcule el tamaño muestral en base a un NC␣
 ↪(pasado como zetas)
# y un margen error (pasado en porcentaje)
def tamaño(z,me):
    tamaño = ((z**2) * (50*50)) / (me ** 2)
    return(tamaño)

#Calculamos
tamaño(1.96,3)
```

1.2.5 Alpha y pvalor

Cuando ya estemos en la práctica estaremos frecuentemente usando dos conceptos: alpha y pvalor.

Por ejemplo siempre que hagamos modelos que vienen de la estadística como las regresiones estaremos usando estos conceptos para ver si los coeficientes son significativos, etc.

Pues con lo que ya has aprendido, no sólo vas a saber usarlos, si no que vas a entender el por qué :-)

Alpha no es más que 1-NC. Es decir, si trabajamos a un NC del 95% entonces Alpha es 5%. Aunque normalmente nos vendrá en tanto por uno, es decir 0.05.

Por otro lado está el pvalor, también llamado la significación estadística, porque ayuda a diferenciar si un resultado obtenido es estadísticamente significativo o es simplemente fruto del azar.

El **Pvalor es la probabilidad de obtener un valor tan extremo o más que el obtenido en la muestra asumiendo que la hipótesis nula es cierta**. (Más sobre la hipótesis nula en los siguientes apartados)

Entonces si esa probabilidad que nos da el pvalor es tan baja que nos parece "rarísimo" que haya pasado lo que hacemos es pensar que quizá la hipótesis nula no esa cierta.

Es decir, pongamos que aceptamos que el valor real de la media en la población es 100 (hipótesis nula).

Pero en nuestra muestra la media es 108 con un pvalor de 0.02. Ello significa que, si efectivamente la media real fuera 100, la probabilidad de que a nosotros nos saliera 108 o más es sólo del 2%.

Esto es super potente porque con él podemos hacer el proceso al revés. Es decir, podemos saber el límite de nivel de confianza que podríamos elegir para considerar el valor como significativo.

En nuestro ejemplo si hemos obtenido un pvalor de 0.02 con el dato de 108 en la muestra, podremos aceptar que el valor en la población es 100 solo a un 98% de NC pero no a un 99% de NC.

Por tanto, si nosotros definimos el mínimo NC que consideramos aceptable sólo tenemos que contrastar el pvalor contra el Alpha y ver directamente si es significativo a ese nivel o no.

En nuestro ejemplo, si trabajamos a NC = 99% y por tanto Alpha = 0.01, no se cumple que Pvalor < Alpha y por tanto no podemos aceptar la hipótesis de que la media sea 100 al 99% de NC.

Sin embargo sí lo aceptaríamos al un 95% NC.

Es decir, operativamente **se interpreta como que si el pvalor es menor al Alpha entonces tenemos que rechazar la hipótesis nula**.

Ya estaremos entrando en el contraste de hipótesis. Y vamos a explicar más lo que es la hipótesis nula y la alternativa.

1.2.6 Contraste de hipótesis

Hipótesis nula e hipótesis alternativa Prácticamente cualquier inferencia que queramos hacer podemos plantearla como el contraste de una hipótesis.

En el ejemplo que estábamos haciendo hasta ahora la hipótesis sería si el valor de la media en la población podría ser de 100 o no.

Vemos que realmente no es una hipótesis, si no dos. Lo que se llaman:

- Hipótesis nula
- Hipótesis alternativa

La **hipótesis nula** en la mayoría de pruebas estadísticas suele ser que no hay "efecto".

Por ejemplo si estamos comparando la efectividad de dos medicamentos, que no haya efecto aquí significa que ambos medicamentos van a tener resultados similares.

O si estamos contratando un coeficiente de una variable de un modelo, que no haya efecto significa que ese coeficiente podría ser cero, y por tanto esa variable no sería predictiva.

Normalmente la hipótesis nula es lo que nos gustaría "rechazar".

Por el contrario, la **hipótesis alternativa** es que sí existe efecto, y normalmente es lo que buscamos.

Por ejemplo en el caso de los medicamentos nos gustaría que sí hubiera diferencias. Por tanto:

- La hipótesis nula será que el resultado provocado por los medicamentos es similar
- La hipótesis alternativa será que el resultado provocado por los medicamentos es significativamente diferente

Pruebas de contraste de hipótesis Generalizando todo lo que hemos visto hasta ahora tenemos que:

1. Tenemos una distribución conocida, la normal, bien porque la variable original ya era normal o bien por lo que nos dice el teorema del límite central
2. Tenemos una hipótesis nula que queremos contrastar
3. Elegimos un Nivel de Confianza al cual queremos trabajar, que nos da un Alpha
4. Según la prueba que estemos haciendo usaremos un estadístico u otro, que nos devolverá un Pvalor
5. Si el Pvalor es menor que el Alpha entonces rechazamos la hipótesis nula, y por tanto aceptamos la alternativa

En cuanto al estadístico del punto 4 es la parte más compleja, por lo que a nivel práctico y por simplificar, vamos a recoger los 4 principales escenarios que se nos van a presentar y el estadístico que debemos usar en cada caso:

1. Contraste de medias en la población: es decir si la media obtenida puede ser compatible con una media hipotética de la población. Usaremos el estadístico t
2. Contraste de medias entre dos muestras: es decir si la diferencia entre las media de grupos diferentes es significativa o no. Usaremos el estadístico t
3. Contraste de proporciones en la población: es decir si la proporción obtenida puede ser compatible con una porporción hipotética en la población. Usaremos el estadístico z
4. Contraste de proporciones entre dos muestras: es decir si la diferencia entre la proporción de grupos diferentes es significativa o no. Usaremos el estadístico z

Contraste de medias en la población

Queremos ver si el valor de la media obtenido en la muestra puede ser compatible con un hipotético valor en la población.

Para ello seguiremos la metodología explicada, usando como estadístico de contraste la prueba t.

Usaremos la implementación de Scipy con la función ttest_1samp() ya que estamos usando el estadístico t en una sola muestra (no comparamos entre dos muestras)

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_1samp.html

A esta función tenemos que pasarle:

- a: los datos
- popmean: el valor de h0 a contrastar

Como ejemplo vamos a calcular si es posible que la media de total_bill en la población sea de 25$.

[ ]:
```
#Primero recordamos cual era el valor en nuestra muestra
df.total_bill.mean()
```

[ ]:
```
#Paso 1: definimos las hipótesis
h0 = 25
#Por tanto h1 es que sea diferente de 25
```

[ ]:
```
#Paso 2: elegimos un nivel de confianza que nos da un alpha.
#Elegimos NC = 95%
```

```
alpha = 0.05
```

[ ]:
```
#Paso 3: calculamos el pvalor según el estadístico de contraste elegido (t␣
↪sobre una muestra)
p_valor = sp.stats.ttest_1samp(a = df.total_bill, popmean = h0)[1]
print(f"{p_valor:.3f}")#Esto es para que no salga con notación científica
```

Por tanto vemos que el pvalor es menor que alpha, por tanto no podemos aceptar la H0 y por tanto no es probable que la media de propinas en la población sea de 25$.

Vamos a repetir el ejemplo, pero ahora con un valor quizá más probable. Testemos si la media de propinas podría ser de 20$.

[ ]:
```
#Paso 1: definimos las hipótesis
h0 = 20
#Por tanto h1 es que sea diferente de 20
```

[ ]:
```
#Paso 2: elegimos un nivel de confianza que nos da un alpha.
#Elegimos NC = 95%
alpha = 0.05
```

[ ]:
```
#Paso 3: calculamos el pvalor según el estadístico de contraste elegido (t␣
↪sobre una muestra)
# como la h0 es una igualdad dejamos el alternative por defecto
p_valor = sp.stats.ttest_1samp(a = df.total_bill, popmean = h0)[1]
print(f"{p_valor:.3f}")#Esto es para que no salga con notación científica
```

Ahora pvalor es mayor que alpha, luego no podemos rechazar H0, luego 20$ sí es un valor posible en la población a tenor de los datos de nuestra muestra.

Contraste de medias entre dos muestras

En este caso lo que queremos ver es si la diferencia obtenida entre 2 medias de dos grupos distintos es o no significativa.

Para ello seguiremos la metodología explicada, usando como estadístico de contraste la prueba t.

Usaremos la implementación de Scipy con la función ttest_ind() ya que estamos usando el estadístico t en dos muestras que son independientes (porque pertenecen a dos poblaciones distintas).

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html

A esta función tenemos que pasarle:

- a y b: los datos en forma de arrays numpy
- equal_var: ponerlo a False porque si no asume que ambas poblaciones tienen la misma varianza

Como ejemplo vamos a calcular si hay diferencias significativas entre las propinas que dejan las mujeres y los hombres.

[ ]:
```
#Primero recordamos cual era el valor en nuestra muestra
df.groupby('sex').total_bill.mean()
```

```
[ ]: df.head()
```

```
[ ]: #Tenemos que crear dos vectores, uno con los hombres y otro con la mujeres
     hombres = df.loc[df.sex == 'Male','total_bill']
     mujeres = df.loc[df.sex == 'Female','total_bill']
```

```
[ ]: #Paso 1: definimos las hipótesis
     #H0: los hombres y las mujeres dejan la misma propina
     #H1: los hombres y las mujeres no dejan la misma propina
```

```
[ ]: #Paso 2: elegimos un nivel de confianza que nos da un alpha.
     #Elegimos NC = 95%
     alpha = 0.05
```

```
[ ]: #Paso 3: calculamos el pvalor según el estadístico de contraste elegido (t␣
      ↪sobre dos muestras independientes)
     p_valor = sp.stats.ttest_ind(a = hombres, b = mujeres, equal_var = False)[1]
     print(f"{p_valor:.3f}")#Esto es para que no salga con notación científica
```

Por tanto vemos que el pvalor es menor que alpha, por tanto no podemos aceptar la H0 y por tanto no es probable que la media de propinas en los hombres sea la misma que en las mujeres.

Rechazamos h0 y concluímos que las diferencias sí son significativas al 95% de confianza.

Pero vamos a ver si también lo son al 99% de confianza.

```
[ ]: #Paso 1: definimos las hipótesis
     #H0: los hombres y las mujeres dejan la misma propina
     #H1: los hombres y las mujeres no dejan la misma propina
```

```
[ ]: #Paso 2: elegimos un nivel de confianza que nos da un alpha.
     #Elegimos NC = 99%
     alpha = 0.01
```

```
[ ]: #Paso 3: calculamos el pvalor según el estadístico de contraste elegido (t␣
      ↪sobre dos muestras independientes)
     p_valor = sp.stats.ttest_ind(a = hombres, b = mujeres, equal_var = False)[1]
     print(f"{p_valor:.3f}")#Esto es para que no salga con notación científica
```

Ahora pvalor es mayor que alpha, luego no podemos rechazar H0, luego no podríamos afirmar con un 99% de confianza que los hombre dejen diferente propina que las mujeres.

Contraste de proporciones en la población

Queremos ver si una proporción obtenida en la muestra puede ser compatible con un hipotético valor en la población.

Para ello seguiremos la metodología explicada, usando como estadístico de contraste la prueba z.

Scipy no tiene implementación (hasta donde yo sé) para hacer contrastes de proporciones, así que vamos a usar la implementación del paquete statsmodels, con la función proportions_ztest() ya

que estamos usando el estadístico z, y como es en una sola muestra a los parámetros count y nobs solo les pasaremos un dato a cada uno.

https://www.statsmodels.org/stable/generated/statsmodels.stats.proportion.proportions_ztest.html

A esta función tenemos que pasarle:

- count: los éxitos (lo que queremos medir)
- nobs: el tamaño de la muestra
- value: el valor de la hipótesis nula a testar

Como estamos haciendo el contraste contra un dato de la población sólo le pasaremos un valor en count y en nobs, y le pasaremos el value.

Si hiciéramos contraste entre 2 muestras (como en el siguiente ejemplo) entonces tendríamos que pasarle un array con dos números a count y a nobs, y no usaríamos el value.

Como ejemplo vamos a calcular si es posible que el porcentaje de fumadores en la población sea del 40%.

[ ]:
```
#Primero recordamos cual era el valor en nuestra muestra (en porcentaje)
df.smoker.value_counts(normalize = True)
```

[ ]:
```
#Primero recordamos cual era el valor en nuestra muestra (en absoluto)
df.smoker.value_counts()
```

[ ]:
```
#Paso 1: definimos las hipótesis
h0 = 0.4
#Por tanto h1 es que sea diferente del 40%
```

[ ]:
```
#Paso 2: elegimos un nivel de confianza que nos da un alpha.
#Elegimos NC = 95%
alpha = 0.05
```

[ ]:
```
#Paso 3: calculamos el pvalor según el estadístico de contraste elegido (z␣
 ↪sobre una muestra)
exitos = 93 #fijarse que aquí tenemos que poner los fumadores que son los que␣
 ↪nos interesan
muestra = 151 + 93
p_valor = proportions_ztest(count = exitos, nobs = muestra, value = h0)[1]
print(f"{p_valor:.3f}")#Esto es para que no salga con notación científica
```

Por tanto vemos que el pvalor NO es menor que alpha, por tanto no podemos rechazar la H0 y por tanto sí puede ser que en esta población el 40% de las personas sean fumadoras (al 95% de confianza)

Contraste de proporciones entre dos muestras

Queremos ver si una diferencia en la proporción obtenida entre dos muestras puede ser estadísticamente significativa.

Para ello seguiremos la metodología explicada, usando como estadístico de contraste la prueba z.

Scipy no tiene implementación (hasta donde yo sé) para hacer contrastes de proporciones, así que vamos a usar la implementación del paquete statsmodels, con la función proportions_ztest() ya que estamos usando el estadístico z, y como son dos muestras, a los parámetros count y nobs les tendremos que pasar un array.

https://www.statsmodels.org/stable/generated/statsmodels.stats.proportion.proportions_ztest.html

A esta función tenemos que pasarle:

- count: los éxitos (lo que queremos medir)
- nobs: el tamaño de la muestra
- value: el valor de la hipótesis nula a testar

Como estamos haciendo el contraste de dos muestras le pasaremos un array con dos números a count y a nobs, y no usaremos el value.

Como ejemplo vamos a calcular si la diferencia entre el porcentaje de fumadores entre hombres y mujeres puede ser significativa.

[ ]:
```
#Primero recordamos cual era el valor en nuestra muestra (en porcentaje)
pd.crosstab(df.sex, df.smoker,normalize='columns')
```

[ ]:
```
#Primero recordamos cual era el valor en nuestra muestra (en absoluto)
pd.crosstab(df.sex, df.smoker)
```

[ ]:
```
#Paso 1: definimos las hipótesis
#h0: el porcentaje de fumadores en hombres es igual que en mujeres
#h1: el porcentaje de fumadores en hombres NO es igual que en mujeres
```

[ ]:
```
#Paso 2: elegimos un nivel de confianza que nos da un alpha.
#Elegimos NC = 95%
alpha = 0.05
```

[ ]:
```
#Paso 3: calculamos el pvalor según el estadístico de contraste elegido (z␣
 ↪sobre dos muestras)
exitos_h = 60
exitos_m = 33
muestra_h = 60 + 97
muestra_m = 33 + 54

#lo pasamos a array porque así lo pide la función
array_exitos = np.array([exitos_h,exitos_m])
array_muestras = np.array([muestra_h,muestra_m])

p_valor = proportions_ztest(count = array_exitos, nobs = array_muestras)[1]
print(f"{p_valor:.3f}")#Esto es para que no salga con notación científica
```

Por tanto vemos que el pvalor NO es menor que alpha, por tanto no podemos rechazar la H0 y por tanto al 95% de confianza no podemos decir que los hombres y las mujeres fumen en distinto porcentaje.

¿Para qué nos sirve esto en data science?

Como ya hemos comentado en la mayoría de los casos, a nivel práctico, cuando queramos comprobar si el valor de un estadístico que hemos calculado (correlación, chi-cuadrado, coeficientes de modelos, …) es significativo a nivel poblacional, la prueba ya nos devolverá el pvalor, por lo que solo tendremos que compararlo contra un Alpha, que normalmente será 0.05.

Si Pvalor $<$ Alpha entonces el valor SI es significativo.

Pero más allá de que sea prácticamente muy sencillo, es conveninete que sepas de donde viene para entenderlo, aplicarlo e interpretarlo correctamente.

Ejemplo práctico

Para terminar de afianzar los conceptos vamos a hacer un ejemplo práctico sobre uno de los casos de contraste de hipótesis que posiblemente tengas más oportunidad de realizar. Los test A/B.

En general hacer test A/B es configurar un experimento donde queremos comprobar si hay diferencias significativas entre algo nuevo que queremos probar (grupo tratamiento) y lo que ya estaba funcionando (grupo de control).

Es muy común actualmente por ejemplo en marketing digital, donde podemos tener una versión de una web y queremos comprobar si ciertos cambios hacen o no que suba la conversión.

Lo mismo puede ser aplicado a emails, páginas de venta, etc.

En este caso vamos a suponer que tenemos la web actual que llamaremos A. Y una variación que llamaremos B.

Y vamos a asignar aleatoriamente la A a 1000 de los próximos visitantes y la B a otros 1000 también aleatorios.

Vamos a suponer que es una web para registrarse y que 285 personas se registraron en la A y 321 en la B.

Por tanto la conversión es:

- Conversión A: 28.5%
- Conversión B: 32.1%

¿Esa diferencia es estadísticamente significativa y por tanto podemos cambiar la web con garantías?

¿O quizá esa diferencia puede ser explicada simplemente por el azar muestral y por tanto no deberíamos cambiarla todavía?

La hipótesis nula por tanto es que las 2 webs convierten igual, y la alternativa que no convierten igual.

Y vamos a trabajar a un 95% de confianza, por tanto alpha = 0.05.

Ya sabemos que al ser un contraste de proporciones podemos usar el test z.

[ ]:
```
#Vamos a usar la implementación del test z para proporciones de statsmodels

conversiones = np.array([285, 321])
muestras = np.array([1000, 1000])
```

```
pval = proportions_ztest(conversiones, muestras)[1]

print('pvalor igual a: %.3f'%pval)
```

Como el pvalor no es menor que 0.05 entonces no podemos rechazar la hipótesis nula.

Es decir, no tenemos pruebas suficientemente sólidas para poder decir que la B convierte mejor, y por tanto la decisión será que por ahora no vamos a cambiar la web.

03_Estadistica_III_ConceptosAvanzados

September 16, 2021

Contenido

1 ESTADÍSTICA III PARA DATA SCIENCE: CONCEPTOS AVANZADOS

1.1 CONCEPTOS AVANZADOS

En esta sección vamos a ver conceptos avanzados de dos tipos:

- conceptos más avanzados que estaremos usando sobre todo cuando hagamos modelización estadística
- errores de interpretación en el día a día de las conclusiones que se sacan en una empresa, pero que no son correctas.

1.1.1 Valor teórico y error de medida

El valor teórico es una combinación lineal de variables con ponderaciones estimadas empíricamente.

Es decir, cuando hagamos modelos predictivos que vienen del campo de la estadística, como regresiones, lo que estará prediciendo el modelo es el valor teórico.

Pero el valor teórico se puede descomponer en dos componentes: la parte "verdadera"de la variable objetivo que se puede predecir directamente a través de las predictoras, y un error.

Ese error se llama **error de medida** y es lo que siempre estaremos intentando minimizar.

A nivel operativo ese error tendrá dos fuentes principales.

Una es la falta de datos.

Y la otra es la falta de adecuación del algoritmo que estemos usando.

Por tanto en data science siempre mejoraremos:

- Incorporando nuevos y/o mejores datos
- Utilizando algoritmos más apropiados al problema

Y a nivel conceptual ese error puede tener dos fuentes, que son importantes para detectar malas prácticas en la empresa:

- La validez: que estemos midiendo algo que no es lo que en realidad queremos medir
- La fiabilidad: que lo estemos midiendo con los instrumentos adecuados

Por ejemplo un error de validez en la empresa es cuando las agencias intentan convencer a los clientes de que invertir en campañas de "likes" mejorará sus resultados comerciales.

Y un ejemplo de error de fiabilidad es intentar medir la importancia de cada canal en nuestro marketing mediante informes basados en atribución "last click".

1.1.2 Supuestos estadísticos

Las técnicas que utilizaremos en la parte de modelización vienen de dos campos:

- Machine Learning: como árboles de decisión, random forest, redes neuronales, …
- De la estadística: como regresión múltiple, regresión logística

Estas últimas fueron creadas bajo una serie de supuestos, que idealmente deberían cumplirse para que se pueda utilizar la técnica.

Es más, los supuestos deberían cumplirse tanto a nivel individual de cada variable como en el valor teórico combinado.

Pero en la práctica es muy raro que estos supuestos se cumplan, y realmente las técnicas han demostrado ser robustas a las violaciones de los mismos.

En su momento aprenderemos la forma de evaluar los modelos en la práctica para ver si, aún sin cumplir totalmente los supuestos, son modelos útiles o no.

Pero es importante conocerlos al menos a nivel conceptual.

Ello nos llevará a entender mucho mejor este tipo de técnicas y a ser capaz de mejorar su capacidad predictiva.

Los supuestos más importantes son:

- Normalidad
- Heterocedasticidad de varianzas
- Linealidad

- Multicolinealidad

Normalidad Ya conocemos lo que es una distribución normal.

Si la variación con respecto a una normal es suficientemente grande, entonces todos los test estadísticos resultantes (que se basarán en estadísticos como t o F) no serán válidos.

Medir la normalidad multivariante es complicado, por lo que la aproximación práctica suele ser medir y corregir la normalidad univariante de todas las variables que formarán el valor teórico.

Formas de identificarla:

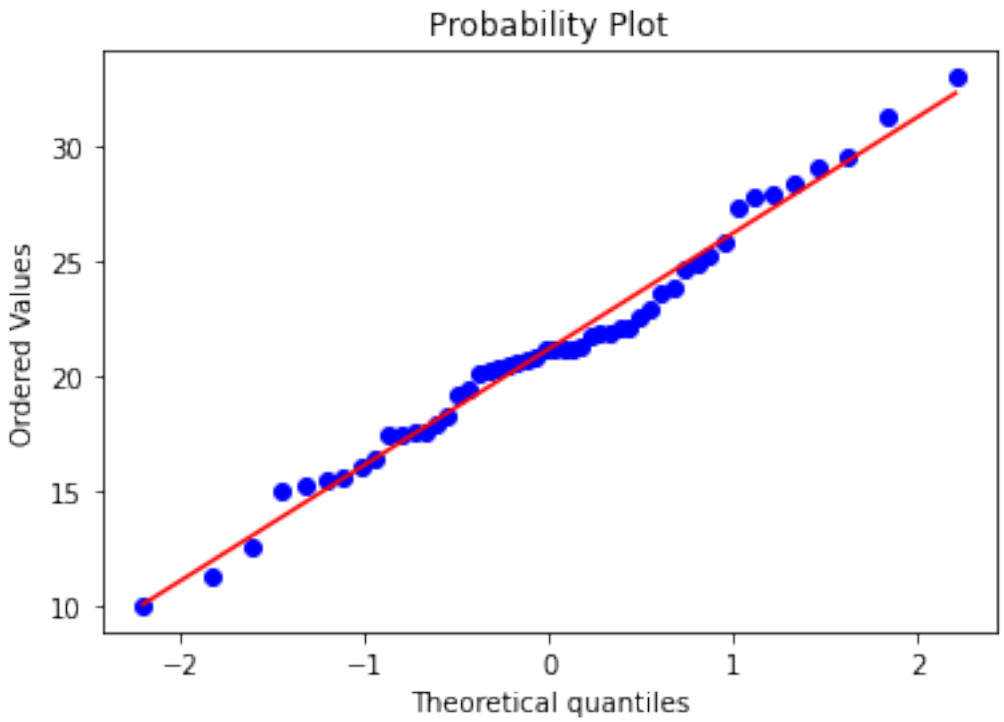
- La normalidad se puede evaluar a nivel gráfico con gráficos como el histograma o el Q-Q.

Solución:

- Transoformaciones de la variable para hacerla más normal: inversa, cuadrado, raiz cuadrada, logaritmo, ...

[3]:
```
#Ejemplo de un qqplot

import matplotlib.pyplot

import scipy.stats

ejemplo = np.random.normal(loc = 20, scale = 5, size=50)

scipy.stats.probplot(ejemplo, dist="norm", plot=matplotlib.pyplot)
```

[3]:
```
((array([-2.20385432, -1.83293478, -1.61402323, -1.45296849, -1.32267759,
         -1.21163342, -1.113805  , -1.02561527, -0.94475674, -0.86964726,
         -0.79915021, -0.73241807, -0.66879925, -0.6077796 , -0.54894415,
         -0.49195112, -0.43651377, -0.38238727, -0.32935914, -0.27724191,
         -0.2258675 , -0.17508277, -0.12474591, -0.07472335, -0.02488719,
          0.02488719,  0.07472335,  0.12474591,  0.17508277,  0.2258675 ,
          0.27724191,  0.32935914,  0.38238727,  0.43651377,  0.49195112,
          0.54894415,  0.6077796 ,  0.66879925,  0.73241807,  0.79915021,
          0.86964726,  0.94475674,  1.02561527,  1.113805  ,  1.21163342,
          1.32267759,  1.45296849,  1.61402323,  1.83293478,  2.20385432]),
  array([ 9.9982128 , 11.25358664, 12.50816057, 15.04766275, 15.19129678,
         15.45240368, 15.59823207, 16.06503015, 16.44925914, 17.48072122,
         17.48595868, 17.54725112, 17.57410167, 17.94965655, 18.29324331,
         19.20796835, 19.41656272, 20.12506503, 20.20314787, 20.37375927,
         20.47667981, 20.61690683, 20.74106137, 20.77267255, 21.10659139,
         21.11733816, 21.18378757, 21.20079063, 21.26711839, 21.74301916,
         21.86393848, 21.89658257, 22.04064297, 22.11068742, 22.57641975,
         22.90455288, 23.58394304, 23.87764212, 24.63303008, 24.86778887,
         25.23203019, 25.80822001, 27.31024167, 27.83580282, 27.95720573,
         28.38674662, 29.08300137, 29.55100675, 31.28380162, 33.00965436])),
 (5.052500398694377, 21.185203710746467, 0.9919537919696491))
```



Heterocedasticidad de varianzas Significa que la varianza de la variable objetivo no es constante en el recorrido de la variable predictora.

Es decir, que para unos valores la predicción será más precisa que para otros.

Formas de identificarla:

- Para variables contínuas: con diagramas de dispersión
- Analizando el gráfico de los residuos (diferencias entre valor predicho y valor real)

Solución:

- En la mayoría de casos es causada por la no normalidad, por lo que corrigiendo la normalidad se corrige también la heterocedasticidad.

Linealidad Se refiere a que exista una relación lineal de cada variable predictora con la target.

Es un supuesto para todas las técnicas que se basen de una u otra forma en la correlación, como la regresión múltiple o la regresión logística.

Formas de identificarla:

- Hacer la matriz de correlaciones de cada predictora con la target
- Gráficos de dispersión de cada predictora con la target
- Análisis de residuos del modelo. Cualquier pauta no lineal visible será la que las variables no han podido explicar linealmente

Solución:

- Linealizar las relaciones mediante transformación de las variables originales
- Usar algoritmos no lineales

Multicolinealidad Se refiere a que exista correlación entre las variables predictoras.

Los modelos que usamos normalmente (aditivos) asumen independencia entre las variables predictoras.

Si eso no se cumple pasará que:

- Podemos estar sobreponderando conceptos
- Podemos causar efectos extraños y variaciones en los modelos: exponentes desproporcionados, signos invertidos, o incluso no convergencia

Formas de identificarla:

- Hacer la matriz de correlaciones entre las variables predictoras
- Identificar durante el desarrollo del modelo variables que apriori deberían predecir pero salen como no predictoras (por la correlación parcial aportada por otras variables)

Solución:

- No meter variables correlacionadas
- Aplicar reducción de variables como Componentes Principales (poco uso en la realidad)

1.1.3 Bootstrapping

Si recuerdas ya introdujimos este concepto cuando hablamos del muestreo.

Y es que de hecho se traduce al español como re-muestreo.

Ya vimos cómo el remuestreo había conseguido demostrar el teorema del límite central, y a partir de ahí todo lo que vimos en la parte inferencial.

Pero tiene más propiedades. Y una de ellas es la capacidad de generalización.

Como veremos en su momento el mayor enemigo que vamos a tener es el sobre ajuste.

Pues bien, hay algoritmos que se basan precisamente en bootstrapping para conseguir evitar ese sobreajuste y ser capaz de predecir mucho mejor ante datos que no han visto nunca.

Lo usan por ejemplo el bagging, que viene de bootstrap aggregating, y básicamente consiste en remuestrear casos y / o variables, hacer un modelo sobre cada una de esas muestras y luego agregarlos para hacer la predicción final.

Por ejemplo Random Forest, que veremos en la parte de modelización, se basa en este principio y es de los algoritmos más estables.

1.1.4 Penetración vs distribución

No es estrictamente un concepto estadístico, pero podemos meterlo dentro de los análisis que más se confunden.

Este efecto está implicado en falacias como:

“Según la DGT el alcohol está implicado en el 30% de los accidentes mortales”.

Luego el alcohol no está implicado en el 70% de los accidentes mortales.

Por tanto podemos concluir que es más peligroso conducir sobrio que borracho.

O su equivalente muy frecuente en el mundo de la empresa:

"El 70% de nuestras compras las hacen clientes varones entre 30 y 45 años, por tanto son los más compradores"

No será cierto si ese perfil fuera el 80% del total de clientes por ejemplo.

En general hay que diferenciar muy bien estos dos conceptos:

- Penetración: qué porcentaje de una base determinada presenta la característica a analizar
- Distribución: qué porcentaje representa un sobconjunto sobre el total

El mejor truco para diferenciarlo es que al sumar penetraciones no tiene por qué sumar 100%.

Pero al sumar distribuciones sí tiene que sumar 100%.

Por ejemplo, decir que el 70% de nuestros clientes son mujeres es un análisis de distribución, y tiene que sumar 100% con el 30% restante que son hombres.

Sin embargo decir que el producto A lo compra el 70% de las mujeres y el 50% de los hombres es un análisis de penetración, y no tiene por qué sumar 100%.

Estos dos son conceptos absolutamente claves cuando estemos haciendo análisis de perfilado, insights o segmentaciones.

1.1.5 Absoluto Vs relativo

3 fuentes de error:

- dar el absoluto sin dar el total
- dar el porcentaje sin dar el término de comparación absoluto
- no tener en cuenta las escalas de los gráficos

Con esto muchas veces se intenta hacer trampas, usando uno u otro según convenga al mensaje que se quiere mandar.

Por ejemplo dar un dato en absoluto, sin dar el total, hará que su efecto parezca mayor o menor.

Decir que este fin de semana han muerto 50 personas en carretera parece mucho. Pero si decimos que ha sido salida de Semana Santa, con 15.000.000 millones de desplazamientos el dato queda matizado.

O al revés, dar el dato en relativo a sabiendas de que la base original es muy pequeña y por tanto parecerá un efecto mayor.

Por ejemplo si decimos que esta semana hemos vendido un 300% más que la anterior parece mucho, pero si la anterior sólo vendimos un producto el dato ya no impresiona tanto.

O jugar con los ejes para que en las comparaciones se perciba un efecto diferente al real, como en este gráfico de RTVE.

[4]:
```
from IPython import display
display.Image("../../99_Media/falsear_con_grafico.png")
```

[4]:

Solución:

- No considerar datos en porcentaje que tengan una base menor a 50 casos
- Siempre poner en contexto ambas dimensiones
- Cuidado con los ejes y las posiciones relativas o absolutas

1.2 REPASO DE LO APRENDIDO

- Descriptiva vs inferencial
- Población y muestra
- Escalas de medida y tipos de variables
- Principales estadísticos para analizar, y cuando usar cada uno
- Principales gráficos para analizar, y cuando usar cada uno
- Chi-cuadrado y correlaciones, y cuando usar cada uno
- Principales distribuciones
- Teorema central del límite y todo lo que supone

- Cómo calcular intervalos de confianza
- Cómo calcular el tamaño de la muestra
- Qué son el Alpha y el Pvalor
- Por qué se hace lo de comparar el pvalor con el Alpha
- Qué son las hipótesis nula y alternativa y cómo se contrastan
- Cómo hacer un contraste de hipótesis de medias en la población
- Cómo hacer un contraste de hipótesis de proporciones en la población
- Cómo hacer un contraste de hipótesis de medias entre dos muestras
- Cómo hacer un contraste de hipótesis de proporcinoes entre dos muestras
- Los principales supuestos estadísticos de los modelos, cómo se identifican y cómo se corrigen
- Fallos más comunes de interpretación y errores a evitar: correlación vs causalidad, penetración vs distribución, absluto vs relativo, etc

05_Estadistica_Ejercicios_Soluciones

October 21, 2021

1 ESTADÍSTICA: SOLUCIONES A EJERCICIOS

Estos ejercicios te permitirán repasar y consolidar lo que has aprendido en el módulo de estadística.

No hagas los ejercicios hasta que hayas estudiado el contenido de su notebook.

IMPORTANTE: el módulo de estadística pretende enseñarte los conceptos de estadística necesarios para data science. El foco no está en usar uno u otro paquete. Por tanto te recomiendo que para resolver los ejercicios no te compliques y uses exactamente los mismos paquetes y funciones que usamos en el módulo (copiar-pegar si quieres) como te digo lo importante es consolidar los conceptos que tendremos que usar en el resto del curso.

Sobre gráficos no haremos ningún ejercicio ya que tenemos un módulo completo dedicado a visualización.

Instrucciones:

- Crea una copia de este notebook antes de escribir nada. De esa forma respetarás el original para volver a hacerlo
- Cada ejercicio tiene unas instrucciones de lo que debes hacer en él
- Bajo las instrucciones estará una celda vacía, aquí es donde tú debes escribir la solución
- Y bajo esta última celda verás que hay otras dos celdas. La primera está vacía y no la debes tocar. Es para que tengas siempre a mano la solución al ejercicio, ya que esta no va a cambiar

[104]:
```
#Te dejo creada la importación de paquetes
#Simplemente ejecuta esa celda para que se cargen
import pandas as pd
import numpy as np
import statistics
import scipy as sp
import seaborn as sns
import random
import math
from statsmodels.stats.proportion import proportions_ztest
```

Vamos a trabajar con un dataset de ventas de productos para hacer los ejerecicios.

Te lo dejo ya cargado como df.

De momento por sencillez le quitaremos todos los nulos, aprendemos mucho más sobre como gestionarlos mejor más adelante en el programa.

[105]:
```
df = pd.read_csv('HistoricoVentas.csv', index_col = 0)
df.dropna(inplace = True)
df
```

[105]:

	Product_Code	Warehouse	Product_Category	Date	Order_Demand
1	Product_1517	Whse_J	Category_019	2014/11/12	200.0
2	Product_1987	Whse_J	Category_005	2014/2/28	700.0
3	Product_0187	Whse_J	Category_007	2016/1/8	3.0
4	Product_1583	Whse_J	Category_005	2014/3/20	100.0
5	Product_0110	Whse_J	Category_032	2013/7/19	360.0
…	…	…	…	…	…
49996	Product_1458	Whse_J	Category_019	2014/6/2	200.0
49997	Product_2082	Whse_A	Category_009	2015/10/21	590.0
49998	Product_0038	Whse_C	Category_005	2015/7/10	5000.0
49999	Product_1393	Whse_J	Category_019	2012/1/17	9000.0
50000	Product_0023	Whse_J	Category_005	2016/6/16	300.0

[49168 rows x 5 columns]

1.1 Ejercicio:

Haz un conteo de frecuencias de cuantas ventas hay de cada categoría de producto.

[106]:
```
df.Product_Category.value_counts()
```

[106]:
```
Category_019    22418
Category_005     4754
Category_001     4634
Category_007     3914
Category_021     2414
Category_006     1686
Category_028     1506
Category_011     1164
Category_015     1045
Category_024      974
Category_009      947
Category_026      708
Category_030      610
Category_032      456
Category_022      403
Category_023      367
Category_018      234
Category_003      185
Category_013      177
Category_020      156
Category_031       85
Category_008       81
Category_033       76
```

```
Category_012       47
Category_010       42
Category_029       37
Category_017       23
Category_004       14
Category_016        5
Category_025        3
Category_002        2
Category_027        1
Name: Product_Category, dtype: int64
```

1.2 Ejercicio:

¿Cual es la categoría de producto más vendida? (saca la moda).

[107]:
```
statistics.mode(df.Product_Category)
```

[107]:
```
'Category_019'
```

1.3 Ejercicio:

¿Cuantas ventas ha habido de cada categoría de producto en cada almacén (Warehouse)? (haz una tabla cruzada)

Consejo: como hay muchas más categorías que almacenes pon las categorías como filas.

[108]:
```
pd.crosstab(df.Product_Category,df.Warehouse)
```

[108]:

Warehouse	Whse_A	Whse_C	Whse_J	Whse_S
Product_Category				
Category_001	8	109	4399	118
Category_002	0	0	0	2
Category_003	96	0	62	27
Category_004	0	0	0	14
Category_005	74	169	4286	225
Category_006	190	41	1303	152
Category_007	706	271	2644	293
Category_008	4	1	76	0
Category_009	359	38	239	311
Category_010	31	0	10	1
Category_011	43	22	1056	43
Category_012	9	5	17	16
Category_013	51	0	109	17
Category_015	238	130	476	201
Category_016	3	0	0	2
Category_017	8	0	15	0
Category_018	117	0	89	28
Category_019	2954	829	17575	1060
Category_020	0	5	0	151

Category_021	634	77	1464	239
Category_022	39	0	79	285
Category_023	20	101	159	87
Category_024	335	97	360	182
Category_025	0	0	0	3
Category_026	171	0	513	24
Category_027	1	0	0	0
Category_028	542	0	887	77
Category_029	37	0	0	0
Category_030	0	96	100	414
Category_031	43	0	42	0
Category_032	0	10	295	151
Category_033	0	0	76	0

1.4 Ejercicio:

Ahora haz lo mismo pero que salga en porcentaje de las categorías de producto(que las filas sumen 100%)

[109]:
```
pd.crosstab(df.Product_Category,df.Warehouse,normalize = 'index')
```

[109]:

Warehouse	Whse_A	Whse_C	Whse_J	Whse_S
Product_Category				
Category_001	0.001726	0.023522	0.949288	0.025464
Category_002	0.000000	0.000000	0.000000	1.000000
Category_003	0.518919	0.000000	0.335135	0.145946
Category_004	0.000000	0.000000	0.000000	1.000000
Category_005	0.015566	0.035549	0.901557	0.047329
Category_006	0.112693	0.024318	0.772835	0.090154
Category_007	0.180378	0.069239	0.675524	0.074859
Category_008	0.049383	0.012346	0.938272	0.000000
Category_009	0.379092	0.040127	0.252376	0.328405
Category_010	0.738095	0.000000	0.238095	0.023810
Category_011	0.036942	0.018900	0.907216	0.036942
Category_012	0.191489	0.106383	0.361702	0.340426
Category_013	0.288136	0.000000	0.615819	0.096045
Category_015	0.227751	0.124402	0.455502	0.192344
Category_016	0.600000	0.000000	0.000000	0.400000
Category_017	0.347826	0.000000	0.652174	0.000000
Category_018	0.500000	0.000000	0.380342	0.119658
Category_019	0.131769	0.036979	0.783968	0.047283
Category_020	0.000000	0.032051	0.000000	0.967949
Category_021	0.262635	0.031897	0.606462	0.099006
Category_022	0.096774	0.000000	0.196030	0.707196
Category_023	0.054496	0.275204	0.433243	0.237057
Category_024	0.343943	0.099589	0.369610	0.186858
Category_025	0.000000	0.000000	0.000000	1.000000

```
Category_026      0.241525  0.000000  0.724576  0.033898
Category_027      1.000000  0.000000  0.000000  0.000000
Category_028      0.359894  0.000000  0.588977  0.051129
Category_029      1.000000  0.000000  0.000000  0.000000
Category_030      0.000000  0.157377  0.163934  0.678689
Category_031      0.505882  0.000000  0.494118  0.000000
Category_032      0.000000  0.021930  0.646930  0.331140
Category_033      0.000000  0.000000  1.000000  0.000000
```

1.5 Ejercicio:

¿Está relacionada la venta de ciertas categorías de productos con ciertos almacenes? ¿o por el contrario se venden igual en todos?

Comprúebalo calculando el pvalor de una prueba de chi-cuadrado. (si es menor que 0.05 es que sí hay relación entre las dos variables).

[110]:
```
tabla = pd.crosstab(df.Product_Category,df.Warehouse)
sp.stats.chi2_contingency(tabla)[1]
```

[110]:
```
0.0
```

1.6 Ejercicio:

Usando la variable ventas (Order_Demand) calcula la media winsorizada un 5% por cada cola

[111]:
```
ventas = df.Order_Demand
winsorizada = sp.stats.mstats.winsorize(ventas, limits = [0.05,0.05])
media = statistics.mean(winsorizada)
print('winsorizada: %.2f'%media)
```

```
winsorizada: 2601.86
```

1.7 Ejercicio:

Usando la variable ventas (Order_Demand) calcula la mediana de las ventas

[112]:
```
print('mediana: %.2f'%statistics.median(ventas))
```

```
mediana: 300.00
```

1.8 Ejercicio:

Ahora calcula la desviación típica y la varianza.

[113]:
```
print('desviación típica: %.2f'%statistics.stdev(ventas))
print('varianza: %.2f'%statistics.variance(ventas))
```

```
desviación típica: 32574.04
varianza: 1061068179.68
```

1.9 Ejercicio:

Voy a crear para ti un nuevo dataset, llamado dfr (de df recortado) con una muestra más manejable que incluya solo las ventas de la categoría de producto 019.

(más adelante aprenderás cómo hacer todo esto, de momento te lo dejo hecho)

[114]:
```
dfr = df[df.Product_Category == 'Category_019']
dfr
```

[114]:

	Product_Code	Warehouse	Product_Category	Date	Order_Demand
1	Product_1517	Whse_J	Category_019	2014/11/12	200.0
6	Product_1463	Whse_J	Category_019	2013/12/5	5000.0
8	Product_1472	Whse_J	Category_019	2015/8/24	300.0
11	Product_1458	Whse_J	Category_019	2015/9/2	4000.0
12	Product_1365	Whse_J	Category_019	2014/4/1	300.0
…	…	…	…	…	…
49991	Product_1381	Whse_J	Category_019	2014/9/24	10000.0
49992	Product_1513	Whse_A	Category_019	2013/6/4	1000.0
49994	Product_1381	Whse_J	Category_019	2014/6/24	1000.0
49996	Product_1458	Whse_J	Category_019	2014/6/2	200.0
49999	Product_1393	Whse_J	Category_019	2012/1/17	9000.0

[22418 rows x 5 columns]

1.10 Ejercicio:

Calcula el error típico de la variable Order_Demand a patir de dfr.

[115]:
```
error_tipico = dfr.Order_Demand.std() / math.sqrt(22418)
error_tipico
```

[115]:
```
311.19219240079184
```

1.11 Ejercicio:

Calcula el margen de error a un 95,5% de confianza.

[116]:
```
margen_error = error_tipico * 2
margen_error
```

[116]:
```
622.3843848015837
```

1.12 Ejercicio:

Calcula el límite inferior y superior del intervalo de confianza a un 95,5% de confianza. (recuerda que deberás calcular también la media)

[117]:
```
media = dfr.Order_Demand.mean()
```

```
inferior = media - margen_error
superior = media + margen_error

print('media: %.2f'%media)
print('limite inferior: %.2f'%inferior)
print('limite superior: %.2f'%superior)
```

```
media: 9137.23
limite inferior: 8514.84
limite superior: 9759.61
```

1.13 Ejercicio:

Calcula qué porcentaje de los registros recogidos en dfr corresponden al producto 1359.

[118]:
```
dfr.Product_Code.value_counts(normalize = True)
```

[118]:
```
Product_1359    0.034570
Product_1295    0.022036
Product_1378    0.021902
Product_1286    0.019582
Product_1382    0.018780
                  …
Product_1907    0.000045
Product_0644    0.000045
Product_1545    0.000045
Product_1625    0.000045
Product_1231    0.000045
Name: Product_Code, Length: 521, dtype: float64
```

1.14 Ejercicio:

Calcula el intervalo de confianza de esa proporción a un 99,7% de confianza.

(Consulta los apuntes para refrescar la fórmula del error típico en proporciones)

[119]:
```
p = 0.034570

q = 1 - p

n = len(dfr.Order_Demand)

error_tipico_prop = math.sqrt((p * q)/n)

margen_error_prop = error_tipico_prop * 3

inferior = p - margen_error_prop
superior = p + margen_error_prop

```

```
print('proporción: %.3f'%p)
print('limite inferior: %.3f'%inferior)
print('limite superior: %.3f'%superior)
```

```
proporción: 0.035
limite inferior: 0.031
limite superior: 0.038
```

1.15 Ejercicio:

Recordemos que la media de ventas en dfr es de 9137$. Pero eso es la muestra que hemos podido recoger.

Sin embargo el director comercial nos dice que realmente la media está en 9500$.

¿Nos podemos creer lo que dice con un nivel de seguridad del 95%?

[120]:
```
#Paso 1: establecemos la hipótesis nula
h0 = 9500

#Paso 2: definimos un alpha
alpha = 0.05

#Paso 3:  calculamos el pvalor según el estadístico de contraste elegido (t␣
 ↪sobre una muestra)
p_valor = sp.stats.ttest_1samp(a = dfr.Order_Demand, popmean = h0)[1]
p_valor
```

[120]:
```
0.24372947061475372
```

1.16 Ejercicio:

Saca de nuevo la tabla cruzada de la categoría de productos cruzada con los almacenes (Warehouse), pero en este caso saca los porcentajes verticales (que las columnas sumen el 100%)

[121]:
```
pd.crosstab(df.Product_Category,df.Warehouse,normalize = 'columns')
```

[121]:

Warehouse	Whse_A	Whse_C	Whse_J	Whse_S
Product_Category				
Category_001	0.001192	0.054473	0.121081	0.028620
Category_002	0.000000	0.000000	0.000000	0.000485
Category_003	0.014301	0.000000	0.001707	0.006549
Category_004	0.000000	0.000000	0.000000	0.003396
Category_005	0.011023	0.084458	0.117971	0.054572
Category_006	0.028303	0.020490	0.035865	0.036866
Category_007	0.105169	0.135432	0.072775	0.071065
Category_008	0.000596	0.000500	0.002092	0.000000
Category_009	0.053478	0.018991	0.006578	0.075431
Category_010	0.004618	0.000000	0.000275	0.000243

Category_011	0.006405	0.010995	0.029066	0.010429
Category_012	0.001341	0.002499	0.000468	0.003881
Category_013	0.007597	0.000000	0.003000	0.004123
Category_015	0.035454	0.064968	0.013102	0.048751
Category_016	0.000447	0.000000	0.000000	0.000485
Category_017	0.001192	0.000000	0.000413	0.000000
Category_018	0.017429	0.000000	0.002450	0.006791
Category_019	0.440042	0.414293	0.483747	0.257094
Category_020	0.000000	0.002499	0.000000	0.036624
Category_021	0.094444	0.038481	0.040296	0.057967
Category_022	0.005810	0.000000	0.002174	0.069124
Category_023	0.002979	0.050475	0.004376	0.021101
Category_024	0.049903	0.048476	0.009909	0.044143
Category_025	0.000000	0.000000	0.000000	0.000728
Category_026	0.025473	0.000000	0.014120	0.005821
Category_027	0.000149	0.000000	0.000000	0.000000
Category_028	0.080739	0.000000	0.024414	0.018676
Category_029	0.005512	0.000000	0.000000	0.000000
Category_030	0.000000	0.047976	0.002752	0.100412
Category_031	0.006405	0.000000	0.001156	0.000000
Category_032	0.000000	0.004998	0.008120	0.036624
Category_033	0.000000	0.000000	0.002092	0.000000

Fíjate en la penetración de ventas de la categoría 006 en los almacenes A y S.

Que es 0.028 y 0.036 respectivamente.

¿Podríamos pensar que esa diferencia es real y esa categoría se vende más en el S? ¿O es simplemente fruto de nuestros datos y no representa una diferencia real?

Queremos estar muy seguros, así que contrástalo al 99% de confianza.

[122]:
```
#Primero sacaremos la tabla cruzada en absoluto y con los marginales
#Ya que lo necesitaremos para lo que le tenemos que pasar a la función
pd.crosstab(df.Product_Category,df.Warehouse,margins = True)
```

[122]:

Warehouse	Whse_A	Whse_C	Whse_J	Whse_S	All
Product_Category					
Category_001	8	109	4399	118	4634
Category_002	0	0	0	2	2
Category_003	96	0	62	27	185
Category_004	0	0	0	14	14
Category_005	74	169	4286	225	4754
Category_006	190	41	1303	152	1686
Category_007	706	271	2644	293	3914
Category_008	4	1	76	0	81
Category_009	359	38	239	311	947
Category_010	31	0	10	1	42
Category_011	43	22	1056	43	1164

Category_012	9	5	17	16	47
Category_013	51	0	109	17	177
Category_015	238	130	476	201	1045
Category_016	3	0	0	2	5
Category_017	8	0	15	0	23
Category_018	117	0	89	28	234
Category_019	2954	829	17575	1060	22418
Category_020	0	5	0	151	156
Category_021	634	77	1464	239	2414
Category_022	39	0	79	285	403
Category_023	20	101	159	87	367
Category_024	335	97	360	182	974
Category_025	0	0	0	3	3
Category_026	171	0	513	24	708
Category_027	1	0	0	0	1
Category_028	542	0	887	77	1506
Category_029	37	0	0	0	37
Category_030	0	96	100	414	610
Category_031	43	0	42	0	85
Category_032	0	10	295	151	456
Category_033	0	0	76	0	76
All	6713	2001	36331	4123	49168

[123]:
```
#En este caso es un contrate de proporciones sobre 2 muestras diferentes
#Una la del almaceén A y otra la del S

#Paso 1: establecemos la hipótesis nula
#h0: la propoción de ventas de la categoría 006 es realmente la misma en los␣
 ↪almacenes A y S

#Paso 2: definimos un alpha
alpha = 0.01

#Paso 3:  calculamos el pvalor según el estadístico de contraste elegido (z␣
 ↪sobre muestras independientes)
exitos_a = 190
exitos_s = 152
muestra_a = 6713
muestra_s = 4123

#lo pasamos a array porque así lo pide la función
array_exitos = np.array([exitos_a,exitos_s])
array_muestras = np.array([muestra_a,muestra_s])

p_valor = proportions_ztest(count = array_exitos, nobs = array_muestras)[1]
print(f"{p_valor:.3f}")
```

```
0.013
```