

Python para cálculos actuariales

16 de abril de 2020

Version: 0.1
Autor: Francisco Gárate
Contacto: fgaratesantiago@gmail.com
Enlace permanente: <https://bit.ly/python-actuarios>

Resumen: Manual de python para cálculos actuariales.

Este documento es una concatenación de las presentaciones utilizadas en el curso de Python del Instituto de Actuarios sin una concienzuda maquetación posterior y, por tanto, la lectura en muchos apartados puede resultar demasiada esquemática y brusca. Aun así, puede servir como un recopilatorio de las principales funcionalidades de Python que podrían añadir valor al trabajo de los actuarios en su día a día. Se acompañan de una serie de ejercicios resueltos que demuestran el potencial que este lenguaje tiene y que cada vez más industrias de distintos sectores, desde farmacéuticas hasta aeronáuticas, utilizan como parte de sus procesos tecnológicos.

Este documento se edita bajo licencia Creative Commons (CC BY-SA 4.0). Además de los usos que permite la ley, queda expresamente autorizado a copiar, distribuir y comunicar públicamente su contenido siempre que se realice sin ánimo de lucro y se mantenga la atribución de la autoría al autor. Todas las colaboraciones serán bienvenidas.

Disclaimer: This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. The author does not take any legal responsibility for the accuracy, completeness, or usefulness of the information herein.

Dedicado a Ana, Mario y Mateo

*Este manual se maqueto en abril de 2020,
durante la cuarenta nacional por
el brote epidémico COVID-19*

Índice

1. Introducción	7
1.1. ¿Qué es la ciencia de datos?	7
1.2. ¿Por qué Python?	7
1.3. Historia de Python	8
1.4. Instalación	9
1.5. Entorno gráfico	9
1.6. Fintech y el Insurtech: APIs y Json	10
1.7. Datos abiertos	11
2. Empezando con Python	12
2.1. Sintaxis	12
2.2. Tipos de objetos	12
2.3. Operaciones básicas	14
2.4. Listas	14
2.5. Rangos, secuencias y selecciones	16
2.6. Incluir comentarios	16
2.7. Operadores lógicos	17
2.8. Condicionales	17
2.9. Bucles	18
2.10. Creación de funciones	19
2.11. Manejo de errores y excepciones	20
2.12. Ejercicio 1. Creación de una función	21
2.13. Librerías incluidas	22
2.13.1. Math	22
2.13.2. csv	22
2.13.3. Otras librerías y funciones útiles	23
2.14. Instalar librerías	23
2.15. version y dir()	24
3. Operaciones matemáticas: NumPy	25
3.1. Indexación	26
3.2. Operaciones con vectores	26
3.3. Types	27
3.4. Operaciones con vectores	28
3.5. Estadísticos	28
3.6. Exponenciales, logaritmos, constantes	30
3.7. Matrices	30
3.8. Ejercicio 2	33
3.9. NumPy Financial	34
3.10. Ejercicio 3	36
3.11. Simulaciones	37
3.12. Números aleatorios	37
3.13. Álgebra lineal con Numpy	39
4. Visualización de datos: Matplotlib	40
4.1. Tipos de gráficas	41
4.2. Histogramas	43
4.3. Estilos	45
4.4. Seaborn	45
4.5. Bokeh	46
4.6. Ejercicio 4	47

5. Análisis de datos: Pandas	48
5.1. Series	48
5.2. DataFrame	49
5.3. Importación desde texto plano (csv, txt, tab...)	52
5.3.1. Importación desde Excel	53
5.3.2. Importación desde url	53
5.3.3. Exportar a CSV o Excel	54
5.4. Ejercicio 5	55
5.5. Vista y verificación de datos	56
5.6. Limpieza de datos	56
5.7. Fechas	57
5.8. Series temporales	57
5.9. Ejercicio 6	59
5.10. Operaciones dentro de un dataframe	60
5.10.1. Añadir columnas	60
5.10.2. Borrar filas o columnas	60
5.10.3. Función lamdba	61
5.11. Selección de datos	63
5.12. Estadísticos	64
5.13. Ejercicio 7	65
5.14. Ordenación	66
5.15. reindex y rename	67
5.16. Bootstrap	67
5.17. Agregación de datos: groupby	69
5.18. Ejercicio 8	71
5.19. Tablas pivotantes: pivot	72
5.20. Ejercicio 9	73
5.21. Gráficos con matplotlib y pandas	74
6. Chain-Ladder con Python	77
6.1. Ejercicio 10	85
7. Probabilidad y estadística	86
7.1. Pandas y Numpy	86
7.2. Scipy	89
7.3. Distribuciones de probabilidad con SciPy	89
7.4. Ejercicio 11	94
7.5. Creación de intervalos	95
7.6. Ejercicio 12	96
7.7. Otras distribuciones conocidas	97
7.7.1. Distribución lognormal	97
7.7.2. Distribución gamma	98
7.8. Ejercicio 13	99
8. Estudio de muestras	101
8.1. Intervalos de confianza	101
8.2. Estadísticos de contraste	105
8.3. Contraste de hipótesis	106
8.4. Bootstrap	108

9. Modelos estadísticos	111
9.1. Regresión lineal	112
9.2. Regresiones múltiples	116
9.3. Modelo lineal generalizado (GLM)	116
9.4. Máxima verosimilitud	118
9.5. Modelos de Ajuste a una distribución	118
9.6. Ejercicio 17	119
9.7. Bondad del ajuste: Calidad estadística	122
9.7.1. Test de Kolmogorov-Smirnov	122
9.7.2. Test de Chi cuadrado o X^2 de Pearson	123
9.7.3. Test de Anderson-Darling	123
9.7.4. Test de Shapiro-Wilk	124
9.7.5. Test de Jarque-Bera	124
9.8. Gráfica Q-Q	125
9.9. Ejercicio 18	127
10. Modelización de seguros de Vida	129
10.1. Manejo de fechas y edades	130
10.2. Cálculo de edades	131
10.3. Ejercicio 19	133
10.4. Generación de series temporales	134
10.5. Ejercicio 20	135
10.6. Librería Pyliferisk	136
10.7. Ejercicio 21	140
10.8. Rentas actuariales:	141
10.9. Ejercicio 22	144
10.10. Flujos de caja con Pandas	145
10.11. Ejercicio 23	147
11. Aumento de velocidad de cálculo	149
11.1. Cython: Compilar código Python en C	149
11.2. Cloud computing	151

Sobre este manual

Este documento es una concatenación de las presentaciones de \LaTeX utilizadas en el curso de Python del Instituto de Actuarios Españoles sin apenas realizarse una concienciada maquetación posterior y, por tanto, la lectura en muchos apartados puede resultar demasiada brusca y esquemática.

Las referencias internas dentro del documento pueden no coincidir, al igual que la numeración de los ejercicios sigue el orden de las sesiones del curso que no coinciden con los apartados del presente documento.

A finales de 2013 liberé en Github la librería `pyliferisk` (librería escrita en python para modelizar contingencias de vida), y en los últimos años recibo gran cantidad de correos electrónicos con comentarios y aportaciones de actuarios de todo el mundo (principalmente US, UK, UAE y países del sureste asiático), demostrándome la capacidad que tiene el software libre para unir a personas con intereses comunes. Espero que este manual también ayude a compartir conocimientos dentro de esta profesión y a afrontar el futuro tecnológico que auguran al sector asegurador.

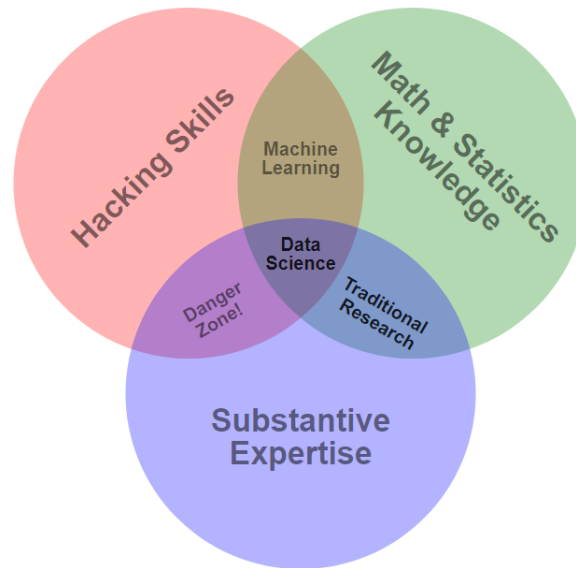
Puede consultarse versiones actualizadas de este manual en el siguiente link: <https://bit.ly/python-actuarios>

Adicionalmente, la base de datos de siniestros ficticios, utilizada en varios ejercicios, puede descargarse en el siguiente link: <https://www.kaggle.com/franciscogarate/fictitious-cases-of-insurance-claims>

1. Introducción

1.1. ¿Qué es la ciencia de datos?

Data Science Venn Diagram (Drew Conway, 2010)



- **Habilidades Hacker.** Ser capaz de manipular archivos de texto en la línea de comandos, comprender operaciones vectorizadas, pensamiento abstracto, identificación de patrones ganadores, pensar algorítmicamente, etc.
- **Conocimientos matemáticos y estadísticos:** familiaridad básica con estas herramientas, ejemplo: qué es una regresión de mínimos cuadrados ordinarios y cómo interpretarla.
- **Experiencia sustancial,** es decir, abarcar más allá del mundo académico y haberse enfrentado a casos reales. (No vale el: *Supongamos una vaca esférica*).

1.2. ¿Por qué Python?

- El conocimiento de lenguajes de programación se ha convertido en algo indispensable para los actuarios.
- Python es uno de los lenguajes más utilizados en la denominada ciencia de datos (*data science*). Es fácil de escribir y de ser entendido sin necesidad de tener conocimientos avanzados de programación.
- Posee una de las mayores comunidades de desarrolladores (principalmente *open-source*), siendo uno de los lenguajes más populares en las comunidades de soporte (Stackoverflow, Github...), muy por encima de cualquier otro lenguaje.
- Python tiene soporte nativo en las principales bases de datos (SQL, DB2, Oracle, SAP DB...) y así como aquellas bases de datos utilizadas en Big Data (Azure SQL, Hadoop, Apache Beam, MongoDB o CDH).
- Los lenguajes interpretados (como R, Python o Julia) consisten en líneas de instrucciones o scripts que son interpretados en tiempo real, por tanto no requieren ser compilados (como si sucede con C o C++) y pueden ser ejecutados en cualquier sistema operativo e incluso bajo entornos web.

- Suele decirse que Python es la navaja suiza de los lenguajes de programación.
- El lenguaje Python fue creado por el holandés Guido van Rossum en 1990. La mentalidad con qué fue creado hace que sea muy intuitivo y uno de los lenguajes más fáciles de aprender.

1.3. Historia de Python

Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office (a government-run research lab in Amsterdam) would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).

Prólogo de Guido van Rossum para el libro "Programming Python" (1st ed. 1996), publicado por O'Reilly.

<https://www.python.org/doc/essays/foreword/>



Guido van Rossum ✓
@gvanrossum

Today's Python history lesson: Python took its control and data structures from ABC, its identifiers, strings and %-string formats from C, and its regular expressions from Perl. But its # comments (and #!) and -c command line flag came from the UNIX v7 shell.

Traducido del inglés al 

Lección de historia de Python de hoy: Python tomó sus estructuras de control y datos de ABC, sus identificadores, cadenas y formatos de% de cadena de C, y sus expresiones regulares de Perl. Pero sus # comentarios (y #!) y el indicador de línea de comando -c vino del shell de UNIX v7.

7:58 a. m. · 30 ene. 2020 · [Twitter Web App](#)

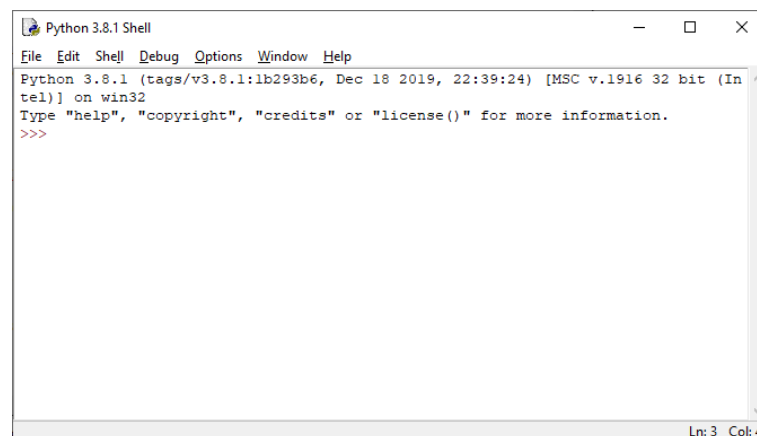
1.4. Instalación

- Python 3.8
- Site: www.python.org
- Licencia: GPL-compatible (open-source)

Python 2.7 *release* (última versión de Python 2) ha dejado de tener soporte en 2020 (PEP 373). Así, las principales librerías han dejado de actualizar su versiones destinadas a Python 2 (como NumPy desde 2019).

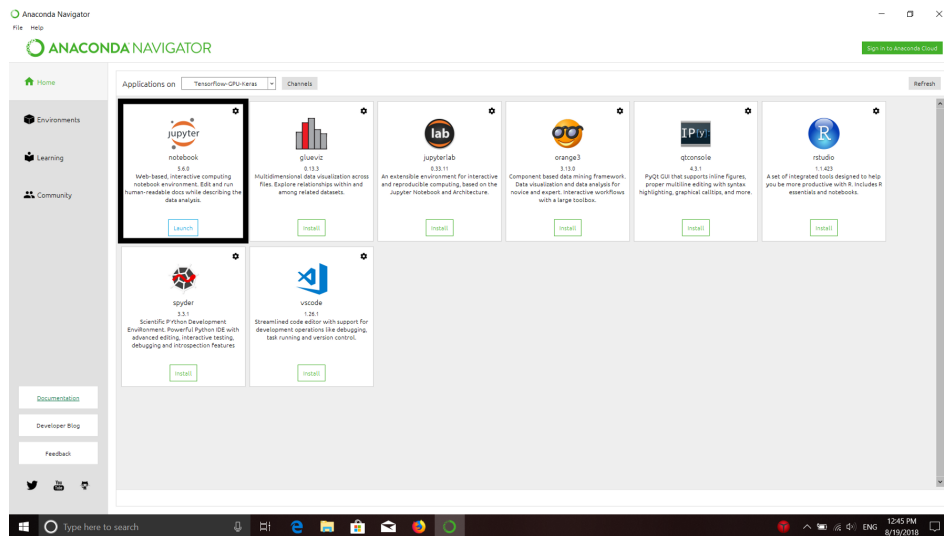
1.5. Entorno gráfico

- Aunque python posee su propio IDE, denominado IDLE (Integrated DeveLopment Environment for Python), éste es un entorno gráfico de desarrollo elemental que permite editar y ejecutar programas en Python.
- Se recomienda instalar un entorno gráfico más amigable.
- Existen múltiples opciones: Visual Studio Code, Sublime Text, Atom, Jupyter Notebook...



Otro IDE all-in-one recomendado es el pack de Anaconda Distribution. Anaconda Distribution 2019.10 que incluye Python 3.7, Spyder, Jupyter Notebook y las principales librerías de Python (y además incluye R y R-Studio).

- Site: <https://www.anaconda.com>
- Licencia: 3-clause BSD License (open-source)



1.6. Fintech y el Insurtech: APIs y Json

- Tanto el **Fintech** y el **Insurtech** esta basado en APIs.
- Una API (application programming interface) es una interfaz que devuelven via HTTPS información en formato json (o XML).
- Python se ha convertido *de facto* en el lenguaje para implementar API debido a su flexibilidad e interoperatividad con otros sistemas.
- Algunas son públicas y abiertas, y en otras hace falta autenticarse con clave (OAuth token).
- La **Directiva PSD2** (Payment Service Providers 2) obliga a todas las instituciones financieras de la UE a desarrollar conexiones seguras e interoperables con terceras empresas a través de API abiertas. Una de las primeras entidades en desarrollarlas ha sido BBVA (BBVA API Market: <https://www.bbvaapimarket.com>)
- La banca se esta adaptando (open banking, portabilidad de datos), y el **sector asegurador** es el siguiente (open insurance).

Open Insurance

- **Modularidad:** Cotizaciones on-line en modo “taller”. Personalización completa del producto. Dime la prima dispuesta a pagar y te creo un producto a medida.
- **Suscripción:** Aceptación del riesgo al momento.
- **Siniestros:** Gestión individualizada a nivel de garantías.
- **API:** Interconexión entre los aplicativos de la compañía. Ejemplo: un Host programado en Cobol y una app en Swift (iOS) deben entenderse.
- **Regulación:** Transferencia de documentación ágil al mediador y al asegurado (propuestas de seguros, condiciones generales y particulares, pólizas, consentimientos, etc.)
- **Pagos:** Blockchain Insurance Industry Initiative (B3i) Ejemplo: Sistema de pagos blockchain en la gestión del reaseguro y coaseguro.
- **Experiencia del cliente:** Capacidad de interactuar racional, física o emocional con cualquier parte de una empresa (Internet of Things).

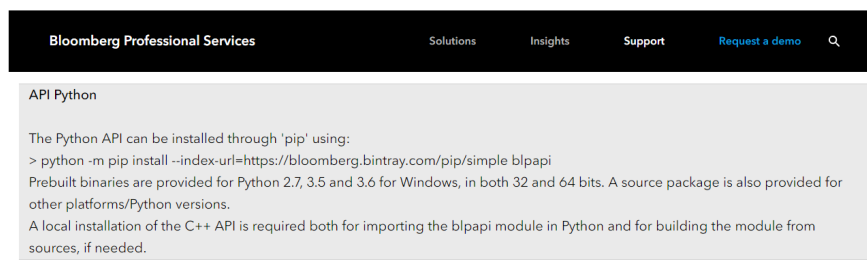
1.7. Datos abiertos

Acceso fácil y rápido a infinidad de datos. APIs útiles accesibles con Python:

- **datos.gob.es:** proporciona acceso al Catálogo de datos de datos.gob.es (25.850 conjuntos de datos a dic-2019): <https://datos.gob.es/es/apidata>
- **API JSON INE:** permite acceder mediante peticiones URL a toda la información disponible en Inebase en formato JSON.
- **inclasns.msssi.es:** permite acceder al repositorio de más de 240 indicadores clave del Sistema Nacional de Salud. (Requiere clave) <http://inclasns.msssi.es/doc/api>
- **Google Maps Platform:** The Geocoding API is a service that provides geocoding and reverse geocoding of addresses (de pago).
- **Openstreetmap API:** proyecto colaborativo de mapas de uso libre bajo licencia abierta.
- **AEMET OpenData:** API REST desarrollada por AEMET que permite la difusión y reutilización de la información meteorológica y climatológica de la Agencia. <https://opendata.aemet.es>
- **OpenWeatherMap:** Datos meteorológicos, con datos históricos de +37.000 ciudades <https://openweathermap.org/api>

Ecosistema Python

Bloomberg ofrece a sus suscriptores una API para interactuar con los datos de mercados financieros



The screenshot shows the Bloomberg Professional Services website. The header is dark with white text for 'Bloomberg Professional Services' and navigation links: 'Solutions', 'Insights', 'Support', 'Request a demo', and a search icon. Below the header, there's a section titled 'API Python'. The text in this section reads: 'The Python API can be installed through 'pip' using: > python -m pip install --index-url=https://bloomberg.bintray.com/pip/simple blpapi'. It also mentions that prebuilt binaries are provided for Python 2.7, 3.5 and 3.6 for Windows, in both 32 and 64 bits, and that a source package is also provided for other platforms/Python versions. Finally, it states that a local installation of the C++ API is required both for importing the blpapi module in Python and for building the module from sources, if needed.

<https://www.bloomberg.com/professional/support/api-library/>

2. Empezando con Python

2.1. Sintaxis

- La sintaxis es concisa y se utiliza de forma coherente en todas las bibliotecas
- La puntuación se utiliza con moderación para separar los bloques de código, lo que hace que el código Python sea muy legible
- El código Python se interpreta a la vez que se ejecuta, en vez de ser compilado*
- No hay necesidad de definir las variables previamente
- Una misma instrucción puede escribirse de varias formas (lo que se conoce como azúcar sintáctico)
- Para sistemas Unix, se aconseja empezar el código con la ruta de Python en el sistema: `#!/usr/bin/python`, ya que python es un lenguaje scripting ejecutable desde consola: `(./myscript.py, en lugar de python myscript.py)`. También es útil cuando disponemos 2 versiones de Python instaladas.

2.2. Tipos de objetos

En Python no haría falta declarar el **tipo de variable** al darle un valor o leerla de una fuente externa, ya que interpreta y asigna el tipo de variables más adecuado. Los más comunes serían:

- **str**: String o cadena de caracteres
- **int**: Números enteros
- **float**: Número con decimales
- **bool**: Booleano: Verdadero o falso (True/False)
- **datetime**: Fecha

Para conocer el tipo de variable se usa **type()**, que nos devolvería: int, str, float, datetime o bool. Del mismo modo, podemos utilizar las siguientes funciones para hacer **conversiones** entre varios tipos:

- **str()**: convierte a carácter
- **int()**: convierte a número entero
- **float()**: convierte a número con decimales

```
1 a = 5
2 b = 25
3 print(a * b)
```

125

```
1 a = 5
2 b = 25.
3 print(a * b)
```

125.0

```
1 a = 5
2 b = 'c'
3 print(a * b)
```

ccccc

```

1 valor = 123
2 print('El resultado final es: ' + str(valor) + ' unidades')
3 print(valor*3)
4
5 texto = "Buenas tardes!"
6 print(texto*3)
7 print(texto[1])
8
9 print(valor[1])

```

```
El resultado final es: 123 unidades
```

```
369
```

```
Buenas tardes!Buenas tardes!Buenas tardes!
```

```
u
```

```
TypeError: 'int' object is not subscriptable
```

Un recurso muy utilizado a la hora de mostrar un output de python es formatear la salida:

print("Texto%s" % (variable))

Pudiendo elegir entre los siguientes tipos de formatos:

- **%d**: decimal
- **%i**: número entero
- **%o**: octal
- **%x**: hexadecimal
- **%f**: flotante (puede fijarse el número de decimales con **%.2f**)
- **%s**: string (texto)

```

1 valor = 123.12456
2 print('El entero es %i y con 2 dec. es %.2f' % (valor, valor))

```

```
El entero es 123 y con 2 dec. es 123.12
```

2.3. Operaciones básicas

Son fácilmente adivinables:

- + Suma
- − Resta
- * Multiplicación
- / División
- ** Exponencial
- // Cociente de la división
- % Resto de la división

```
1 a = 5
2 b = 2
3 print(a + b)
4 print(a - b)
5 print(a * b)
6 print(a / b)
7 print(a ** b)
8 print(a // b)
9 print(a % b)
```

```
7
3
10
2.5
25
1
2
```

2.4. Listas

En Python, al igual que en otros lenguajes de programación, se empieza a contar a partir del 0.

Las listas pueden ser construidas de varias maneras:

- Usando un par de corchetes para denotar una lista vacía: []
- Usando corchetes, separando los elementos con comas: [a], [a, b, c]
- Usando la función `list()` sobre un objeto que sea iterable (por ejemplo un rango de valores).

```
1 x = []
2 y = [50, 51, 52, 53, 54, 55]
3 z = list(range(10))
```

Operar con listas:

- **s[i] = x** el item i de la lista s es reemplazada por x
- **s[i:j] = t** El contenido del segmento de s (desde i a j) es reemplazado por el contenido de t
- **s.append(x)**: añade x a la lista s

```
1 s = []  
2 s.append(10)  
3 print(s)
```

10

- **s.clear()**: Elimina todos los items de s
- **s.copy()**: crea una copia de s. Ejemplo **a = s.copy()**
- **s.insert(i, x)**: inserta x dentro de s en el índice dado por i
- **len(s)**: longitud s
- **min(s)**: El valor más pequeño (mínimo) de s
- **max(s)**: El valor más grande (máximo) de s
- **round(number[, ndigits])**: Redondea con los decimales establecidos. Por defecto sin decimales.
- **sort()**: ordena los elementos de la lista.
- **s.count(x)**: número total de ocurrencias de x en s

```
1 x = [25, 45, 36, 12, 25, 65]  
2 print(x.count(25))
```

2

- **abs()**: Devuelve el valor absoluto de un número
- **x in s** Devuelve True si un item de s está presente en x, de lo contrario False
- **x not in s** Devuelve False si un item de no está en x, sino True

```
1 print('C' in 'ABCD')
```

True

- **all()**: Devuelve True si todos los elementos del iterable son verdaderos (o si el iterable está vacío)
- **any()**: Devuelve True si algún elemento del iterable es True. Si el iterable está vacío, devuelve False.

2.5. Rangos, secuencias y selecciones

Los rangos son un recurso muy utilizado.

- **range(i, j)**: Crea un rango del valor *n* al valor *m* (*j* debe $\geq i$, sino será vacío)
- **range(i, j, k)**: en *k* pasos (*k* puede ser un número negativo. Ejemplo `range(100,0,-1)`)

Desde Python 3, para que los rangos puedan interactuar con funciones como si fueran listas hay que convertirlo previamente en lista utilizando **list(range())**

```
1 a = list(range(10,0,-1))
2 print(a)
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Para seleccionar los valores de una lista se utilizan los corchetes, teniendo en cuenta que en Python **se empieza a contar por cero**.

- **lista[n]**: Selecciona el valor de la *n*-ésima posición
- **lista[n:m]**: Selecciona el valor de la *n*-ésima a la *m*-ésima posición.
- **lista[n:m:i]**: Selecciona el valor de la *n*-ésima a la *m*-ésima posición con saltos de *i*

En caso de omitirse *n* y/o *m*, se entenderá que la selección es desde el principio hasta el final. Si fueran negativos, se entiende que dicho valor negativo se resta al valor del inicio o final.

```
1 lista = list(range(10))
2 print(lista[5:9])
3 print(lista[::-1])
4 print(lista[: -1])
```

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

2.6. Incluir comentarios

Una buena costumbre a la hora de programar es **comentar el código** para facilitar el entendimiento de una tercera persona, o incluso de nosotros mismo si ha pasado tiempo desde que lo programamos.

Con **#** y **'''(triple apóstrofe ')** se puede comentar una línea o un grupo de líneas respectivamente:

```
1 #!/usr/bin/python
2 # Esto es un comentario en una linea
3 a = 5
4 b = 2
5 ''' Esto son varias
6 lineas comentadas
7 en python '''
8
9 a + b # Resultado
```


2.7. Operadores lógicos

Los operadores lógicos se utilizan principalmente a la hora de realizar funciones, en donde se deba interactuar con las variables o listas:

- `==` Igual que
- `!=` No es igual que
- `<>` Diferente que (equivalente a `!=`)
- `>` Mayor que
- `<` Menor que
- `>=` Mayor o igual que
- `<=` Menor o igual que

Que a su vez pueden enlazarse con la condición OR y AND:

- `or`
- `and`

2.8. Condicionales

- El funcionamiento de `if` es fácil: se evalúa la condición, si es verdadera se ejecuta el código, si es falsa, no se ejecuta nada o se ejecuta otro código.

```
1 if True:
2     print('Se cumple la condicion')
```

- En cualquier función `if .. else`, los **:** (dos puntos) y el **espaciado** son necesarios.
- La instrucción **`pass`** es una operación nula y no pasa nada al ejecutarse. No es obligatorio utilizar **`else`** en un condicional. Suele usarse para rellenar funciones inacabadas.

```
1 if x >= 65:
2     print('El valor es mayor o igual a 65')
3 else:
4     pass
```

- La instrucción **`continue`** finaliza el bucle hasta ese punto, aunque continua ejecutándose el bucle (es decir, ignora todo lo que venga a continuación de `continue`).
- Puede haber tantos `if` como se quiera con la expresión **`elif`** (*else if*)
- La ejecución o lectura del código finalizará en la primera condición que se cumpla.

```
1 x = 60
2 if x >= 65:
3     print('El valor es mayor o igual a 65')
4 elif x > 50:
5     print('El valor es mayor que 50')
6 else:
7     pass
```

```
El valor es mayor que 50
```

Otra forma compactada (en una sola línea) de hacer condicionales:

```
1 x = 10
2 print('mayor igual a 65') if x >= 65 else print('menor a 65')
```

```
menor a 65
```

- La instrucción **if** puede usarse también con **in** o **not in**:

```
1 producto = '0117'
2 cuota_parte = 0.65
3
4 if producto in ('0116', '0117'):
5     cuota_parte = 0.75
6
7 print(cuota_parte)
```

```
0.75
```

2.9. Bucles

- Los bucles **for** son estructuras que repiten un bloque de instrucciones un número determinado de veces.
- El bloque de instrucciones que se repiten se llaman cuerpo del bucle y cada repetición se llama iteración:
- Los bucles son un recurso muy utilizado en la programación de Python:

```
1 for i in range(5):
2     print(i)
```

```
0
1
2
3
4
```

Utilizar el bucle **for** para anexar datos a una lista es una funcionalidad muy utilizada:

```
1 s = []
2 for i in range(5):
3     s.append(i)
4
5 print(s)
```

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

- Toda lista es iterable:

```
1 subriesgos = ['Mercado', 'Salud', 'Contraparte', 'Vida', 'No
2 Vida']
3 for i in subriesgos:
4     print(i)
```

```
Mercado
Salud
Contraparte
Vida
No Vida
```

- Y puede ser numerada:

```
1 subriesgos = ['Mercado', 'Salud', 'Contraparte', 'Vida', 'No
2 Vida']
3 for i in enumerate(subriesgos):
4     print(i)
```

```
(0, 'Mercado')
(1, 'Salud')
(2, 'Contraparte')
(3, 'Vida')
(4, 'No Vida')
```

2.10. Creación de funciones

- Podemos crear nuestras propias funciones fácilmente con la palabra reservada **def**:
`def nombre-funcion(argumentos):`
- Los **argumentos** van entre paréntesis y son opcionales e incluso se pueden especificar valores por defecto.
Ej.: `def nombrefun(x, y=100)` o `def nombrefun(x, y, *args)`
- A continuación, se introduce el cuerpo de la función con una tabulación (o 4 espacios según guía de estilo PEP 8).
- El resultado de la función deberá devolverse con la instrucción **return**.

Ejemplo de función que elevada al cuadrado el número dado:

```
1 def nombrefuncion(x):
2     m = x ** 2
3     return m
4
5 print(nombrefuncion(5))
```

25

- Cuando está previsto utilizar nuestras funciones (o variables) creadas en varias partes de un proyecto o en varios proyectos, estas pueden guardarse en un fichero único y ser importado como si de un paquete o librería se tratara.

- Para ello, dichas funciones deben guardarse en un fichero .py dentro de la misma carpeta.
- Así, nuestra función puede ser llamada aunque precedida con el nombre del fichero .py que la contiene y que previamente ha sido importado usando **import fichero**

```
1 import intro_def
2 print(intro_def.nombrefuncion(6))
```

36

- Otra opción:

```
1 from intro_def import nombrefuncion
2 print(nombrefuncion(7))
```

49

2.11. Manejo de errores y excepciones

La declaración **try .. except** puede resultar bastante útil en nuestras funciones:

```
1 def divide(x, y):
2     try:
3         return(x / y)
4     except:
5         return 0.
```

Puede especificarse el mensaje o instrucción por tipos de errores:

- ZeroDivisionError: Error de división por cero.
- TypeError: la función se aplica a un objeto de tipo inapropiado.
- ValueError: el valor no está en el rango de valores posibles
- NameError, RuntimeError
- KeyboardInterrupt: cuando se interrumpe durante la ejecución (p.e. Control-C)
- Otros: OSError, BufferError, MemoryError, LookupError, IndexError, KeyError,

```
1 def divide(x, y):
2     try:
3         return(x / y)
4     except ZeroDivisionError:
5         print('Division por cero!')
6     except (TypeError, NameError, RuntimeError):
7         pass
8
9 print(divide(2, 0))           #ZeroDivisionError
10 print(divide('ABCD', 2))     #TypeError
```

2.12. Ejercicio 1. Creación de una función

Ejercicio_01_01

La tasa de bajas de nuestra cartera es del 2.5% cada mes, excepto los meses de Julio y Agosto que es un 1%, y los meses impares que sube al 3.5%.

Se pide, crear un función que introduciendo el número del mes nos devuelva la tasa de baja mensual, mostrando un mensaje de error para meses que no correspondan con los números enteros del 1 al 12.

```
1 def tasa_bajas(mes):
2     try:
3         if mes > 12:
4             raise ValueError
5
6         if isinstance(mes, float):
7             raise TypeError
8
9         if mes == 6 or mes == 7:
10            return .01
11        elif mes % 2 != 0:
12            return .035
13        else:
14            return 0.025
15
16    except(TypeError):
17        print("El mes debe ser un valor entero")
18
19    except(ValueError):
20        print("El valor no esta dentro del rango")
21
22 for i in range(1,13):
23     print(i, tasa_bajas(i))
24
25 print(tasa_bajas(5.5))
26 print(tasa_bajas('12'))
27 print(tasa_bajas(15))
```

NOTA: Ejemplo ilustrativo. Si se diera justo este caso, la mejor opción sería crear una lista 'tasa_bajas[0.035, 0.025..]' obteniendo el dato con un simple 'tasa_bajas[mes-1]'.

2.13. Librerías incluidas

<https://docs.python.org/3/library/>

Las más útiles:

- **csv**: Lectura y escritura de ficheros csv
- **datetime**: Operar con fechas
- **decimal**: Aporta mayor precisión a operaciones con decimales (usa código C)
- **gzip**, **zlib**, **bz2**, **zipfile**, **tarfile**: para manejo de ficheros comprimidos
- **math**: alternativa ligera de numpy (exp, sqrt...)
- **itertools**: Permite crear combinaciones y permutaciones fácilmente.
- **os**: Acceso directo al sistema operativo (OS) a través de comandos.
- **sys**: Obtiene información del sistema. Útil para añadir seguridad al código.
- **tempfile**: manejo de ficheros temporales que pueda requerir nuestro código
- **urllib**: Acceso al código html de cualquier url pública.

2.13.1. Math

- **Math** permite acceder a operaciones matemáticas sin necesidad de cargar otras librerías mas complejas en memoria, como pudiera ser Numpy.
- La función **dir()** lista todas las funciones que posee cualquier librería:

```
1 import math
2 print(dir(math))
```

```
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc'
```

```
1 print('La raiz de 4 es:', math.sqrt(4))
```

```
La raiz de 4 es: 2.0
```

2.13.2. csv

- La librería **csv** permite la lectura y escritura de ficheros de csv. Aunque dicha funcionalidad es incluso más fácil con librerías como pandas, en ocasiones es útil poder acceder a ficheros de texto con librerías estándar de Python.

```
1 import csv
2
3 with open('polizas.csv','r') as fichero:
4     polizas = csv.reader(fichero,delimiter=';')
5     for linea in polizas:
6         print(linea)
```

2.13.3. Otras librerías y funciones útiles

- Algunas librerías, como `urllib`, fueron revolucionarias al principio, aunque a día de hoy han sido superadas por otras más avanzadas.
- En el caso de `urllib`, que sirve para descargarse mediante reglas el contenido de páginas web accediendo a su código html (el conocido como Web Scrapping), librerías como `beautifulsoap` o `scrapy`, permiten por ejemplo loguearse en webs y obtener la información buscada fácilmente.
- la librería `sys` y `os`, permiten un fácil uso de información del sistema, así como ejecutar comandos del sistema, utilizando el resultado de dichos comandos como input de nuestro código.

Otras funciones útiles:

- `zip(x, y)`: Empareja posición a posición dos listas (deben tener la misma longitud)

```
1 a = [1, 2, 3, 4, 5]
2 b = [6, 7, 8, 9, 10]
3 z = list(zip(a,b))
4 print(z)
```

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

- `isinstance(x, type)`: Devuelve True o False dependiendo del tipo de variable. La instrucción `raise` fuerza el error:

```
1 x = 5
2 if isinstance(x, int):
3     pass
4 else:
5     raise ValueError
```

2.14. Instalar librerías

- Además de las librerías que Python dispone de serie, es posible instalarse cualquiera de las librerías que están disponibles con un simple comando: `pip install nombre`
- El repositorio de Python se encuentra disponible en <https://pypi.org>, existiendo actualmente más de 165.000 librerías/bibliotecas disponibles para su instalación.
- Muchas librerías de python tienen dependencias a otras. En ese caso, deben instalarse conforme van siendo demandadas.
- Para desinstalar una librería: `pip uninstall nombre`
- Para actualizar una librería: `pip install nombre --upgrade`
- Por último, para saber las librerías que tenemos instaladas: `pip list`

2.15. version y dir()

Mostrar la versión y funciones incluidas de una librería:

```
1 import numpy
2 print(numpy.__version__)
3 print(dir(numpy.random))
```

1.18.1

'Lock', 'RandomState', 'absolute_import', 'beta', 'binomial', 'bytes', 'chisquare', 'choice', 'dirichlet', 'division', 'exponential', 'f', 'gamma', 'geometric', 'get_state', 'gumbel', 'hypergeometric', 'info', 'laplace', 'logistic', 'lognormal', 'logseries', 'mtrand', 'multinomial', 'multivariate_normal', 'negative_binomial', 'noncentral_chisquare', 'noncentral_f', 'normal', 'np', 'operator', 'pareto', 'permutation', 'poisson', 'power', 'print_function', 'rand', 'randint', 'randn', 'random', 'random_integers', 'random_sample', 'ranf', 'rayleigh', 'sample', 'seed', 'set_state', 'shuffle', 'standard_cauchy', 'standard_exponential', 'standard_gamma', 'standard_normal', 'standard_t', 'test', 'triangular', 'uniform', 'vonmises', 'wald', 'warnings', 'weibull', 'zipf'

3. Operaciones matemáticas: NumPy

- NumPy (*Numerical computation Python*) es la librería matemática de Python por excelencia, nacida de la unión de varias librerías matemáticas.
- **Principales utilidades:**
 - Operaciones con vectores y matrices (N-arrays)
 - Estadística básica
 - Manejo de fechas (datetime64)
 - Álgebra lineal (numpy.linalg)
 - Generación de números aleatorios (numpy.random)
- **Sitio:** <http://www.numpy.org>
- **Documentación:** <https://numpy.org/devdocs/reference/index.html>
- **Licencia:** open-source BSD-new

Características:

- Trabajar con **vectores y matrices** (N-arrays) es la verdadera funcionalidad de NumPy.
- La estructura de datos son los "ndarray": n-dimensional array

```
1 import numpy as np
2 a = np.array([1, 2, 3, 5])
3 print(a * 3)
4
5 b = np.arange(10)
6 # arange = similar a range, pero devuelve un array, no un
  listado
7 print(b * 3)
```

```
[ 3  6  9 15]
```

```
[ 0  3  6  9 12 15 18 21 24 27]
```

- **np.linspace(inicio, fin, n-pasos):** Devuelve los números espaciados uniformemente durante un intervalo especificado. Por defecto: 50 números.

```
1 import numpy as np
2 c = np.linspace(5, 18, 6)
3 print(c)
```

```
[ 5.  7.6 10.2 12.8 15.4 18. ]
```

- **np.zeros(n)** y **np.ones(n):** Creación de vectores n-dimensionales con ceros o unos:

```
1 print(np.zeros(5))
2 print(np.ones(5))
```

```
[0.  0.  0.  0.  0.]
```

```
[1.  1.  1.  1.  1.]
```

3.1. Indexación

```
1 import numpy as np
2 x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

La sintaxis básica de las secciones es `i:j:k`, donde `i` es el índice inicial, `j` es el índice final y `k` son los `n`-pasos.

- `x[1:7:2]` es `[1 3 5]`
- `x[-2:10]` es `[8 9]` *al igual que* `x[-2:]`
- `x[-3:3:-1]` es `[7 6 5 4]`
- `x[::-1]` es `[9 8 7 6 5 4 3 2 1 0]`

Manejar este concepto es importante para comprender más adelante el funcionamiento de muchos enfoques.

3.2. Operaciones con vectores

Asignar un valor para modificar el valor existente es sencillo. Ejemplo `x[5] = 9`

```
1 import numpy as np
2 x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
3 x[5] = 9
4 print(x)
```

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

`np.append(a,b)`: Anexar items en un array

```
1 import numpy as np
2 x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
3 c = np.append(x, 99)
4 print(c)
```

```
[ 0 1 2 3 4 5 6 7 8 9 99]
```

Concatenación

- `np.concatenate((a,b))`: Concatena los arrays, horizontalmente en columnas.
- `np.vstack((a,b))`: Apila los arrays, verticalmente en filas.

```
1 import numpy as np
2 a = np.array([1, 2, 3])
3 b = np.array([4, 5, 6])
4
5 c = np.concatenate((a, b))
6 print(c)
7
8 d = np.vstack((a, b))
9 print(d)
```

```
[1 2 3 4 5 6]
[[1 2 3]
 [4 5 6]]
```

Ordenación

- **np.sort()**: Ordena los items dentro de un array:

```
1 import numpy as np
2 x = np.array([1, 2.5, 5, 3])
3 print(np.sort(x))
```

```
[1. 2.5 3. 5.]
```

- **np.argsort()**: Muestra las posiciones de los valores una vez ordenados:

```
1 import numpy as np
2 x = np.array([1, 2.5, 5, 3])
3 print(np.argsort(x))
```

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

3.3. Types

Por defecto, NumPy asigna el tipo de dato (*data-type* o *dtype*) como float. Aun así, opcionalmente, puede especificarse el tipo de dato introducido:

np.array = ([1, 2, 3, 5.5], dtype=int) o **np.array = ([1, 2, 3, 5.5], np.int)**

- int: Números enteros (int, int32, int64)
- float: Número con decimales (float, float32, float64)
- str: String o Caracteres
- datetime64: Fechas

```
1 import numpy as np
2 x = np.array([1, 2, 3, 5.5], dtype="int")
3 print(x)
```

```
[1 2 3 5]
```

Adicionalmente, Numpy posee la función **.astype()**, la cual permite convertir el tipo de objeto de cualquier n-array.

```
1 import numpy as np
2 x = np.array([1, 2.5])
3 print(x.astype(int))
```

1 2

3.4. Operaciones con vectores

Con `np.array` se pueden utilizar las funciones incluidas en Python:

```
1 import numpy as np
2 x = np.array([1, 2, 3, 5])
3 print(len(x))
```

4

Así como las funciones que el propio NumPy incorpora:

- `np.sum(x)`: Suma de los valores del vector. En este caso: 11
- `np.prod(x)`: Producto de los valores del vector. En este caso: 30
- `np.cumsum(x)`: Suma acumulada de los valores del vector. En este caso: [1 3 6 11]
- `np.cumprod(x)`: Producto acumulado. En este caso: [1 2 6 30]
- `np.diff(x)`: Diferenciales o valores añadidos en cada salto. En este caso: [1 1 2]

Multiplicación

En el caso de querer multiplicar dos vectores, existe la función `np.vdot(a, b)`:

```
1 import numpy as np
2
3 a = np.array([1, 2, 3, 4, 5])
4 b = np.array([3, 6, 8, 9, 10])
5
6 print(np.sum(a * b))
7 print(np.vdot(a, b))
```

125

125

3.5. Estadísticos

```
1 import numpy as np
2 x = np.array([1, 2, 3, 5])
```

Estadísticos de centralización y dispersión:

- `np.median(x)` es 2.5

- **np.mean(x)** es **2.75**
- **np.std(x)** es **1.479019945774904**
- **np.var(x)** es **2.1875**

En el caso que en el array tenga algún dato NaN y quisieramos hacer, por ejemplo, la media:

- puede sustuirse por cero con la fórmula **np.nan_to_num(array)**.
- O realizar directamente la media despreciando el NaN **np.nanmean(array)**.

```
1 import numpy as np
2 x = np.array([1, 2, np.NaN, 6])
3
4 print(np.nan_to_num(x))
5 print(np.mean(np.nan_to_num(x)))
6 print(np.nanmean(x))
```

[1. 2. 0. 6.]

2.25

3.0

- Otras funciones disponibles son: **np.nanvar()**, **np.nanstd()** y **np.nanmedian()**.

```
1 import numpy as np
2 x = np.array([1, 2, 3, 5])
```

Estadísticos de posición:

- **np.amin(x)** es **1**
- **np.amax(x)** es **5**
- **np.percentile(x, 50)** es **2.5**
- **np.quantile(x, 0.5)** es **2.5**

Las fórmulas incluidas en NumPy Pueden aplicarse de dos formas. Ejemplo:

```
1 import numpy as np
2 x1 = np.array([1, 2, 3, 5]).sum()
3 x2 = sum(np.array([1, 2, 3, 5]))
4 print(x1, x2)
```

11 11

3.6. Exponenciales, logaritmos, constantes

Exponenciales y logaritmos:

- `np.exp(1.25)` es `3.4903429574618414`
- `np.log(1.25)` es `0.22314355131420976`
- `np.log10(1.25)` es `0.09691001300805642`

Principales constantes:

- `np.e` es `2.718281828459045`
- `np.pi` es `3.141592653589793`
- `np.nan` es `NaN` (*Not a Number*)

Otras funciones útiles:

- `np.power(1.25, 2)` es `1.5625`
- `np.sqrt(1.5625)` es `1.25`

3.7. Matrices

Creación de matrices:

```
1 import numpy as np
2 x = np.array([[1, 2, 3], [4, 5, 6]])
3 print(x)
4 print(x.shape)
```

```
[[1 2 3]
 [4 5 6]]
```

```
(2, 3)
```

Seleccionar fila, puede indicarse con corchetes. Ejemplo: `x[0]` devuelve `[1 2 3]`

Y para seleccionar un valor dentro de la matriz, indicando su posición.

Ejemplo: `x[0][0]` o su equivalente `x[0,0]` devuelve `1`.

Selecciones:

```
1 import numpy as np
2 x = np.array([1, 2, 3, 4, 5, 6])
3 print(x[x>2])
```

```
[3 4 5 6]
```

```

1 import numpy as np
2 x = np.array([[1, 2, 3], [4, 5, 6]])
3 x[0,0] = 5
4 print(x)

```

```

[[5 2 3]
 [4 5 6]]

```

```

1 print(x[0:2, 1])

```

```

[2 5]

```

np.eye(n): Crea una matriz identidad. Equivalente a np.identity, aunque .eye, permite desplazar la diagonal k o -k lineas.

```

1 import numpy as np
2 a = np.eye(3) # eye = crea array con 1 en la diagonal ppal
3 print(a)

```

```

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

```

np.full(n): Crea una matriz del tamaño y valores especificados.

```

1 import numpy as np
2 e = np.full((2,2),0.25)
3 print(e)

```

```

[[0.25 0.25]
 [0.25 0.25]]

```

- **np.zeros((n,n)):** Crea una matriz n por n con ceros en su interior. Ejemplo np.zeros((3,3)):

```

[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]

```

- **np.ones((n,n)):** Crea una matriz n por n con unos en su interior. Ejemplo np.ones((3,3)):

```

[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]

```

Axis

Toda matriz posee dos ejes. El `axis=0` corresponde con el eje horizontal (lectura por defecto) y `axis=1` con el vertical.

- `np.sum(x, y)`: Devuelve la suma de los elementos del array sobre el eje definido
- `np.prod(x, y)`: Devuelve el producto de los elementos del array sobre un eje dado.

```
1 import numpy as np
2 a = np.prod([[1.,2.],[3.,4.]], axis=0)
3 print(a)
4
5 b = np.sum([[1.,2.],[3.,4.]], axis=1)
6 print(b)
```

```
[3. 8.]
```

```
[3. 7.]
```

Del mismo modo, en el caso de `cumsum()` y `cumprod()`, si queremos acumular la suma o multiplicación de izquierda a derecha debemos especificar `axis=1`

Multiplicación de matrices.

`np.dot(x, y)`: Multiplica dos matrices, donde la matriz identidad y debe tener el mismo orden que `x`, es decir, que el número de columnas de `x` debe coincidir con el número de filas de `y`.

```
1 import numpy as np
2 a = np.array([[1, 2, 3], [4, 5, 6]])
3 b = np.array([[7, 8], [9, 6], [5, 1]])
4
5 print(a.shape, b.shape)
6 print(np.dot(a, b))
```

(previamente comprobamos que el número de filas de *A* sea igual que el número de columnas de *B*)

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

```
[[ 40 23]
 [103 68]]
```

Otras operaciones con matrices

- `.transpose()` o `.T`: Devuelve la matriz transpuesta.
- `.inverse()` `.I`: Devuelve la matriz inversa
- `.diagonal()`: Devuelve la diagonal principal.

Reordenación de elementos:

- `flip(m[, axis])`: Invierte el orden de los elementos de un array según el eje dado.
- `flipr(m)`: Voltea la matriz sobre el eje vertical (izquierda/derecha). Ejemplo `np.flipr(A)`
- `flipud(m)`: Voltea la matriz sobre el eje horizontal (arriba/abajo).

3.8. Ejercicio 2

Ejercicio_02_01

Dados los siguientes valores de los diferentes submódulos de SCR:

Mercado	Crédito	Vida	Salud	No Vida
95 000	65 000	25 000	35 000	125 000

Se pide:

- Calcular el BSCR aplicando la matriz de correlaciones del Reglamento Delegado de Solvencia II:

1	0.25	0.25	0.25	0.25
0.25	1	0.25	0.25	0.5
0.25	0.25	1	0.25	0
0.25	0.25	0.25	1	0
0.25	0.5	0	0	1

- Calcular el efecto de diversificación.

3.9. NumPy Financial

NumPy Financial es el reemplazo de las funciones financieras originales de NumPy. Al ser el *spin off* de NumPy, hereda sus funciones financieras elementales con objeto implementar nuevas funcionalidades a futuro.

Las funciones financieras en NumPy (npv, irr, etc..) serán eliminadas en la próxima versión 1.20 de NumPy.

- **Funciones:** (versión 1.0.0):
 - Descuento de flujos: fv, pv, npv
 - Tipo de interés: irr, mirr, rate
 - Cálculo de pagos y periodos de pago: pmt, ppmt, ipmt, nper
- **Sitio:** <https://numpy.org/numpy-financial/>
- **Documentación:** <https://numpy.org/doc/1.17/reference/routines.financial.html>
- **Licencia:** open-source BSD-new

Internal Rate of Return

- **npf.irr(values):** Devuelve la TIR o IRR (Internal Rate of Return):

```
1 import numpy_financial as npf
2 payments = [-100, 39, 59, 55, 20]
3 tir = npf.irr(payments)
4 print(tir)
```

```
0.2809484211599611
```

Descuentos de flujos

El descuento de los flujos es la tarea actuarial básica para calcular los valores actuales de pagos futuros.

- **npf.npv:** Calcula el valor actual neto de una serie de pagos utilizando una única tasa de descuento. Es el equivalente al NPV en MS Excel.

```
1 import numpy_financial as npf
2 payments = [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100]
3 net_present_value = npf.npv(0.02, payments)
4 print(net_present_value)
```

```
916.2236706367081
```

- **npf.pv(rate, n^o period, payment, when='end')** Calcula el valor actual de una serie de pagos utilizando una única tasa de descuento. Es el equivalente al PV en MS Excel. *When* indica cuando se realizan los pagos: ('begin' (1) or 'end' (0)).

```
1 import numpy_financial as npf
2 present_value = npf.pv(0.02, 10, 100, when=1)
3 print(present_value)
```

```
-916.22367063670872
```

- **np.vdot:** Multiplica los componentes de dos conjuntos y devuelve la suma de los productos. El conjunto debe tener las mismas dimensiones. Es el equivalente a SUMAPRODUCTO en MS Excel.

```
1 import numpy as np
2 payments = [100, 100, 100, 100, 100, 100, 100, 100, 100, 100]
3 discount = [1, 0.980392, 0.961168, 0.942322, 0.923845, 0.90573,
4             0.887971, 0.87056, 0.85349, 0.836755]
5 sumproduct = np.vdot(payments, discount)
6 print(sumproduct)
```

```
916.2233
```

3.10. Ejercicio 3

Ejercicio_02_02

Supongamos que en una cartera a prima única se espera una siniestralidad total acumulada del 50 % de la prima repartida linealmente durante los 7 años de vigencia.

Se pide calcular la siniestralidad estimada que tendría la cartera valorada en $t=0$ teniendo en cuenta una prima de 1.000, una hipótesis del 2 % anual de inflación en los importes a pagar y una tasa de descuento del 0.5 % anual

Para simplificar, puede utilizarse $npv(tasa, pagos)$ para descontar, aunque no descuenta el primer año.

3.11. Simulaciones

Las **simulaciones** requieren el uso de muestras aleatorias para encontrar una solución a un problema matemático.

Existen varias técnicas de muestreo estadístico (**Monte Carlo, bootstrap...**).

Dado que la simulación implica un muestreo aleatorio a partir de un conjunto de posibilidades inciertas, el proceso de simulación requiere:

- la especificación del conjunto de posibilidades inciertas,
- las estimaciones de probabilidad asignadas relacionadas con el conjunto de posibilidades
- un generador de números aleatorios.

3.12. Números aleatorios

Numpy posee el módulo `numpy.random` que implementa **generadores de números pseudo-aleatorios** para varias distribuciones.

- Ofrece números aleatorios de distribuciones discretas y continuas, así como algunas utilidades relacionadas con los números aleatorios.
- Usa PCG64 como generador interno¹ (uno de los nuevos generadores más avanzados que existen). Esta basado en C que utiliza los ciclos del reloj del procesador.

Principales funciones de **numpy.random**:

- **np.random.rand(x,y)**: Devuelve valores aleatorios en una forma dada, entre 0 y 1.
- **np.random.randn(x,y)**: Devuelve una muestra (o muestras) de la distribución normal estándar (Gaussiana), entre 0 y 1.
- **np.random.randint(low[, high, size])** números enteros aleatorios desde una base (inclusive) a un tope (exclusivo).
- **np.random.random(size)** Devuelve numeros *float* aleatorios en el intervalo medioabierto [0.0, 1.0)
- **np.random.choice(a[, size, replace])** Genera una muestra aleatoria de un array 1-D dado.
 - a: 1-D array
 - size: tamaño de la muestra
 - replace: True (con reemplazamiento), False (sin reemplazamiento)

¹<https://www.pcg-random.org/index.html>

Generador de números aleatorios acorde a una función de distribución dada

- **np.random.Distributions** : Numpy posee 35 distribuciones de probabilidad: `.normal()`, `.lognormal()`, `.poisson()`, `.gamma()`, `.f()`, `.binomial()`, etc.
Ejemplo para una distribución normal:

```
1 import numpy as np
2 mu, sigma = 0, 0.1 # mean and standard deviation
3 s = np.random.normal(loc=mu, scale=sigma, size=1000)
```

- Cada distribución requiere de sus **propios parámetros**. Ejemplo para una distr. gamma:

```
1 import numpy as np
2 shape, scale = 2., 2. # mean=4, std=2*sqrt(2)
3 s = np.random.gamma(shape=shape, scale=scale, size=1000)
```

- Documentación: <https://docs.scipy.org/doc/numpy/reference/random/generator.html#distributions>

Otras funcionalidades útiles de **numpy.random**:

- **np.random.seed(n)** siendo n la semilla del generador. Si se utiliza la misma semilla generará siempre la misma secuencia de valores.
- **np.random.RandomState()** Paquete para el generador de números pseudo-aleatorios Mersenne Twister (MT19937). Además de ser similar que NumPy, aporta mayor consistencia en la generación aleatoria y proporciona 48 distribuciones de probabilidad para elegir
Ejemplo: **np.random.RandomState.normal()**
- **np.random.choice(m, n)**: rango m-valores, n-items:

```
1 import numpy as np
2 np.random.seed(999)
3 x = np.random.choice(5, 3) #Equivale a np.random.randint
   (0,5,3)
4 print(x)
```

```
[0 4 1]
```

- Especificando el rango de posibles valores:

```
1 import numpy as np
2 np.random.seed(999)
3 a = [1, 3, 5, 7, 9]
4 x = np.random.choice(a, 3)
5 print(x)
```

```
[1 9 3]
```

3.13. Álgebra lineal con Numpy

Otro módulo útil de Numpy es **numpy.linalg** enfocado al álgebra lineal, el cual nos permite resolver ecuaciones de una manera fácil y rápida:

- **linalg.solve(a, b)**: Resuelve una ecuación o sistema de ecuaciones lineales.

```
1 import numpy as np
2 a = np.array([[3, 1], [1, 2]])
3 b = np.array([9, 8])
4 x = np.linalg.solve(a, b)
5 print(x)                # [2. 3.]
6 print(np.dot(a, x))     # Chequeo: a * x = b
```

```
[2. 3.]
[9. 8.]
```

- **linalg.tensorsolve(a, b[, axes])**: Cálculo tensorial de ecuaciones multimatriz: $ax = b$ para x .
- **linalg.lstsq(a, b[, rcond])**: Devuelve la solución de mínimos cuadrados a una ecuación matricial lineal.

El módulo `numpy.linalg` también posee funciones útiles para transformaciones y descomposiciones de matrices.

- **np.linalg.cholesky(a)** Descomposición de Cholesky (halla la matriz triangular inferior para matrices positivas simétricas).
- **np.linalg.qr(a[, mode])** Calcula la factorización QR de una matrix, usando la transformación de Householder .

4. Visualización de datos: Matplotlib

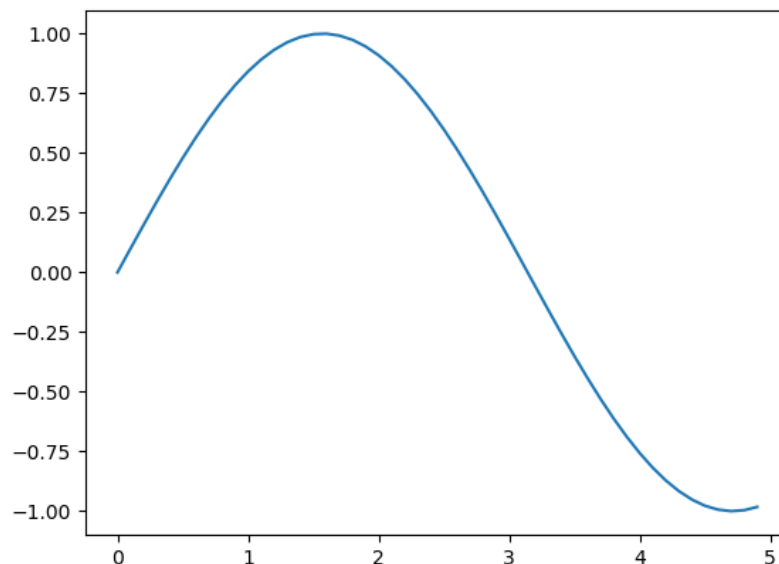
El módulo de **.pyplot** dentro de matplotlib nos ofrece una interface similar a MATLAB para la generación de gráficos.

- Los datos pueden ser tanto vectores como matrices 2D
- Los ejes deben tener la misma cantidad de elementos
- Para gráficos 3D se recomienda el toolkit **mplot3d**

Con la instrucción **pyplot.plot(x, y)** se crea la gráfica de x frente a y. Ejemplo:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0, 5, 0.1)
5 y = np.sin(x)
6 plt.plot(x, y)           # Crea la grafica
7 plt.show()               # Muestra la grafica
```

Gráfica creada con 1 sola línea de código: **matplotlib.pyplot.plot(x, y)**



- Por defecto se usa una línea continua para unir los puntos, a no ser que se indique otra cosa.
- La gráfica puede mostrarse en pantalla con **.show()** y/o guardarse en formato PNG con **.savefig('fichero.png', transparent=True)**.

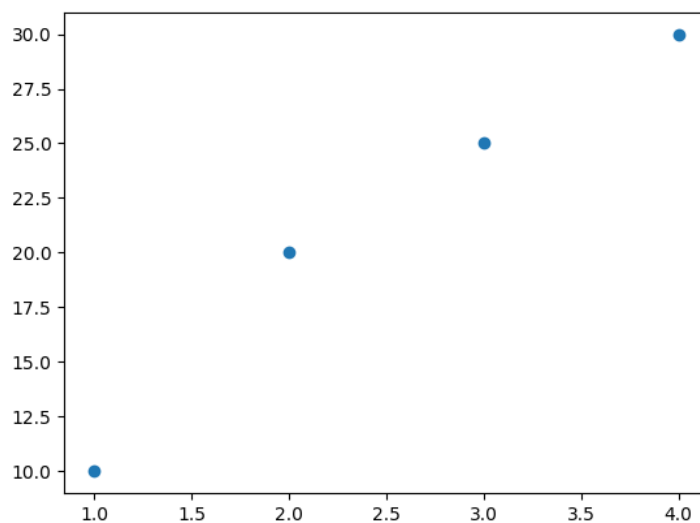
4.1. Tipos de gráficas

- **.plot**: line plot (*por defecto*)
- **.bar**: vertical bar plot
- **.barh**: horizontal bar plot
- **.hist**: histogram
- **.box**: boxplot
- **.kde**: Kernel Density Estimation
- **.density**: same as 'kde'
- **.area**: area plot
- **.pie**: pie plot
- **.scatter**: scatter plot
- **.hexbin**: hexbin plot

```
1 import matplotlib.pyplot as plt
2 plt.plot(x, y)
3 plt.show()
```

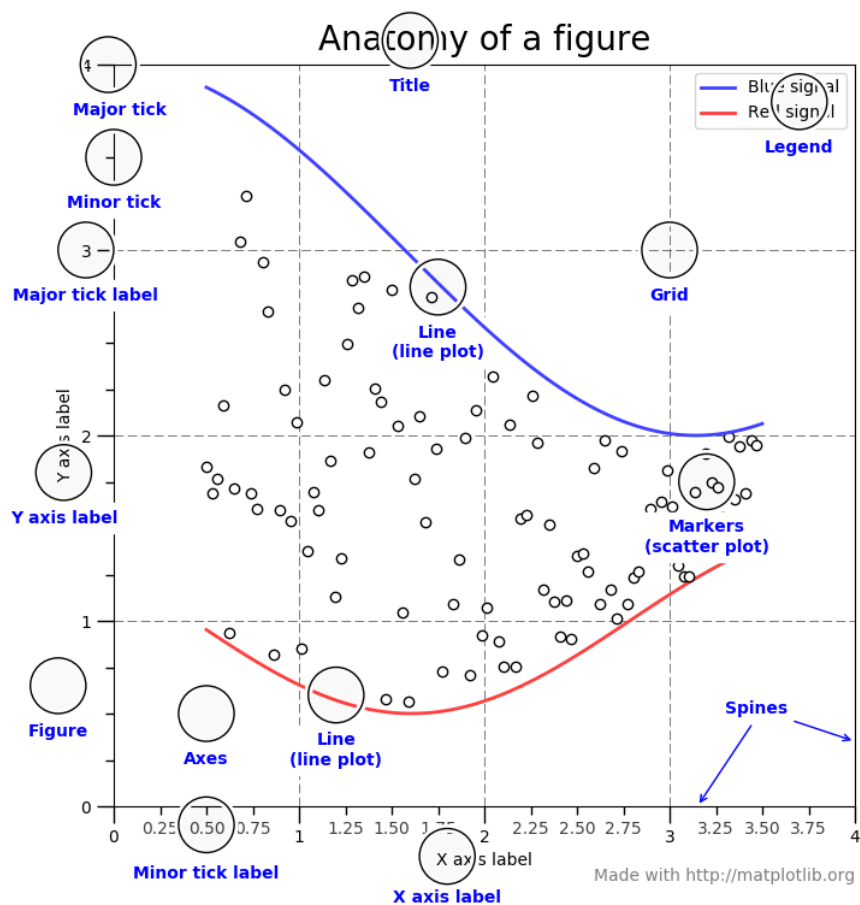
Este mismo gráfico puede realizarse con `plot.bar(x,y)`, `plot.pie(x)`, etc.

```
1 import matplotlib.pyplot as plt
2 x = [1, 2, 3, 4]
3 y = [10, 20, 25, 30]
4 plt.scatter(x, y)
5 plt.show()
```



A pesar de su aparente sencillez, la cantidad de parámetros (en este caso funciones) que soporta **pyplot** es muy numerosa, pudiéndose crear cualquier gráfica deseada. Las funcionalidades más usadas son:

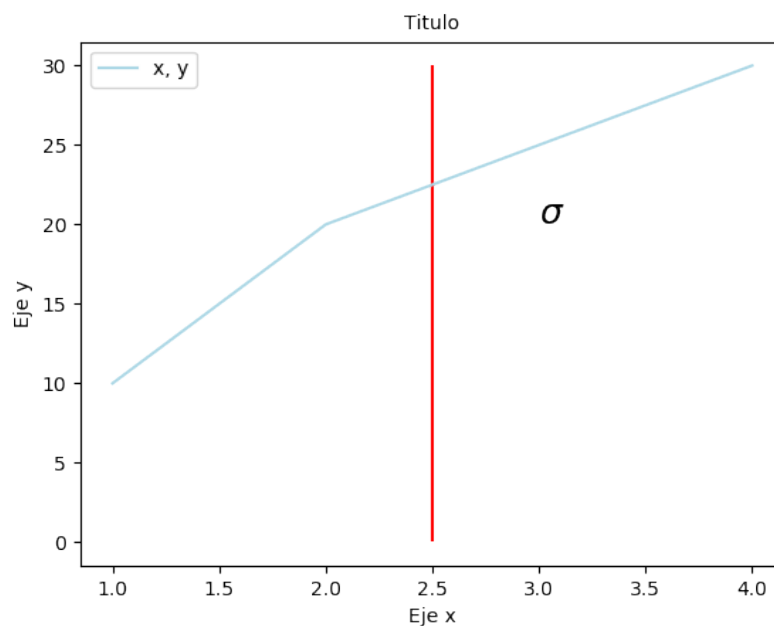
- **plt.xlabel()**: nombramos el eje x
- **plt.ylabel()**: nombramos el eje y
- **plt.title()**: título de la gráfica
- **plt.hlines(y, xmin, xmax[, colors, linestyles...])**: Dibuja una línea horizontal en el valor y
- **plt.vlines(x, ymin, ymax[, colors, linestyles, ...])**: Dibuja líneas verticales
- **plt.legend(loc='upper left')**: Muestra leyenda. En este caso superior izquierda. Por defecto loc='best'.
- **plt.text()**: añade el texto en la posición que indiquemos (admite formato \LaTeX)
- **Otros**: plt.grid(True), .table(), .bar(), .box(), .pie(), .subplot(), .xkcd()



Ejemplo de creación de gráfica

```
1 import matplotlib.pyplot as plt
2 x = [1,2,3,4]
3 y = [10,20,25,30]
4
5 plt.plot(x, y, label='x, y',color='lightblue')
6 plt.xlabel('Eje x', fontsize=10)
7 plt.ylabel('Eje y', fontsize=10)
8 plt.title('Titulo', fontsize=10)
9 plt.vlines(2.5, 0, 30, color='red')
10 plt.legend(loc='best')
11 plt.text(3, 20, '$\sigma$', fontsize=18)
12 plt.savefig('fichero.png', transparent=True)
13 plt.show()
```

Resultado de la gráfica



4.2. Histogramas

matplotlib.pyplot.hist, Dibuja un histograma. Parámetros:

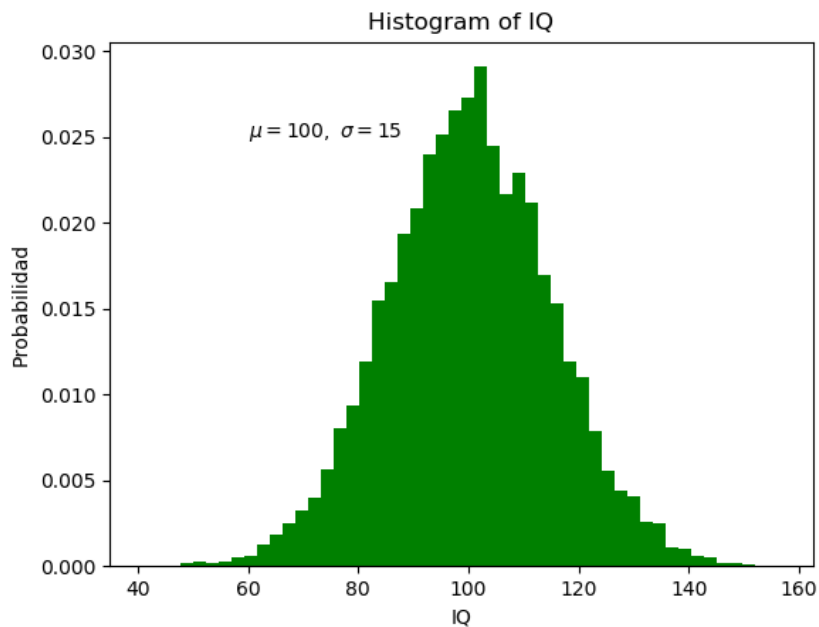
- **x**: array o n-array
- **bins**: *int* o rango (*opcional*)
- **range**: Rango de datos a graficar. Es opcional y por defecto: (x.min(), x.max())
- **density**: (*True* o *False*), Si es *True*, el primer elemento serán los datos normalizados para formar una densidad de probabilidad, es decir, el área bajo el histograma sumará 1. (*opcional*), por defecto es *False*.

- **cumulative:** (*True o False*), Si es True, entonces calcula un histograma acumulado, donde la última ubicación muestra el número total de puntos, (si es normalizado será 1). (*opcional*), por defecto es False.
- **histtype:** Tipo de histograma. Los valores posibles son 'bar' (*por defecto*), 'bars-tacked', 'step', 'stepfilled'.
- **Otros:** align ('left', 'mid', 'right') , orientation ('horizontal', 'vertical'), label, color ('b', 'g', 'r', 'c', 'm', 'y', 'k', 'w', '#hex')

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  mu, sigma = 100, 15
5  x = mu + sigma * np.random.randn(10000)
6  plt.hist(x, bins=50, density=True, color='g')
7  plt.xlabel('IQ')
8  plt.ylabel('Probabilidad')
9  plt.title('Histogram of IQ')
10 plt.text(60, .025, "$\mu=100,\sigma=15$")
11 plt.show()

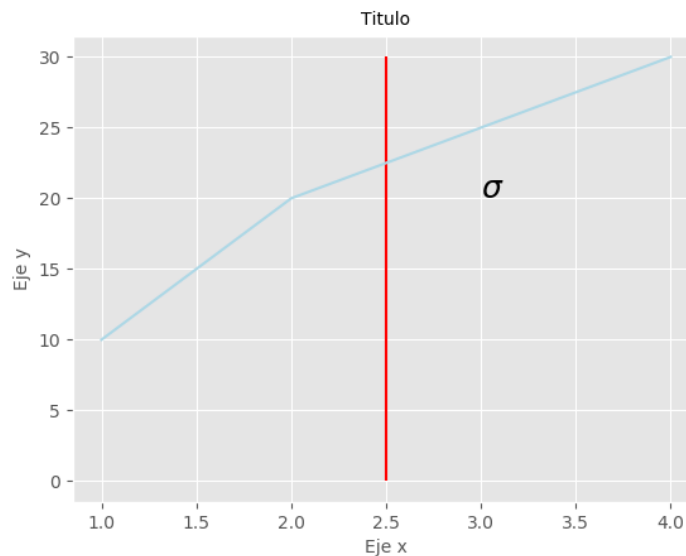
```



4.3. Estilos

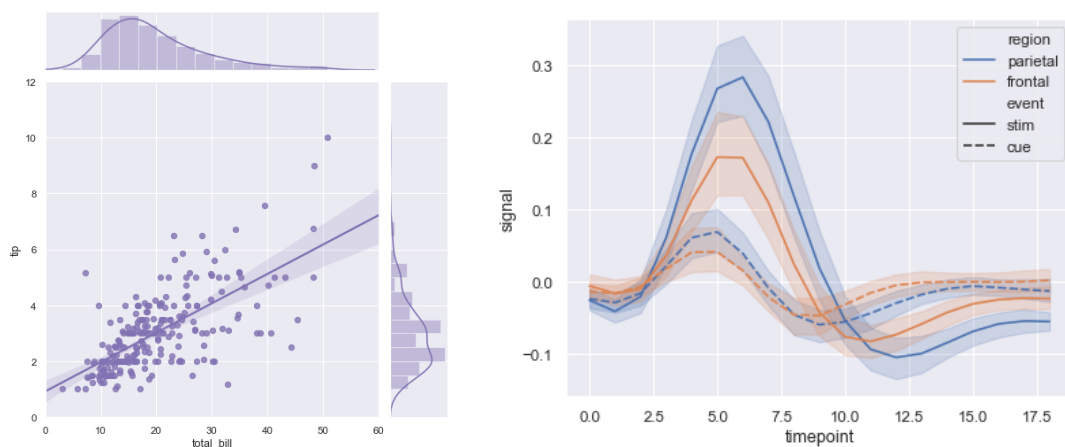
Matplotlib incorpora una amplia **hoja de estilos** entre los que destacamos:

- `plt.style.use('seaborn')`
- `plt.style.use('ggplot')`, basado en el paquete ggplot2 de R:



4.4. Seaborn

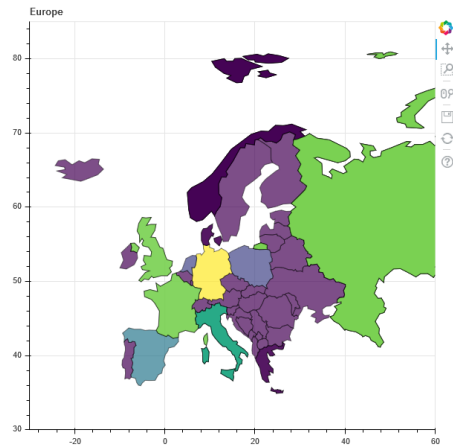
Seaborn es otra librería basada en matplotlib con una gran colección de recursos para gráficos estadísticos, principalmente funciones de distribución, y regresiones lineales (por ejemplo colorea los posibles valores dentro de un intervalo de confianza).



Site: <http://seaborn.pydata.org>

4.5. Bokeh

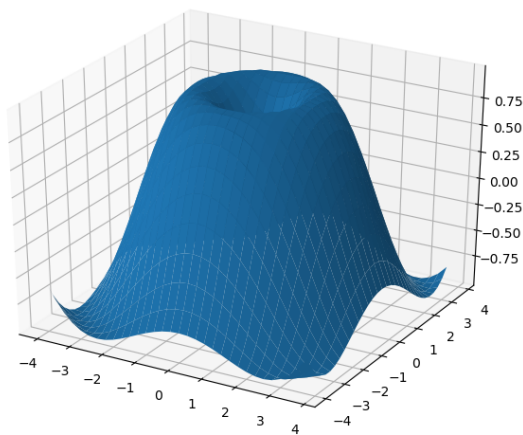
Bokeh es una librería para visualizaciones interactivas para navegadores (necesita fichero javascript D3.js). Site: <https://bokeh.pydata.org>



Gráficos 3D

Con `.plot_surface()` y el toolkit de mplot3d (`mpl_toolkits.mplot3d.Axes3D`) se pueden realizar gráficos en 3D:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from mpl_toolkits.mplot3d import Axes3D
4 ax = Axes3D(plt.figure())
5 X = np.arange(-4, 4, 0.25)
6 Y = np.arange(-4, 4, 0.25)
7 X, Y = np.meshgrid(X, Y)
8 R = np.sqrt(X**2 + Y**2)
9 Z = np.sin(R)
10 ax.plot_surface(X, Y, Z)
11 plt.show()
```



4.6. Ejercicio 4

Ejercicio_03_01

Con los datos del ejercicio 02_02, se pide crear un diagrama de barras con cada uno de los importes de los submodulos del BSCR, así como el importe del efecto de diversificación, utilizando para ello **numpy** y **matplotlib**.



Material adicional

Versión utilizada de Matplotlib: 3.1.3

5. Análisis de datos: Pandas

Pandas (*panel data*) es la extensión de Numpy que ofrece trabajar con estructuras de datos y operar con tablas numéricas y series temporales.

- **Sitio:** <https://pandas.pydata.org>
- **Documentación:** <https://pandas.pydata.org/pandas-docs/stable/>
- **Licencia:** open-source BSD 3-Clause License
- El motor de Pandas es **NumPy**, por tanto, las operaciones matriciales son similares e intuitivas.
- En pandas existen dos tipos básicos de objetos, que están basados a su vez en Numpy:
 - Series: vectores 1-D
 - DataFrame: matrices 2-D

Operaciones típicas que pueden realizarse fácilmente con **Pandas**:

- Crear un DataFrame a partir de una consulta SQL o SAS, una lista, un archivo externo (csv o xls entre otros), o incluso una página web.
- Filtrar las filas o columnas de interés
- Limpiar o borrar determinados valores
- Calcular nuevas columnas basada en columnas existentes
- Resumir con una simple función los principales estadísticos de un conjunto de datos
- Graficar una columna contra otra
- Modelar matemáticamente una columna en función de otra, por ejemplo, utilizando regresión lineal.

5.1. Series

pd.Series(data). Atributos:

- **data:** pueden ser datos almacenados en arrays o constantes
- **index:** Por defecto será (0, 1, 2, ..., n)
- **dtype:** Si no existe, pandas deducirá el tipo.

```
1 import pandas as pd
2 s = pd.Series([1, 2, 3, 4])
```


5.2. DataFrame

¿Qué es un **DataFrame**?

- Es una estructura de datos en formato tabla, similar al dataframe disponible en R.
- El formato utilizado en los proyectos de análisis de datos, a pesar de restar velocidad de cálculo (aunque cada nueva versión de Pandas mejora este aspecto).
- Un DataFrame se diferencia de una tabla SQL en que se manipulan directamente en la memoria de su programa ya que no residen en un servidor remoto.
- Son más adecuados para el análisis “offline” complejos con estadísticas, visualización y modelos matemáticos.
- El límite tamaño de un df lo determina la cantidad memoria RAM. Existen ejemplos de implementaciones con 100 millones de filas.

pd.DataFrame(data). Atributos:

- **data**: pueden ser Series, arrays, constants
- **index**: Por defecto será (0, 1, 2, ..., n)
- **columns**: Nombre de las columnas. Por defecto será (0, 1, 2, ..., n)
- **dtype**: Si no existe, pandas deducirá el tipo. Puede cambiarse posteriormente.

```
1 | df = pd.DataFrame([1, 2, 3, 4], columns=['Importe'])
```

Si el nombre de la columna no contiene espacios, puedo hacer mención a las columnas ya existentes de dos formas:

`df.Importe` *equivale a* `df['Importe']`

NumPy incluido

- Pandas posee como dependencias, entre otras, las librerías **NumPy** y **dateutil**.
- Por tanto, la sintaxis que hemos visto anteriormente en NumPy se aplica exactamente igual en Pandas, pudiendo acceder a las mismas funciones.
- Aun así, es recomendable importar NumPy para operar totalmente con Pandas ya que ambas son perfectamente compatibles.

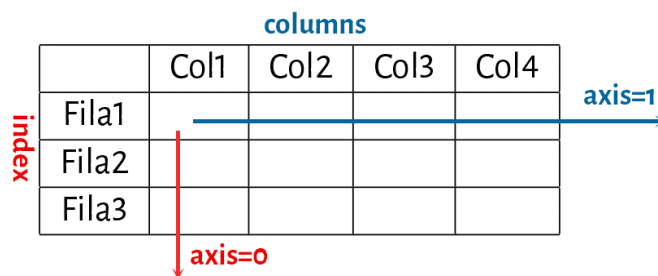
```
1 import pandas as pd
2 df = pd.DataFrame([[1, 2, 3], [4, 5, 6]])
3 print(df)
4 print(df.mean())
```

	0	1	2
0	1	2	3
1	4	5	6

0	2.5
1	3.5
2	4.5

Pandas posee las mismas funciones que **Numpy** y sus llamadas son intuitivas. Los **DataFrame** se componen de filas y columnas. Para cualquier cálculo (por ejemplo la suma), conviene tener en cuenta si las funciones se aplican a la columna (*por defecto*) o a la fila.

- Cada eje (axis) de un **DataFrame** tiene un índice (index). Por defecto: 0, 1, 2 .. n.
- **axis=0**: Lectura en horizontal. **axis=0** puede cambiarse por **axis='index'**
- **axis=1**: Lectura en vertical. **axis=1** equivale a **axis='columns'**



Partimos de este DataFrame:

	0	1	2
0	1	2	3
1	4	5	6

```
1 import pandas as pd #solo importo pandas
2 df = pd.DataFrame([[1, 2, 3], [4, 5, 6]])
3 print(df.mean())
```

0	2.5
1	3.5
2	4.5

```
1 print(df.mean(axis=1)) # o axis='columns'
```

0	2.0
1	5.0

- Tanto pandas como numpy admiten argumentos en las fórmulas:

```
1 import pandas as pd
2 import numpy as np
3 df = pd.DataFrame([[1, 2, 3, 4, 5, 6]])
4 print(df.mean(axis=1))
5 print(np.mean(df, axis=1))
```

0	3.5
0	3.5

- Aunque pandas admite determinados argumentos que numpy todavía no, como **skipna**.

```
1 import pandas as pd
2 import numpy as np
3 df = pd.DataFrame([[1, 2, 3, np.NaN, 5, 6]])
4 print(df.median(axis=1, skipna=True))
5 print(np.median(df, axis=1)) # Numpy usa nanmedian()
```

0	3.0
---	-----

Invalid value encountered in median for 1 results	
---	--

Modificación de Series/DataFrame

Se puede crear un DataFrame partiendo de una series con `.to_frame()`:

```
1 import pandas as pd
2 s = pd.Series([1, 2, 3, 4])
3 df = s.to_frame()
```

Así como crear un DataFrame vacío o con una sola columna, y posteriormente añadir nuevas.

```
1 import pandas as pd
2 import numpy as np
3 df = pd.DataFrame([1, 2, 3, 4], columns=['Col1'])
4 df['Col2'] = df['Col1'] ** 2
5 print(df)
```

	Col1	Col2
0	1	1
1	2	4
2	3	9
3	4	16

5.3. Importación desde texto plano (csv, txt, tab...)

Creación de DataFrame partiendo de un fichero csv.

```
1 import pandas as pd
2 datos = pd.read_csv('fichero.csv', sep=';')
```

`pd.read_csv(fichero)`. Principales parámetros:

- **sep**: Separador en formato *str*. Ejemplo: ';' o '\t' (tabulado). Por defecto: ','
- **decimal**: Separador de decimales. Ejemplo: ',' . Por defecto '.'
- **header**: *int*, número de fila donde figura los nombres de las columnas.
- **names**: *str*, Listado con los nombres a usar como cabecera del DataFrame.
- **index_col**: *int*, columna a usar como Index (nombre de las filas) del DataFrame. `index_col=False` fuerza a pandas a no usar la primera columna como index.
- Otros ²: `skiprows`, `nrows`, `skipfooter`, `skip_blank_lines`, `parse_dates`, `compression` (lee zip), `memory_map`

²https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

5.3.1. Importación desde Excel

Creación de DataFrame partiendo de un fichero MS Excel.

```
1 import pandas as pd
2 datos = pd.read_excel('datos.xls')
```

`pd.read_excel(fichero)`. Principales parámetros:

- **sheet_name**: *string, int o list*
- **header**: *int*, número de fila con los nombres de las columnas.
- **skiprows**: *int*, indica las primeras filas que deben ignorarse. (Si no hay cabecera)
- **usecols**: *int o list*, indica la columna/s a importar. Ej: 'B:C' o range(1,3)
- **skipfooter**: *int*, indica las filas del final que deben ignorarse. Por defecto: None
- **index_col**: columna a usar como Index del DataFrame.
- **names**: *str*, Listado con los nombres a usar como cabecera del DataFrame.

5.3.2. Importación desde url

Creación de DataFrame partiendo de información publicada en internet. Para ello, Pandas hace uso de las siguientes librerías de *Web scraping*: BeautifulSoup4, htmllib5 o lxml.

`pd.read_html(url, flavor)`. Principales argumentos:

- **url**: dirección url
- **flavor**: bs4, htmllib5 o lxml (por defecto)
- otros: skiprows, header, index_col...

```
1 import pandas as pd
2 web = 'http://www.bolsamadrid.es/esp/aspx/Mercados/Precios.aspx?
      indice=ESI100000000'
3 df = pd.read_html(web, flavor=['html5lib'])
4 print(df)
```

5.3.3. Exportar a CSV o Excel

Pandas, con la misma facilidad que puede leer datos de un fichero csv o Excel, puede exportar cualquier dataframe a dichos formatos con la instrucción **df.to_csv()** o **df.to_excel()**

- Exportación de un df a un fichero csv:

```
1 import pandas as pd
2 import numpy as np
3
4 df = pd.DataFrame(np.zeros(5))
5 df.to_csv('df_empty.csv', sep=';', decimal=',')
```

- Exportación de un df a fichero Excel:

```
1 import pandas as pd
2 import numpy as np
3
4 df = pd.DataFrame(np.zeros(5))
5 df.to_excel('Fichero.xlsx')
```

Consultar la función `pd.ExcelWriter` para trabajar con pestañas dentro del Excel.

5.4. Ejercicio 5

Ejercicio_04_01

Importar en un dataframe la curva de tipos de interés sin riesgo publicada por EIOPA, sin ajuste de volatilidad, desde el fichero **EIOPA_RFR_20191231_Term_Structures.xlsx**

5.5. Vista y verificación de datos

Una buena costumbre antes de empezar a analizar, es chequear la consistencia de los datos:

- **df.head(n)**: Muestra las n primeras filas del DataFrame/Serie.
- **df.tail(n)**: Muestra las n últimas filas del DataFrame/Serie.
- **df.describe()**: Muestra las principales métricas (conteo, medias, percentiles)
- **df.info()**: Muestra información del df, como el tipo de dato y la memoria utilizada.

Top-level missing data:

- **df.isnull().any()**: `.isnull()` muestra si hay valores nulos celda a celda, y `.any()` resume el chequeo por columna.
- **df.isna().any()**: detecta valores vacíos.
- **df.notna().any()**: `.notna()` muestra si hay valores NaN celda a celda.
- **Series.nunique()**: Indica por columna el número de valores únicos.
- **Series.value_counts()**: Cuenta valores (identifica duplicados) en columnas o Series.

5.6. Limpieza de datos

Las siguientes instrucciones pueden ser útiles en caso de ser necesario limpiar nuestros datos de valores vacíos, nulos o duplicados:

- **df.fillna(value)**: Rellena los campos NaN con el valor indicado (por ejemplo con 0 o con el valor indicado)

```
1 df['subp'].fillna(0)
2 df['sexo'].fillna('M')
3 df['edad'].fillna(value=df['edad'].mean())
```

- **df.dropna(inplace=True)**: Borra aquellas filas con valores NaN. Puede indicarse `index=1` para borrar columnas con datos ausentes.
- **df.drop_duplicates(keep='last')**: Borra duplicados, `'first'` borra todos excepto el primero (*por defecto*), `'last'` borra todos excepto el último, `'None'` borra todos los duplicados.
- **df.index.duplicated()**: Chequea valores index duplicados.

5.7. Fechas

Pandas infiere el formato de las columnas importadas. Aun así, a veces es necesario especificar el formato deseado, principalmente cuando manejamos fechas, ya que puede entender un formato erróneo. Así nos aseguramos que Pandas entienda perfectamente la fecha.

Para ello, existe la función **Series.to_datetime(date, args)** con los siguientes argumentos:

- **date**: en formato *int*, *float*, *str*, *datetime* o *list*.
(en algún caso, puede resultar útil forzarlo a ser *str* con un *.astype(str)* y aplicarle el formato correcto)
- **format**: Formato del dato origen strftime, ej “%d %m %Y”.
Los formatos posibles (%d, %m, %Y, %y, %j...) pueden consultarse en: <https://docs.python.org/2/library/time.html>
- **errors**: En caso de error, que debe hacer **pandas** =‘ignore’ (ignorar) o =‘coerce’ (forzar).

```
1 data['Accident_Date'] =  
2     pd.to_datetime(data['Accident_Date'].astype(str), format=  
    '%Y%m')
```

5.8. Series temporales

Una utilidad disponible en Pandas es crear directamente rangos de fechas: **Series.dt.date_range()** con los siguientes parametros:

- **start**: Inicio del rango. Límite izquierdo para generar fechas.
- **end**: Fin del rango. Límite derecho para generar fechas.
- **period**: Número de periodos a generar
- **freq**: Frecuencia de las proyecciones.
 - D: Diaria (*por defecto*)
 - Y: Anual
 - M: Mensual
- **closed**: Si queremos excluir el inicio (closed='right') o el final (closed='left')
- **name**: Nombre del DatetimeIndex resultante (*por defecto ninguno*)

Ejemplo:

```
1 import pandas as pd
2 s = pd.date_range(start='31/12/2019', periods=8, freq='Y')
3 df = pd.DataFrame(s)
4 print(df)
```

```
0
0 2019-12-31
1 2020-12-31
2 2021-12-31
3 2022-12-31
4 2023-12-31
5 2024-12-31
6 2025-12-31
7 2026-12-31
```

Por último, una funcionalidad muy útil para agrupar por fechas es convertir/agrupar la fecha en periodos (semanas, meses, trimestres, años). Para ello se utiliza la instrucción **Series.dt.to_period(freq=*str*)**, siendo los principales valores:

- **W**: weekly
- **M**: month end frequency
- **SM**: semi-month end frequency (15th and end of month)
- **SMS**: semi-month start frequency (1st and 15th)
- **Q**: quarter end frequency
- **Y**: year end frequency
- Lista completa: <https://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases>

```
1 import pandas as pd
2 s = pd.date_range('2019-01-01', freq='M', periods=12)
3 p = s.to_period('Q')
4 print(p)
```

```
['2019Q1', '2019Q1', '2019Q1', '2019Q2', '2019Q2', '2019Q2',
 '2019Q3', '2019Q3', '2019Q3', '2019Q4', '2019Q4', '2019Q4']
```

5.9. Ejercicio 6

Ejercicio_04_02

Los datos de la siniestralidad (2013-2018) de una determinada cartera de pólizas se encuentra en el fichero: **Siniestros_2013_2018.csv**. Se pide:

- Importar en un dataframe el fichero .csv y dar formato correcto a las columnas de fecha.
- Convertir las columnas de **Fec_ocu** y **Fec_pago** en periodicidad mensual.

5.10. Operaciones dentro de un dataframe

5.10.1. Añadir columnas

Cuando la columna que queramos añadir sea un cálculo sencillo, o incluso asignarle un valor fijo, simplemente indicamos la variable y la operacion a realizar. Por ejemplo:

```
1 import pandas as pd
2 import numpy as np
3 df = pd.DataFrame([1, 2, 3, 4], columns=['Col1'])
4 df['Col2'] = 100
5 df['Col3'] = np.exp(df['Col1'])
6 print(df)
```

	Col1	Col2	Col3
0	1	100	2.72
1	2	100	4.39
2	3	100	20.09
3	4	100	54.59

Otro ejemplo:

```
1 import pandas as pd
2 import numpy as np
3 df = pd.DataFrame([1, 2, 3, 4], columns=['Col1'])
4 df['Col2'] = np.ones(4, dtype=int)
5 df['Col3'] = df['Col1'] + df['Col2']
6 df['Col4'] = np.linspace(1, 10, 4).astype(int)
7 df['Col5'] = 5
8 print(df)
```

	Col1	Col2	Col3	Col4	Col5
0	1	1	2	1	5
1	2	1	3	4	5
2	3	1	4	7	5
3	4	1	5	10	5

5.10.2. Borrar filas o columnas

- Creamos un DataFrame (3x3) con datos aleatorios:

```
1 data = np.random.rand(3, 3)
2 df = pd.DataFrame(data, columns=['Col1', 'Col2', 'Col3'])
3 print(df)
```

	Col1	Col2	Col3
0	0.738582	0.402065	0.853371
1	0.650983	0.068258	0.738703
2	0.225089	0.598409	0.616579

- Borramos una columna:

```
1 df.drop('Col2',axis=1, inplace=True)
2 print(df)
```

	Col1	Col3
0	0.738582	0.853371
1	0.650983	0.738703
2	0.225089	0.616579

- Borramos filas siguiendo una regla:

```
1 df.drop(df[df.Col1>=0.5].index, inplace=True)
2 print(df)
```

	Col1	Col3
2	0.225089	0.616579

5.10.3. Función lambda

- Se conoce a **lambda** como la función anónima (no es exclusiva de pandas).

```
1 x = lambda a : a + 10
2 print(x(5))
```

15

- Una función lambda puede tomar cualquier número de argumentos, pero sólo puede tener una expresión.

```
1 x = lambda a, b, c : a + b + c
2 print(x(3, 4, 5))
```

12

- Posee funciones especiales como **map**, **reduce** y **filter**. Su uso se aleja de la facilidad de entendimiento de Python pero en pandas puede añadir limpieza al código.

```
1 listado = [1, 3, 6, 9, 12, 15, 26, 37, 45, 56]
2 impares = list(filter(lambda x: (x%2 != 0), listado))
3 print(impares)
```

[1, 3, 9, 15, 37, 45]

- Así como aplicar la expresión **if** y **else** a los valores de una columna.

```
1 df['IMP'] = df['IMP'].map(lambda x: int(x) if isinstance(x,
    float) else x)
```

- Su uso en pandas facilita la creación de campos calculados:

```

1 import pandas as pd
2 import numpy as np
3
4 df = pd.DataFrame([1, 2, 3, 4], columns=['t'], dtype='int')
5 df['exp'] = df['t'].apply(lambda x: np.exp(x))
6 print(df)

```

	t	exp
0	1	2.718282
1	2	7.389056
2	3	20.085537
3	4	54.598150

En el caso que sea necesario hacer mención a una función, esta puede llamarse con la instrucción `.apply(lamdba x: funcion(x))` añadiendo limpieza al código:

```

1 import pandas as pd
2 import numpy as np
3
4 def estacionalidad(x):
5     if x == 7:
6         return 0.9
7     else:
8         return (1.01 ** x)
9
10 df = pd.DataFrame(pd.date_range(start='2018-01-31', periods=12,
11     freq='M', name='Fecha'))
12 df['t'] = np.arange(12)
13 df['i'] = 0.02
14 df['factor'] = df['t'].apply(lambda x : estacionalidad(x))
15 print(df)

```

Como hemos visto, una función lambda puede tomar cualquier número de argumentos, pero sólo puede tener una expresión. Para utilizar dos o más valores del dataframe en la formula, debemos especificar a lambda que lea los valores en horizontal.

Ejemplo: `df.apply(lamdba x: f(x.col1, x.col2), axis=1)`

```

1 import pandas as pd
2
3 df = pd.DataFrame([1, 2, 3, 4], columns=['Mes'], dtype="int")
4
5 df['Single Premium'] = df.apply(lambda x: single_risk_premium(x['
    age'], x['duration']), axis=1)

```

5.11. Selección de datos

```
1 import pandas as pd
2 df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
3                    index=[4, 5, 6], columns=['A', 'B', 'C'])
```

	A	B	C
4	0	2	3
5	0	4	1
6	10	20	30

- `df.at[4, 'B']` devuelve `2`
- `df.iat[1, 2]` devuelve `1`
- `df['A']` o `df.A` devuelve `0 0 10`
- `df.loc[5]` devuelve `0 4 1`
- `df.iloc[1]` devuelve `0 4 1`
- `df.loc[5].at['B']` devuelve `4`

```
1 import pandas as pd
2 df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
3                    index=[4, 5, 6], columns=['A', 'B', 'C'])
```

	A	B	C
4	0	2	3
5	0	4	1
6	10	20	30

Pandas también permite utilizar **operadores lógicos**:

- `df[df.B > 3]`

	A	B	C
5	0	4	1
6	10	20	30

Copiar dataframes

```
1 import pandas as pd
2 df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
3                    index=[4, 5, 6], columns=['A', 'B', 'C'])
```

	A	B	C
4	0	2	3
5	0	4	1
6	10	20	30

Con la instrucción `.copy()` se puede copiar un dataframe a otro para seguir operando libremente sin modificar el original. Ejemplo:

- `datos = df[df.B > 3].copy()`

	A	B	C	D
5	0	4	1	1
6	10	20	30	1

Adicionalmente, para filtrar datos con varias restricciones puede hacerse uso de la función `isin`:

```
1 import pandas as pd
2
3 df = pd.DataFrame([207,208,210,211,212,213,214,215], columns=['CTA
4 df['IMPORTE'] = 100.
5
6 existencias = [212, 213]
7 df2 = df[df['CTA'].isin(existencias)]
8 print(df2)
```

	CTA	IMPORTE
4	212	100.0
5	213	100.0

5.12. Estadísticos

Como hemos visto, el motor de Pandas estaba basado en NumPy, y por tanto podemos acceder fácilmente a todas las funciones estadísticas (en Series y DataFrames):

- `df.mean()`: media o promedio
- `df.median()`: mediana
- `df.quantile(q)`: Percentil. $0 \leq q \leq 1$ (por defecto 0.5)
- `df.max()`: valor máximo
- `df.min()`: valor mínimo
- `df.std()`: desv. estándar. Se puede fijar grados de libertad: `ddof=0` o `ddof=1` (*por defecto*)
- `df.var()`: varianza

```
1 import pandas as pd
2 import numpy as np
3 s = pd.Series(np.arange(101))
4 df = s.to_frame()
5 print(df.quantile(0.995)) #99.5
```


5.13. Ejercicio 7

Ejercicio_04_03

Con los cálculos del Ejercicio nº 1 (curva RFR de EIOPA), se pide:

- Añadir nueva columna con un ingreso constante de 100 durante 15 años
- Añadir nueva columna con el factor de descuento a utilizar
- Calcular el NPV de los ingresos descontados con la curva de riesgo

5.14. Ordenación

Con la siguientes funciones podemos ordenar los variables de una serie o dataframe:

- **df.sort_values()**. Sintaxis: `df.sort_values(by, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last')`. (*kind coincide con los mismos algoritmos que numpy*)

```
1 import pandas as pd
2 df = pd.DataFrame([1, 5, 9, 2], columns=['Importe'])
3 print(df.sort_values(by='Importe'))
```

	Importe
0	1
3	2
1	5
2	9

- **Series.argsort()**: Devuelve la posición ordenada de los valores de una serie. Sintaxis: `Series.argsort(axis=0, kind='quicksort')`

```
1 import pandas as pd
2 s = pd.Series([1, 5, 9, 2])
3 print(s.argsort())
```

0	0
1	3
2	1
2	2

- **Series.describe()**: Devuelve los estadísticos básicos.

```
1 import pandas as pd
2 import numpy as np
3 s = pd.Series(np.arange(100))
```

count	100.000000
mean	49.500000
std	29.011492
min	0.000000
25%	24.750000
50%	49.500000
75%	74.250000
max	99.000000

- Si queremos conocer el valor de un percentil concreto, **Series.quantile(n)** devuelve el n-ésimo percentil siendo n un valor entre 0 y 1. En el ejemplo: **s.quantile(0.995)** devuelve **98.505**

5.15. `reindex` y `rename`

En alguna ocasión, es posible que necesitemos reasignar un nuevo *index* a nuestro *data-frame*, o bien, resecarlo con los valores del 0 ... n. Para ello disponemos de la funcionalidad `.reindex()`:

- `df.reindex()`: resetea los valores por defecto, consecutivos del 0 .. n
- `df.reindex(['A'])`. En este caso, asigna los valores de A como index.

En otras ocasiones, es posible que necesitemos renombrar los nombres de las columnas. En ese caso podemos utilizar la función `df.rename()` con los siguientes argumentos:

- `mapper`: diccionario de nombres o función.
- `axis`: 0,1

```
1 import pandas as pd
2 df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
3 df = df.rename({0: 'x', 1: 'y'}, axis=0)
4 df = df.rename({'A': 2, 'B': 4, 'C': 5}, axis=1)
5 print(df)
```

	2	4	5
x	1	2	3
y	4	5	6

5.16. Bootstrap

Así como obtener muestras aleatorias de los datos de partida con `Series.sample()` con los siguientes atributos:

- `n`: número de muestras
- `frac`: Fraction of axis items to return. ej: 0.1
- `replace`: True (con reemplazamiento) o False (sin reemplazamiento)

```
1 import pandas as pd
2 import numpy as np
3
4 data = pd.Series([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
5 sample = data.sample(3)
6 print(sample)
```

1	0.2
4	0.5
2	0.3

- De los n datos originales $y_1, y_2 \dots y_n$, tomar una muestra con reemplazo, de tamaño n .
- Calculamos la media muestral con esa 'pseudomuestra'.
- Repetir B veces. Al final tendremos B estimaciones de la media.

```

1 import pandas as pd
2 import numpy as np
3
4 data = pd.Series([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
5 print(data.describe())
6
7 np.random.seed(100)
8 sample = data.sample(5, replace=True)
9 statistic = sample.mean()
10 print(statistic) # 0.2 vs 0.35

```

```

...
mean 0.35
...

```

```
0.2
```

```

1 import pandas as pd
2 import numpy as np
3
4 data = pd.Series([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
5 np.random.seed(100)
6
7 b = 1000
8 s = 5
9
10 boot = pd.Series(np.zeros(b))
11 for i in range(b):
12     sample = data.sample(s, replace=True)
13     statistic = sample.mean()
14     boot[i] = statistic
15
16 # Media: 0.348 vs 0.35 + Intervalo confianza 95%:
17 print(boot.mean(), boot.quantile(0.025), boot.quantile(0.975))

```

```
0.34842
```

En este caso, 0.3484 se aproxima más a 0.35

5.17. Agregación de datos: groupby

Para agregar los datos usa `df.groupby(by=['campo']).funcion` calculándose, entre otras, las siguientes posibles funciones a la hora de agrupar los datos:

- `.sum()`
- `.count()`
- `.mean()`
- `.max()`
- `.min()`
- `.std()`
- `.first()`

Se pueden mostrar el estadístico deseado o varios de ellos. En ese caso, la sintaxis debe ser con `.agg()` y `{}` en caso de mostrar varios campos. Por ejemplo:

- `df.groupby(['campo']).agg(['min','mean'])`
- `df.groupby(['campo']).agg({'campo1':'first','campo2':'mean'})`

```
1 import pandas as pd
2 df = pd.read_csv("Siniestros_Auto.csv", sep=";", decimal=",")
3 datos = df.groupby(['Delegacion']).agg({'Poliza': 'first', 'Importe': 'mean'})
4 print(datos)
```

	Poliza	Importe
Delegacion		
Centro	650372	414.654788
Este	650010	409.912736
Insular	650857	397.174533
Norte	651459	403.703011
Oeste	652037	411.773443
Sur	652726	403.047109

Igualmente podemos agregar varios campos: `df.groupby(['campo1','campo2']).f()`

En este caso, por defecto muestra las columnas con un formato esquemático o de árbol. En caso de querer respetar el formato tabular (necesario para tablas pivotantes) debemos usar `.reset_index()` para que añada un index al inicio.

```
1 import pandas as pd
2 df = pd.read_csv("Siniestros_Auto.csv", sep=";", decimal=",")
3 datos = df.groupby(['Delegacion', 'Categoria']).agg({'Importe': ['mean', 'min']}).reset_index()
4 print(datos)
```

	Delegacion	Categoria	Importe	
			mean	min
0	Centro	1_CAT	418.538364	5.41
1	Centro	2_CAT	415.717771	2.41
2	Centro	3_CAT	389.956300	4.57
...				
16	Sur	2_CAT	398.954946	8.98
17	Sur	3_CAT	403.195000	5.85

5.18. Ejercicio 8

Ejercicio_04_04

Con los mismos datos que el ejercicio nº 2, se pide crear un fichero .csv con los datos de siniestralidad agrupados por mes de ocurrencia y mes de pago.

5.19. Tablas pivotantes: pivot

Con Pandas podemos mostrar los datos en dos dimensiones en formato **tabla pivotante** (pivot table). La instrucción **df.pivot()** donde debemos especificar los siguientes campos:

- index: Eje vertical
- columns: Eje horizontal
- values: Valores a mostrar

Pivot

df

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

➔

```
df.pivot(index='foo',  
         columns='bar',  
         values='baz')
```

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

5.20. Ejercicio 9

Ejercicio_04_05

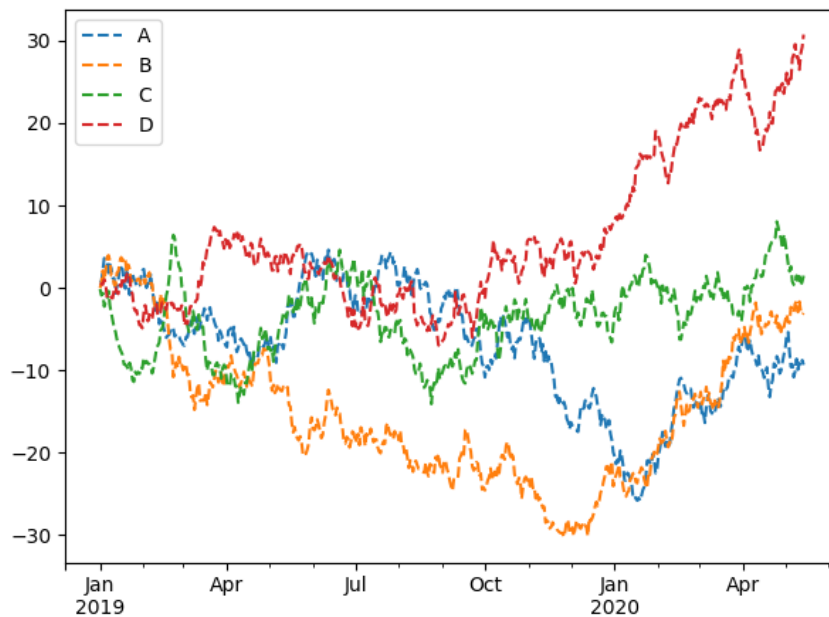
Con los mismos datos que el ejercicio nº 2, se pide:

- Realizar el triangulo de pagos por periodos mensuales, trimestrales y anuales, utilizando tablas pivotantes.
- Una vez calculado, exportar a un fichero Excel el triangulo anual de pagos acumulado.

5.21. Gráficos con matplotlib y pandas

- Pandas incluye algunas funcionalidades de la librería **matplotlib** incorporadas, la cual permite imprimir gráficas sin necesidad de llamarla previamente, con la simple instrucción: **Series.plot()** o **df.plot()**.
- El tipo de gráfico a mostrar será de líneas por defecto, a no ser que se especifique con el parámetro **kind=“tipo-gráfico”**.
Ejemplo: `df.plot(kind='hist', x=df.x)`

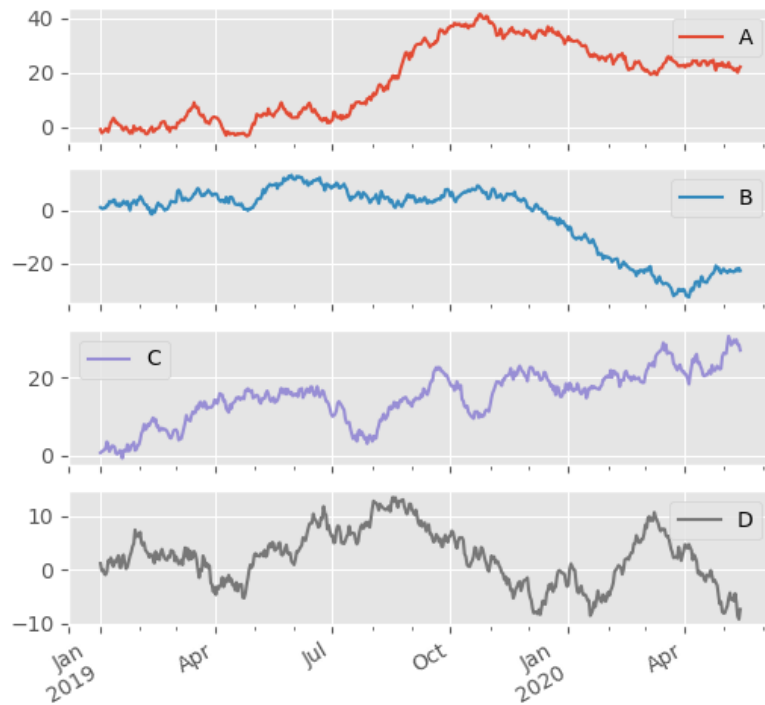
```
1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 index = pd.date_range('2019-01-01', periods=500)
6 df = pd.DataFrame(np.random.randn(500, 4), index=index, columns=
7     list('ABCD')).cumsum()
8
9 df.plot(style='--')
10 plt.show()
```



```

1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 index = pd.date_range('2019-01-01', periods=500)
6 df = pd.DataFrame(np.random.randn(500, 4), index=index, columns=
7     list('ABCD')).cumsum()
8 plt.style.use('ggplot')
9 df.plot(subplots=True)
10 plt.legend(loc='best')
11 plt.show()

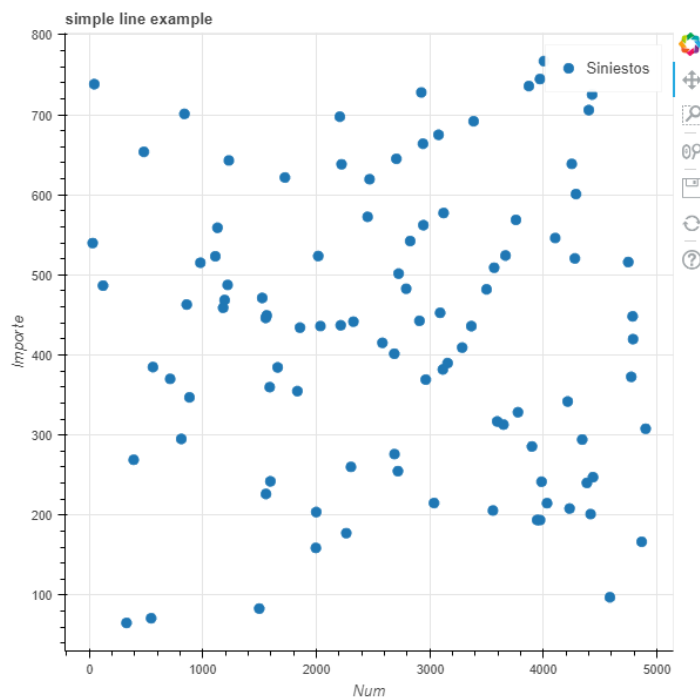
```



Gráficos interactivos

Bokeh es una biblioteca de visualización interactiva enfocada a visualizar los datos en navegadores web. Permite realizar gráficos interactivos y elegantes sobre conjuntos de datos muy grandes de forma rápida y sencilla. Ejemplo:

```
1 import bokeh.plotting as bplot
2 import numpy as np
3 import pandas as pd
4
5 claims = pd.read_csv("Siniestros_Auto.csv", sep=";", decimal=",")
6 bplot.output_file("lines.html") # Crea fichero HTML
7 p = bplot.figure(x_axis_label='Num', y_axis_label='Importe')
8 p.circle(claims.index, claims.Importe, legend="Siniestros", size=8)
9 bplot.show(p)
```



Nota

Versión utilizada de Pandas: 1.0.1

6. Chain-Ladder con Python

Texto obtenido de "Loss reserving based on run-off triangles implemented in Python using pandas and numpy", Francisco Gárate Santiago, 2019

The chain-ladder (CL) method (van Eeghen, 1981) is a well-known mathematical model in which uncertainty is not taken into account when estimating the amounts of future payments (in our case, the value of the financial provision to be constituted).

The CL method assumes that the patterns of loss development are based on historical behavior. The triangle is a figure of claims data in a two-dimensional matrix under 2 axes: date of occurrence and date of payment. The objective is to obtain the payment growth factor (age-to-age factors) or link ratios based on the trend observed (triangle development).

Pandas and Numpy

There is a python chain-ladder library (<https://github.com/jbogaardt/chainladder-python>) which facilitates calculations (determinist and stochastic). However, for this illustrative example of calculating an accumulated triangle of payments from a file of claims, we are going to use only the **pandas** and **numpy** libraries.

The first step is to import the data (10,000 invented claims from 2013 to 2018), check that there are no missing data, and show the first 5 records (to allow a visual check):

```
1 import pandas as pd
2 import numpy as np
3
4 data = pd.read_csv('data_claims.csv', sep=';')
5
6 # Check for missing data
7 print(data.isnull().any())
8 print(data.head())
```

Claim_ID	False
Policy_ID	False
Accident_Date	False
Report_Date	False
Payment_Date	False
Indemnity	False

Once the data are imported and formatted the date fields, we can create new fields with the year of the accident and payment (ie. **Accident_Year**) or, in this case, modify the existing ones:

```
1 data['Accident_Date'] = pd.to_datetime(data['Accident_Date'])
2 data['Payment_Date'] = pd.to_datetime(data['Payment_Date'])
```

```

3
4 data['Accident_Date'] = data['Accident_Date'].dt.to_period(freq='Y
    ')
5 data['Payment_Date'] = data['Payment_Date'].dt.to_period(freq='Y')

```

We calculate a new field called 'Dev' with the overtime period (or development) as the difference between the two dates (as integer type):

```

1 data['Dev'] = data['Payment_Date'].astype(int) - data['
    Accident_Date'].astype(int)

```

We group by year of accident and year of development using `groupby()`, and add the amounts of the claims:

```

1 datagg = data.groupby(['Accident_Date', 'Dev']).agg({'Indemnity': '
    sum'}).reset_index()

```

Once grouped, we save the data in a 2-D (pivot) table based on these two variables:

```

1 triangle = datagg.pivot(index='Accident_Date', columns='Dev',
    values='Indemnity')

```

Note that at any time we can visually check the data by exporting it to a csv file with the instruction: `triangle.to_csv('triangle.csv',sep=';',decimal=',')` or by showing it in screen using `print(triangle)`.

The triangle is calculated with the payments accumulated over time. The function `cumsum()` can be used to add data from a dataframe, although by default, it adds the data of a column, so must sum rows (`axis=1`). Easiest impossible:

```

1 accumulate = triangle.cumsum(axis=1)

```

Now is the crowning moment, that is, when the payments pattern or development factors are calculated: incremental quotas, link ratios or loss development factors (LDFs), and the cumulative quotas, final ratios or cumulative development factors (CDFs).

The calculation in this step is the main difference among the different methods based on run-off triangles: minimum/maximum exclusion, averages, seasonal nature, working with variances, covariances, resamples, distributions, and MLE with Poisson among others.

It is important to remember that the calculation of provisions has an essential impact on the accounts of the insurance companies, and the expert judgment of the actuary must play a role in this selection.

I hope that the code shown for a classic CL is sufficiently illustrative. For convenience, I have **lists** of ones instead of `pd.Series()`, and `n-1` is used to avoid division by zero in the last value of the triangle. We assume that 100 % of the payments have been processed and there are no more data in the tail.

```

1  n = len(accumulate)-1
2
3  LDFs = np.ones(n)
4  for i in range(n):
5      SumDev = pd.Series(accumulate[i][:n-i]).sum(skipna=True)
6      SumDevNext = pd.Series(accumulate[min(i+1,n)][:n-i]).sum(
7          skipna=True)
8      LDFs[i] = SumDevNext/SumDev
9
10 CDFs = np.ones(n+1) #fix last value as 1.
11 for i in range(n):
12     CDFs[i] = np.prod(LDFs[i:n])

```

Calculating the payment pattern with these values is easy. The **reciprocal()** function returns the inverse of the number (1/x). In the incremental rates column, NaN is replaced by the value of the accumulated rates column because the first value of **diff()** is always NaN.

```

1  payment_pattern = pd.DataFrame(data=np.reciprocal(CDFs), columns=[ '
2      Cum' ])
3  #First value of diff() is always NaN
4  payment_pattern['Incr'] = payment_pattern['Cum'].diff().fillna(
5      payment_pattern['Cum'])

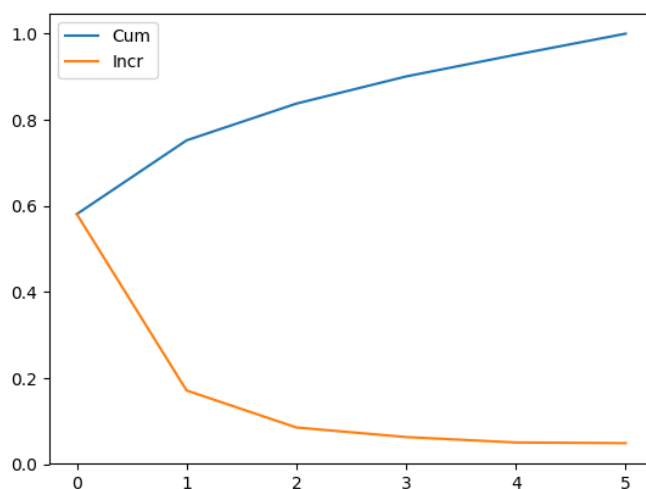
```

The graph can be seen below:

```

1  import matplotlib.pyplot as plt
2  payment_pattern.plot()
3  plt.show()

```



Now, it is only necessary to multiply payments by the final development ratios (to the

last value) to obtain the value of the last payment, that is, by subtracting the current payments we obtain the value of the necessary monetary reserve.

To calculate the total amount of current payments we have two options:

1. sum horizontally (axis=1) the incremental triangle,
2. obtain the diagonal of the accumulated triangle.

Although the first option is easier, `triangle.sum(axis=1, skipna=True).sum()`, in this example the `diag()` function of numpy is used to illustrate the possibilities of pandas. Specifically, `flipud` serves to rotate the triangle because by default the diagonal function returns the diagonal starting in `[0,0]` and ending in `[n,n]`.

```
1 diagonal = np.diag(np.flipud(accumulate))
2 payments = pd.Series(data=diagonal)
```

Ultimate calculation:

```
1 ultimate = np.dot(CDFs, payments)
```

Calculation of the reserve:

```
1 reserve = ultimate - np.sum(payments)
2 print(reserve)
```

```
209718.3485682155
```

Frequency

Do you want to calculate ratios with information grouped quarterly rather than annually? While this may be a tedious task if the triangles are in Excel, we only need to change two lines to do it here: the variable 'Y' to 'Q' in the previous code.

```
1 data['Accident_Year'] = data['Accident_Date'].dt.to_period(freq='Q')
2 data['Payment_Year'] = data['Payment_Date'].dt.to_period(freq='Q')
```

The list of possible values for the 'freq' attribute can be found in the timeseries documentation of pandas: https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases, such as:

- Y: year end frequency
- Q: quarter end frequency
- M: month end frequency
- W: weekly frequency

Factor selection

It's pretty common, when calculating the factors, to make a selection excluding the maximums and minimums, or whatever an expert judgment considered.

In this case, we can create a dataframe to do the factor calculation, and operate as necessary, for example ordering it to exclude first and last factors:

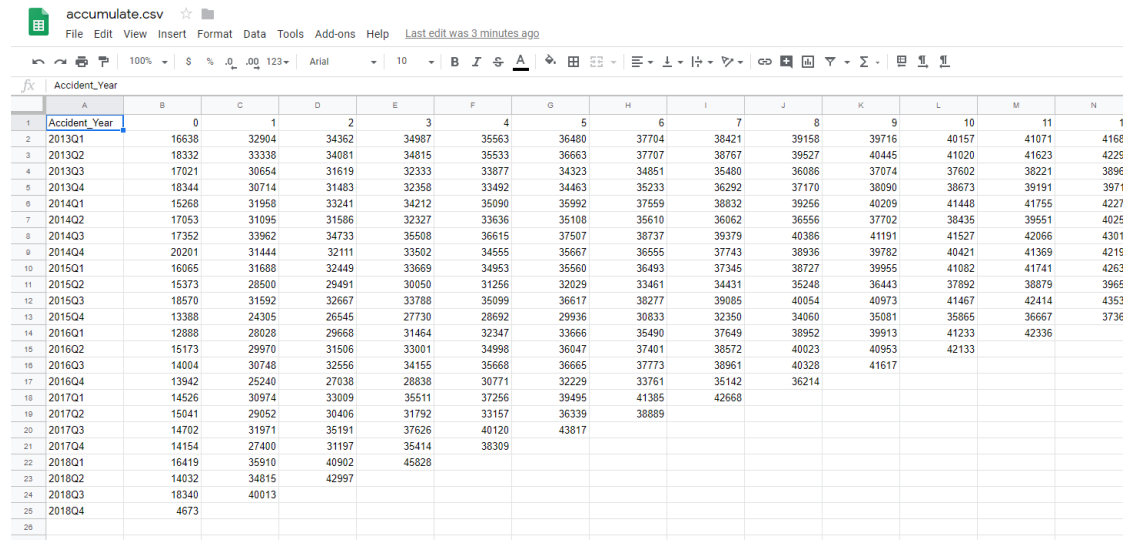
```
1 LDFs = np.ones(n)
2
3 for i in range(n):
4     SumDev = pd.Series(accumulate[i], name='SumDev')
5     SumDevNext = pd.Series(accumulate[i+1], name='SumDevNext')
6     LDF_w = pd.concat([SumDev, SumDevNext], axis=1)[-i-1]
7     LDF_w['Perc'] = LDF_w['SumDevNext'] / LDF_w['SumDev']
8     datasort = LDF_w.sort_values(by=['Perc'])
9     if len(datasort) > 2:
10         ratio = datasort[1:-1]['SumDevNext'].sum() /
11             datasort[1:-1]['SumDev'].sum()
12     else:
13         ratio = datasort['SumDevNext'].sum() / datasort['SumDev'].sum()
14     LDFs[i] = ratio
```

Results

Freq	Factor selection	Amount
Y	No	209.718,35
Q	No	215.174,03
Y	Yes	202.931,33
Q	Yes	210.891,32

Exporting

The resulting triangle exported to CSV then looks like the following screenshot:



	Accident_Year	0	1	2	3	4	5	6	7	8	9	10	11	1
2	2013Q1	16638	32904	34362	34987	35563	36480	37704	38421	39158	39716	40157	41071	4168
3	2013Q2	18332	33338	34081	34815	35533	36663	37707	38767	39527	40445	41020	41623	4225
4	2013Q3	17021	30654	31619	32333	33877	34323	34851	35480	36086	37074	37602	38221	3896
5	2013Q4	18344	30714	31483	32358	33492	34463	35233	36292	37170	38090	38673	39191	3971
6	2014Q1	15268	31958	33241	34212	35090	35992	37559	38832	39256	40209	41448	41755	4227
7	2014Q2	17053	31095	31586	32327	33636	35108	35610	36062	36556	37702	38435	39551	4025
8	2014Q3	17352	33962	34733	35508	36615	37507	38737	39379	40386	41191	41527	42066	4301
9	2014Q4	20201	31444	32111	33502	34555	35667	36555	37743	38936	39782	40421	41369	4215
10	2015Q1	16065	31688	32449	33669	34953	35560	36493	37345	38727	39955	41082	41741	4263
11	2015Q2	15373	28500	29491	30050	31256	32029	33461	34431	35248	36443	37892	38879	3965
12	2015Q3	18570	31592	32667	33788	35099	36617	38277	39085	40054	40973	41467	42414	4353
13	2015Q4	13388	24305	26545	27730	28692	29936	30833	32350	34060	35081	35865	36667	3736
14	2016Q1	12888	28028	29668	31464	32347	33666	35490	37649	38952	39913	41233	42336	
15	2016Q2	15173	29970	31506	33001	34998	36047	37401	38572	40023	40953	42133		
16	2016Q3	14004	30748	32556	34155	35668	36665	37773	38961	40328	41617			
17	2016Q4	13942	25240	27038	28838	30771	32229	33761	35142	36214				
18	2017Q1	14526	30974	33009	35511	37256	39495	41385	42668					
19	2017Q2	15041	29052	30406	31792	33157	36339	38889						
20	2017Q3	14702	31971	35191	37626	40120	43817							
21	2017Q4	14154	27400	31197	35414	38309								
22	2018Q1	16419	35910	40902	45828									
23	2018Q2	14032	34815	42997										
24	2018Q3	18340	40013											
25	2018Q4	4673												
26														

Resumen:

Patrón de pagos

- Partiendo de un triángulo de pagos acumulado es simple calcular los link-ratios para hallar el **patrón de pagos**.
- Para evitar periodos sin dato, se debe hacer un reindex del dataframe: **.reindex(columns=range(10))**

```
1 triangle = triangle.reindex(columns=range(50))
```

- Ejemplo de cálculo del vector **LDF (Loss Development Factor)**:

```
1 n = len(accumulate)-1
2 LDFs = np.ones(n)
3
4 for i in range(n):
5     SumDev = pd.Series(accumulate[i][:n-i]).sum(skipna=True)
6     SumDevNext = pd.Series(accumulate[min(i+1,n)][:n-i]).sum(
7         skipna=True)
8     LDFs[i] = SumDevNext/SumDev
```

- El anterior cálculo se basa en n iteraciones que cumplimentan un array creado previamente con *unos*, y se utiliza *skipna=True* para evitar periodos de desarrollos sin datos.
- Partiendo del LDF, calculamos el **CDF o Cumulative Development Factors** como una iteración del producto de los LDF (en este caso fijamos la cola en 100 % añadiendo un elemento más a la lista de *unos* que creamos):

```
1 CDFs = np.ones(n+1)
2 for i in range(n):
3     CDFs[i] = np.prod(LDFs[i:n])
```

- Y por último, calculamos el patrón de pagos invirtiendo el CDF y hallando sus incrementales con *diff()*. Como primer valor de *diff* es NaN, lo reemplazamos con *.fillna()* con el primer valor del patrón:

```
1 payment_pattern = pd.DataFrame(data=np.reciprocal(CDFs), columns=['
2     Cum'])
3 payment_pattern['Incr'] = payment_pattern['Cum'].diff().fillna(
4     payment_pattern['Cum'])
```

Chainladder clásico

- Por último, tenemos diferentes opciones para hallar el **valor último de los pagos** (*ultimate*) así como la reserva a constituir como la diferencia entre el valor último y los pagos. Ejemplo:

```
1 # Payments = diagonal de flipud o suma de triangulo incremental
2 diagonal = np.diag(np.flipud(accumulate))
3 payments = pd.Series(data=diagonal)
4
5 # Ultimate = CDFs * payment_k
6 ultimate = np.vdot(CDFs, payments)
7
8 # Reserve = ultimate - Sum(payments)
9 reserve = ultimate - np.sum(payments)
10 print(reserve)
```

Chainladder estocástico

Como se ha visto en el tema 7, podemos usar la técnica de remuestreo o bootstrap para la estimación de la provisión IBNR con dos enfoques:

- Bootstrap vinculado: el remuestreo se realiza directamente desde las observaciones en un modelo de regresión.
- Bootstrap de residuos: el remuestreo se aplica a los residuos del modelo (residuos estandarizados Pearson o Chi).

6.1. Ejercicio 10

Ejercicio 08_01

Con el fichero “*data_claims.csv*” que recoge el fichero de pagos histórico, se pide:

- Calcular el triángulo acumulado de pagos
- Calcular el patrón de pagos y graficarlo.
- Calcular del importe último de la siniestralidad y de la reserva IBNR siguiendo un ChainLadder clásico (utilizar frecuencia mensual y anual)
- Calcular de los link-ratios con selector de factores (despreciando el factor máximo y mínimo)

7. Probabilidad y estadística

7.1. Pandas y Numpy

Numpy es *de facto* el estándar utilizado en Python para cálculos estadísticos, tales como:

- Medidas de centralización (media, mediana, moda...)
- Medidas de posición (percentiles y cuantiles)
- Medidas de dispersión (desviaciones, varianzas...)

Por otro lado, **Pandas** nos facilita el análisis de los datos agrupándolos en dataframes, facilitándonos la tarea de visualizarlos.

En cambio, **Scipy** nos va a aportar un manejo avanzado para manejar con facilidad distribuciones de probabilidad.

- Tal como hemos visto anteriormente, **NumPy** y **Pandas** disponen de diferentes métodos y maneras de calcular estadísticas descriptivas para un conjunto de datos.
- La mayoría devuelven un dato, como `.sum()` o `.mean()`, pero algunas de ellas devuelven un objeto del mismo tamaño, como `.cumsum()`.
- En general, estos métodos toman como argumento un eje (axis) del dataframe, ya sea con su nombre o posición.

```
1 import pandas as pd
2 import numpy as np
3
4 data = pd.Series(data=np.random.randint(0, 50, size=100))
5 print(data.describe())
```

Medidas de centralización: media, mediana y moda:

- **np.mean(x)**: Media (o Promedio). Suma de todos los valores dividida entre el número de valores.
- **np.median(x)**: Mediana. Valor de la variable que deja el mismo número de datos antes y después que él.
- **np.mode(x)**: Moda. Valor con mayor frecuencia en una distribución de datos.

La tendencia central (por ejemplo, la media y la mediana) puede que no sea el único valor estadístico que queramos conocer. Posiblemente también queramos conocer información de la variabilidad de los datos.

- **np.percentile(a, q)**: Percentil. Se define el cuantil de orden α como un valor de la variable por debajo del cual se encuentra una frecuencia acumulada α .

- **np.quantile(a, q)**: Cuantil. Equivalente a percentile, excepto con q en el rango [0, 1].
- **np.std(a)**: desvi. estándar, (ddof=0) El divisor utilizado en el cálculo es N-ddof, donde N representa el número de elementos.
- **np.var(a)**: varianza
- **np.cov(a)**: covarianza

Como se ha visto anteriormente, estas funciones también pueden ser aplicadas tanto a **Series** como **Dataframes** de **Pandas**:

```
1 import pandas as pd
2 import numpy as np
3 s = pd.Series(np.arange(101))
4 print(s.var())
5
6 df = s.to_frame()
7 print(df.var())
```

```
858.5
0    858.5
```

Para un conjunto de datos que dispongan de una variable discreta, podemos realizar fácilmente una tabla de frecuencias con el objeto de conocer cuantas veces se repite el valor de la variable.

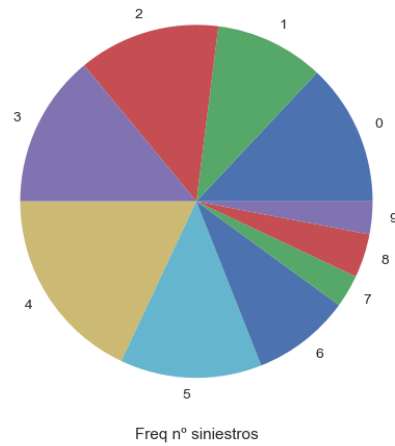
Para ello usamos **pandas.crosstab(index=df)**, que devuelve otro DataFrame independiente.

- **index**: la variable que queremos contar
- **columns**: el nombre de la columna de salida, por ejemplo: columns=['freq']

```
1 import pandas as pd
2
3 df = pd.read_csv('siniestros.csv')
4 tab = pd.crosstab(index=df.num_sini, columns=['freq'])
5 print(tab)
```

Crear una gráfico de tarta:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 df = pd.read_csv('siniestros.csv')
4 tab = pd.crosstab(index=df.num_sini, columns=['freq'])
5
6 plt.style.use('seaborn')
7 plt.pie(tab, labels=tab.index)
8 plt.xlabel('Freq no siniestros')
9 plt.savefig('freq_siniestros.png')
10 plt.show()
```

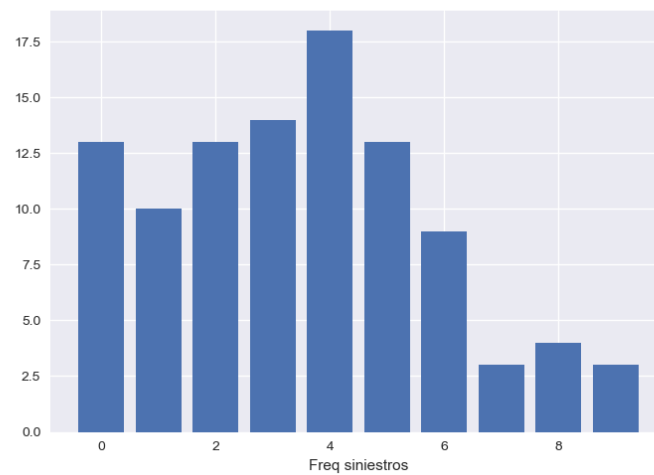


Crear una gráfico de barras:

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 df = pd.read_csv('siniestros.csv')
4 tab = pd.crosstab(index=df.num_sini, columns=['freq'])
5
6 plt.style.use('seaborn')
7 plt.bar(tab.index, tab['freq'])
8 plt.xlabel('Freq siniestros')
9 plt.savefig('bar_freq_sini.png')
10 plt.show()

```



7.2. Scipy

La librería **SciPy** (*Scientific computing*, pronunciado “Sigh Pie”) posee diferentes módulos (**stats**, **linalg**, **interpolate...**) orientados a diversas áreas científicas. De momento, el que nos interesa para estadística es **scipy.stats**.

- Sitio: <https://www.scipy.org/scipylib/index.html>
- Documentacion: <https://docs.scipy.org/doc/scipy/reference/tutorial/stats.html>
- Licencia open-source propia.
- **scipy.stats** aporta algunas funciones que pueden complementar a NumPy:

```
1 import numpy as np
2 import scipy.stats as st
3 np.random.seed(999)
4 x = np.random.normal(size=1000)
5 print(np.median(x))
6 print(st.scoreatpercentile(x, 50)) # mediana
7 print(st.scoreatpercentile(x, 95))
```

```
0.007576375024717093
0.007576375024717093
1.6906947688452745
```

7.3. Distribuciones de probabilidad con SciPy

Sin embargo, donde destaca **SciPy** es en el manejo de **distribuciones de probabilidad**:

- En estadística, toda variable aleatoria se distribuye en base a una función de probabilidad.
- Dependiendo del evento aleatorio, las distribuciones pueden ser **discretas** (toma ciertos valores aislados -puntos- dentro de un intervalo) o **continuas** (puede tomar cualquier valor real del rango).

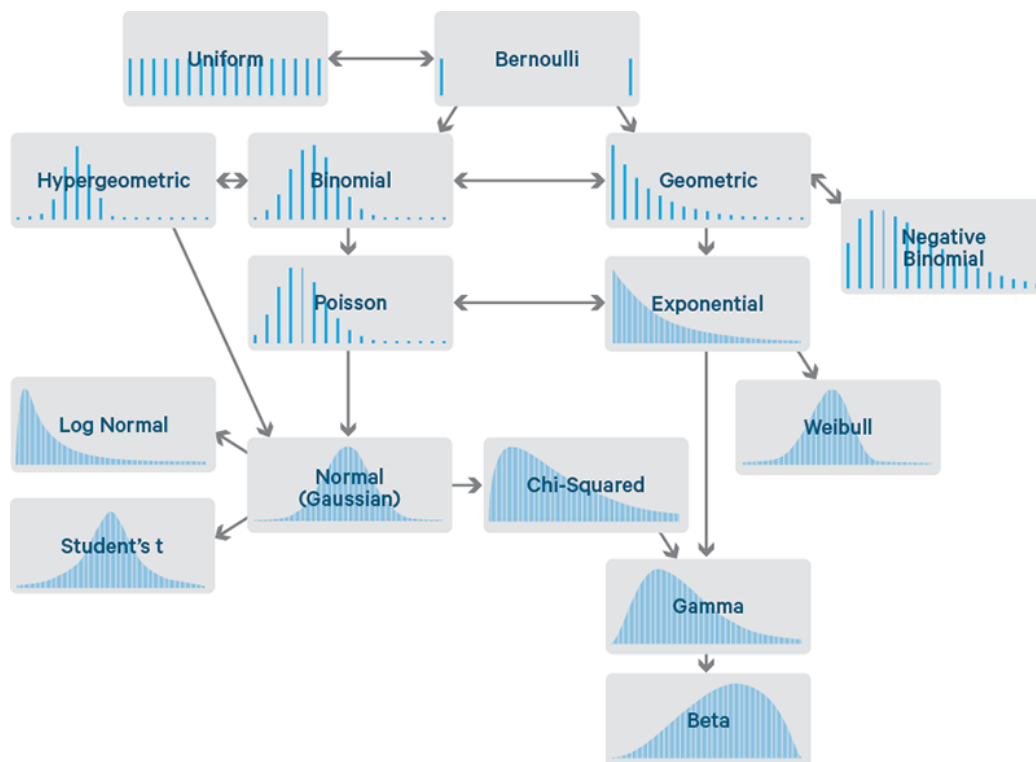
La versión 1.4 de SciPy tiene un catálogo de 98 distribuciones continuas y 12 discretas:

```
1 import scipy.stats as st
2 print(dir(st))
```

Las más conocidas:

- `.norm()`: Normal (Gausiana)
- `.lognorm()`: Lognormal
- `.expon()`: Exponencial
- `.t()`: Student's t
- `.chi2()`: Chi-cuadrado
- `.gamma()`: Gamma
- `.beta()`: Beta
- `.bernoulli()`: Bernoulli
- `.binom()`: Binomial
- `.poisson()`: Poisson

<https://docs.scipy.org/doc/scipy/reference/tutorial/stats/continuous.html>



Función de densidad

- Cada distribución puede ilustrarse con su **función de densidad de probabilidad** (pdf), es decir, el eje horizontal es el conjunto de posibles resultados numéricos y el eje vertical describe la probabilidad de resultados.
- En la función de densidad de probabilidad, la **suma** de las alturas de las líneas (distrib. discretas) o las áreas bajo las curvas (distrib. continuas) son **siempre 1**.

Tomaremos como ejemplo la distribución normal (gaussina) como la función de distribución de nuestra variable a estudiar. La distr. normal es ampliamente usada en inferencia estadística, ya que con muestras grandes los errores *se aproximan bien* a una distribución normal.

La función de densidad de probabilidad normal para un número real x es:

$$f(x) = \frac{\exp(-x^2/2)}{\sqrt{2\pi}}$$

`norm.pdf(x, loc=0, scale=1)` para el valor x , siendo:

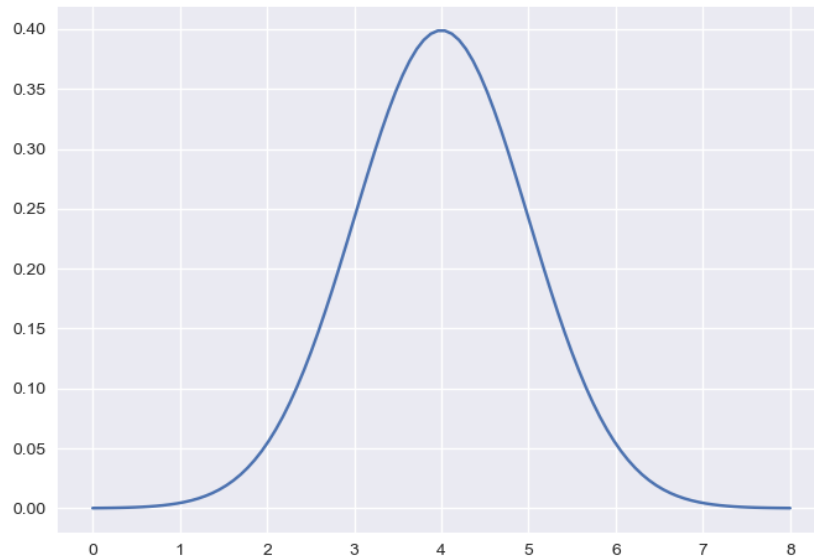
- **loc**: media de la distribución
- **scale**: desviación estándar de la distribución.

```
1 import scipy.stats as st
2 a = st.norm.pdf(0.5)
3 print(a)
```

```
0.3520653267642995
```

Ejemplo usando *mi_distribución.pdf(x)*

```
1 import numpy as np
2 import scipy.stats as st
3 import matplotlib.pyplot as plt
4 gaussian = st.norm(loc=4.0, scale=1.0)
5 x = np.linspace(0.0, 8.0, 100)
6 y = gaussian.pdf(x)
7 plt.style.use('seaborn')
8 plt.plot(x, y)
9 plt.show()
```



- **.cdf(x, loc=0, scale=1)**: Función de distribución acumulativa (*Cumulative distribution function*)
- **.ppf(q, loc=0, scale=1)**: Función de punto porcentual (*Percent point function*) (la inversa de cdf, equivalente a los percentiles).

```

1 import scipy.stats as st
2 gaussian = st.norm(loc=4.0, scale=1.0)
3 a = st.norm.cdf(1.65)
4 b = st.norm.ppf(0.95) #percentil
5 print(a, b)

```

```
0.9505285319663519
```

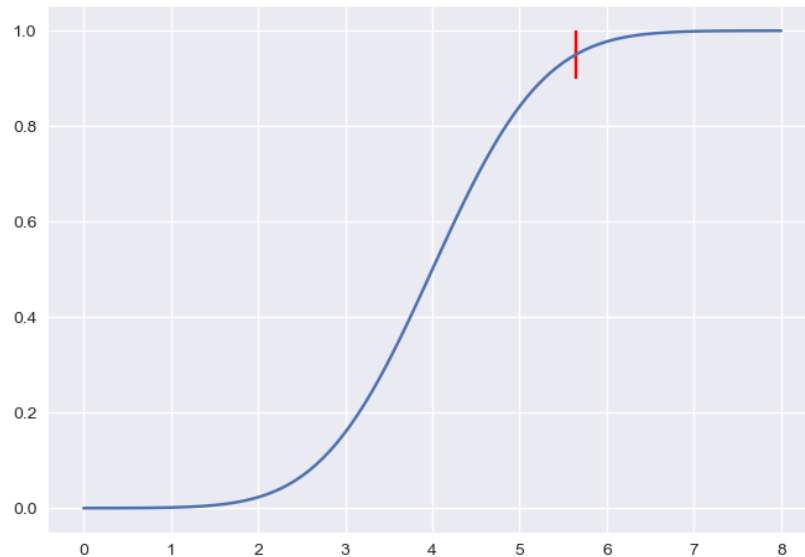
```
1.6448536269514722
```

Ejemplo usando **.cdf(x)** y **.ppf(0.95)** para calcular el percentil 95

```

1 import numpy as np
2 import scipy.stats as st
3 import matplotlib.pyplot as plt
4 gaussian = st.norm(loc=4.0, scale=1.0)
5 x = np.linspace(0.0, 8.0, 100)
6 y = gaussian.cdf(x)
7 plt.style.use('seaborn')
8 plt.plot(x, y)
9 plt.vlines(gaussian.ppf(0.95), 0.9, 1, 'r')
10 plt.show()

```



- `.norm.mean(loc=0, scale=1)`: Media de la distribución
- `.norm.median(loc=0, scale=1)`: Mediana de la distribución
- `.norm.var(loc=0, scale=1)`: Varianza de la distribución
- `.norm.std(loc=0, scale=1)`: Desviación estándar de la distribución

```
1 import scipy.stats as st
2 media = st.norm.mean(loc=0, scale=1) #mu=0 sigma=1
3 mediana = st.norm.median(loc=0, scale=1)
4 var = st.norm.var(loc=0, scale=1)
5 std = st.norm.std(loc=0, scale=1)
6 print(media, mediana, var, std)
```

```
0.0 0.0 1.0 1.0
```

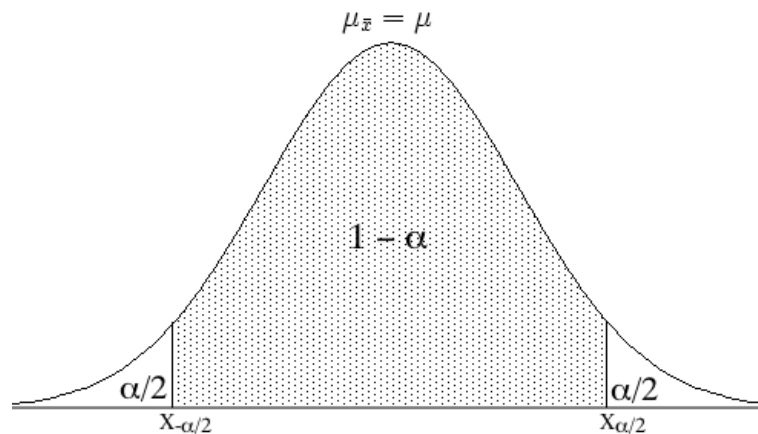
7.4. Ejercicio 11

Ejercicio_05_01

Una compañía deberá indemnizar con 167 euros por día hospitalizado en caso de accidente. Los datos disponibles de siniestralidad histórica muestran una probabilidad anual de accidente del 1.2 %, con una duración de la estancia que sigue una distribución normal de media 7.5 y desviación típica de 1.75.

Se pide calcular la prima de riesgo necesaria para que con un 99.5 % de probabilidad las duraciones de la estancia estén por debajo de la esperada.

- `.norm.interval(alpha, loc=0, scale=1)`: Puntos finales del rango que contiene el porcentaje alfa de la distribución.



```

1 import scipy.stats as st
2 alpha = 0.95
3 intervalo = st.norm.interval(alpha, loc=0, scale=1)
4 print(intervalo)

```

```

(-1.959963984540054, 1.959963984540054)

```

7.5. Creación de intervalos

`mi_distribución.stats(loc=0, scale=1, moments='mvsk')`: Devuelve los momentos que especifiquemos:

- **m**: Media (*mean*)
- **v**: Varianza (*variance*)
- **s**: Sesgo (*skew*)
- **k**: kurtosis

```

1 import scipy.stats as st
2 gaussian = st.norm(loc=4.0, scale=1.0) #mu=4 #sigma=1
3 mean, var, skew, kurt = gaussian.stats(moments='mvsk')
4 print(mean, var, skew, kurt)

```

```

4.0 1.0 0.0 0.0

```

7.6. Ejercicio 12

En una distribución normal estándar, al ser simétrica respecto de su media μ , es posible relacionar fácilmente el resto de variables aleatorias.

$$\text{Si } X \sim N(0, 1) \text{ entonces } Z = \frac{x - \mu}{\sigma} \Rightarrow Z \sim N(0, 1) \dots Pr(x < z) = \Phi\left(\frac{x - \mu}{\sigma}\right)$$

Ejercicio_05_02

La siniestralidad agregada del seguro de Salud sigue una distribución normal de media 110€. Sabiendo que el 97.5% de la siniestralidad no excede los 127.64€. Calcular:

- a) La desviación típica (σ)
- b) La probabilidad de que una póliza elegida al azar posea una siniestralidad agregada entre 101 y 119 €.

7.7. Otras distribuciones conocidas

7.7.1. Distribución lognormal

- Los pagos de reclamaciones asociados a catástrofes naturales se calculan a menudo como una distribución conjunta de frecuencia y gravedad, donde la frecuencia mide el número de reclamaciones o eventos (por ejemplo, huracanes y terremotos) y la gravedad mide la magnitud de la pérdida de una reclamación o evento individual.
- Una entidad no sólo se preocupará por el importe esperado o medio de los siniestros, sino también por el importe total probable de lo que podrían ser los siniestros.

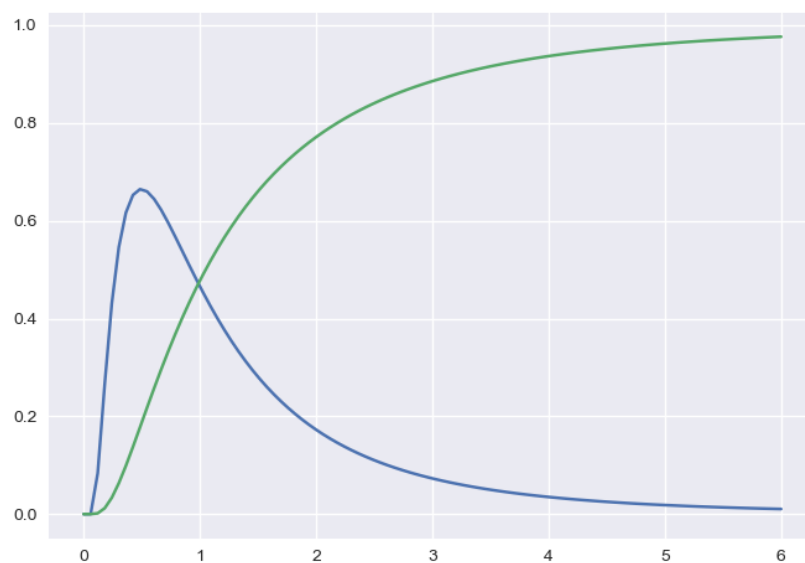
Ejemplo con la función de lognormalidad con `st.lognorm(s=sigma, scale=exp(mu))`

s = sigma std, mu = mean

sigma = std / la media de $\ln(x)$

std = desviación estandar de $\ln(x)$

```
1 import numpy as np
2 import scipy.stats as st
3 import matplotlib.pyplot as plt
4 # std normal
5 sigma = 0.859455801705594
6 # mean normal
7 mu = 0.418749176686875
8 lognormal = st.lognorm(s=sigma, scale=np.exp(mu))
9 x = np.linspace(0, 8.0, 100)
10 plt.style.use('seaborn')
11 plt.plot(x, lognormal.pdf(x))
12 plt.plot(x, lognormal.cdf(x))
13 plt.show()
```



7.7.2. Distribución gamma

`st.gamma(a=alpha, scale=1/lambda)`

a: alpha

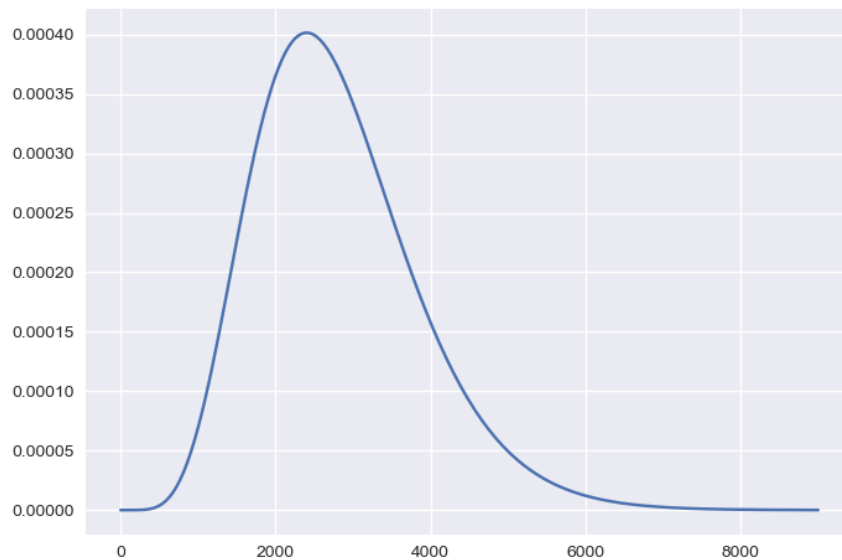
scale: $\theta = 1 / \lambda$ (por defecto = 1)

`st.gamma.pdf(x, a, scale)`

x: valor para evaluar la función

```
1 import numpy as np
2 import scipy.stats as st
3 from matplotlib import pyplot as plt
4
5 alpha = 7.
6 lamdba = 0.0025
7 x = np.linspace(0, 9000, 200)
8 y = st.gamma.pdf(x, a=alpha, scale=1/lamdba)
9
10 plt.style.use('seaborn')
11 plt.plot(x, y)
12 plt.show()
```

Cuando a es un entero, gamma se reduce a la distribución Erlang, y cuando $a=1$ a la distribución exponencial.



7.8. Ejercicio 13

Ejercicio obtenido de “Economic Scenario Generators. A Practical Guide (pags 26-28)”
Society of actuaries, 2016

*Nota: En el documento del SOA, la fórmula de la σ en verdad es la varianza $V(X)$,
faltaría aplicar la raíz cuadrada aunque el resultado final es válido.*

Ejercicio_05_03

Supongamos que para un evento dado, el tamaño de la distribución de pérdidas (gravedad) de nuestra línea de negocio se describe mejor mediante una distribución gamma de $\alpha = 7$ y $\lambda = 0.0025$.

La distribución gamma (Γ como un escalar) con parámetros forma = λ y escala = α tiene como función de densidad:

$$f(x) = \lambda e^{-\lambda x} \frac{(\lambda x)^{\alpha-1}}{\Gamma(\alpha)}$$

El valor esperado (media) y la varianza de una variable aleatoria x de distribución gamma son:

$$\mathbf{E}(\mathbf{x}) = \alpha\theta = \alpha/\lambda \quad y \quad \mathbf{Var}(\mathbf{x}) = \alpha\theta^2 = \alpha/\lambda^2$$

Se pide:

- Calcular la media y la desviación estándar de la siniestralidad total
- Calcular la siniestralidad esperada en los siguientes percentiles: 90th, 95th, 99th, 99.6th y 99.8th
- Aplicar 10.000 simulaciones aleatorias con numpy (`np.random.gamma`) y calcular los anteriores percentiles.

NOTAS:

- En Scipy (`st.gamma`), los parámetros de la gamma son: `a=alpha` y `scale=1/lamdba` (`theta`)
- Los percentiles de una distribución corresponde con el `.ppf` (función de punto porcentual). Ejemplo: `st.gamma.ppf(0.5,a=alpha,scale=1/lambda)`
- Los parámetros de numpy son `np.random.gamma(alpha, 1/lamdba, size=10000)`

Información de interés

Versión utilizada de Scipy: 1.4.1

Economic Scenario Generators – A Practical Guide (SOA, 2016): <https://www.soa.org/resources/research-reports/2016/2016-economic-scenario-generators/>

8. Estudio de muestras

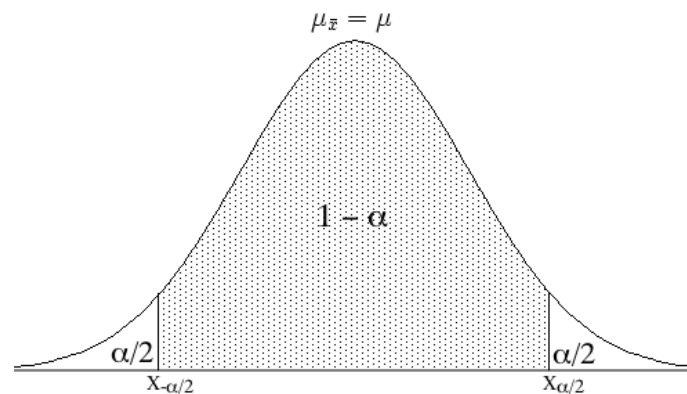
La **estadística inferencial** o inferencia estadística comprende los métodos y procedimientos que por medio de la inducción determina propiedades de una población estadística, a partir de una parte de esta conocida como muestra.

En esta sesión veremos como calcular los siguientes aspectos en Python:

- Intervalos de confianza y p-valor.
- Contraste de hipótesis
- Bootstrap

8.1. Intervalos de confianza

- Las aproximaciones estadísticas están sujetas a error, por lo que nunca podemos estar completamente seguros de que nuestras **estimaciones** sean ciertas, aunque podemos medir el grado de incertidumbre.
- El **intervalo de confianza** es el rango de valores entre los cuales se estima que estará el valor desconocido con cierta probabilidad de acierto.
- La prob. de éxito se representa con $1 - \alpha$ y se denomina **nivel de confianza**.



- Si para una población, tomamos n muestras obtenemos n valores medios.
- En muestras suficientemente **grandes**, estos valores medios tienden a seguir una distribución normal (a no ser que sepamos su distribución de probabilidad) con un valor medio μ y desviación estándar $\frac{\sigma}{\sqrt{n}}$ (denominada **error estándar** de la media).
- En cambio, si la muestra es **pequeña** ($n < 30$), la práctica es usar la distribución t de Student en vez una distribución de medias gaussiana o normal. Así, utilizaríamos por tanto dicha distribución para el cálculo del intervalo.
- El **intervalo de confianza** se determina calculando una estimación “central” (conociendo la distribución teórica que sigue el parámetro a estimar) y luego determinando su margen de error.

- Así, si tomamos por ejemplo, $1 - \alpha$ igual a 0.95 (es decir $\alpha=0.05$):

$$\begin{aligned} P(z \leq Z \leq z) &= 1 - \sigma = 0,95 \\ \Phi(z) = P(<z) &= 1 - \frac{\sigma}{2} = 0,975 \\ z = \Phi^{-1}(\Phi(z)) &= \Phi^{-1}(0,975) = 1,96 \end{aligned}$$

Intervalo de confianza (área entre cuantiles de orden 0.025 y 0.975):

Extremo inferior: $\bar{X} - 1,96 \frac{\sigma}{\sqrt{n}}$

Extremo superior: $\bar{X} + 1,96 \frac{\sigma}{\sqrt{n}}$

Para calcular los **intervalos de confianza en Python** podemos usar las siguientes bibliotecas:

- `scipy.stats`
- `statsmodels.stats.weightstats`

SciPy

En `scipy.stats` disponemos la función `interval(alpha, loc=0, scale=1)` que nos proporciona el intervalo $-z$ y $+z$. Por ejemplo, para una $N(\mu = 0, \sigma = 1)$:

```
1 import scipy.stats as st
2 z = st.norm.interval(0.95, 0, 1) # N(x=0,sigma=1) al 95%
3 print(z)
```

```
(-1.959963984540054, 1.959963984540054)
```

Que sería equivalente a:

```
1 import scipy.stats as st
2 confianza= 0.95
3 alfa = 1 - confianza
4 z = st.norm.ppf(1 - alfa / 2)
5 print(z)
```

```
1.959963984540054
```

Statsmodels

Por último, la librería **statsmodels** proporciona funciones para la estimación de muchos modelos estadísticos diferentes, así como para la realización de pruebas estadísticas y la exploración de datos estadísticos.

- Statsmodels (*statistical models*)
- **Principales utilidades:** Estimación de diferentes modelos estadísticos, realización de pruebas estadísticas y exploración de datos.

- Sitio: <https://www.statsmodels.org/>
- Documentación: www.statsmodels.org/stable/index.html#basic-documentation
- Licencia: open-source BSD-Modified

statsmodels.stats.weightstats

.DescrStatsW(s).zconfint_mean(alpha): devuelve los límites inferior y superior del intervalo de confianza. Con los siguientes argumentos:

- s: la muestra de estudio.
- alpha: El intervalo de confianza. Por defecto alfa es 0.05.

```

1 import statsmodels.stats.weightstats as sms
2 import numpy as np
3 np.random.seed(123)
4 s = np.random.normal(20, 2.5, 1000)
5 IC = sms.DescrStatsW(s).zconfint_mean() # alpha=0.05
6 print(IC)

```

```
(19.74594128016874, 20.05623803942731)
```

Para muestras inferiores ($n < 30$) se puede aplicar **.tconfint_mean(alpha)**.

Ejercicio 14

Ejercicio_06_01

Una entidad tiene previsto adquirir una cartera de 32.000 pólizas en una población donde no posee presencia.

En el fichero “ siniestros.xls” se recoge una muestra de 2.000 siniestros de una población de similar extensión y densidad. Se pide:

- Calcular el intervalo de confianza al 95 % para la media de siniestralidad total del negocio a adquirir.

8.2. Estadísticos de contraste

- A partir de la muestra de un estudio podemos calcular una serie de estadísticos (p.e. su media, su desviación típica, etc.). Un estadístico de contraste no es más que un estadístico más que nos ayuda a conocer la población. Un test estadístico es, por tanto, un indicador de decisión.
- Por ejemplo, si tenemos **dos conjuntos de datos independientes**, que supongamos han sido generados por un proceso gaussiano, podemos utilizar la **T-test** para decidir si las medias de los dos conjuntos de observaciones son significativamente diferentes:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{Sx_1x_2\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

- numerador: mayor cuanto mayor es la diferencia entre las medias de los dos grupos.
- denominador: mayor cuanto mayor sea la varianza, y menor cuanto mayor sea el tamaño de las muestras
- Si los dos grupos estuvieran extraídos de la misma población (o de poblaciones idénticas), esa diferencia observada entre las medias sería atribuible sólo al azar.

T-test con `scipy.stats`

Normalmente, la prueba o test t-student es utilizada para comparar medias entre dos poblaciones independientes. Aun así, posee diferentes usos:

- El **test de posición** para dos muestras, por el cual se comprueba si las medias de dos poblaciones distribuidas en forma normal son iguales.
- El **test de hipótesis nula** por el cual se demuestra que la diferencia entre dos respuestas medidas en las mismas unidades estadísticas es cero.
- El test para comprobar si la pendiente de una regr. lineal difiere estadísticamente de cero.

`scipy.stats.test_ind(x, y)`: Calcula el T-test para la media de dos muestras independientes. La salida se compone de:

- El **valor estadístico T**: es un número cuyo signo es proporcional a la diferencia entre los dos procesos aleatorios y cuya magnitud está relacionada con la importancia de esta diferencia.
- el **valor p**: la probabilidad de que ambos procesos sean idénticos. Si es cercano a 1, es casi seguro que los dos procesos son idénticos. Cuanto más se acerca a cero, más probable es que los procesos tengan diferentes medias.

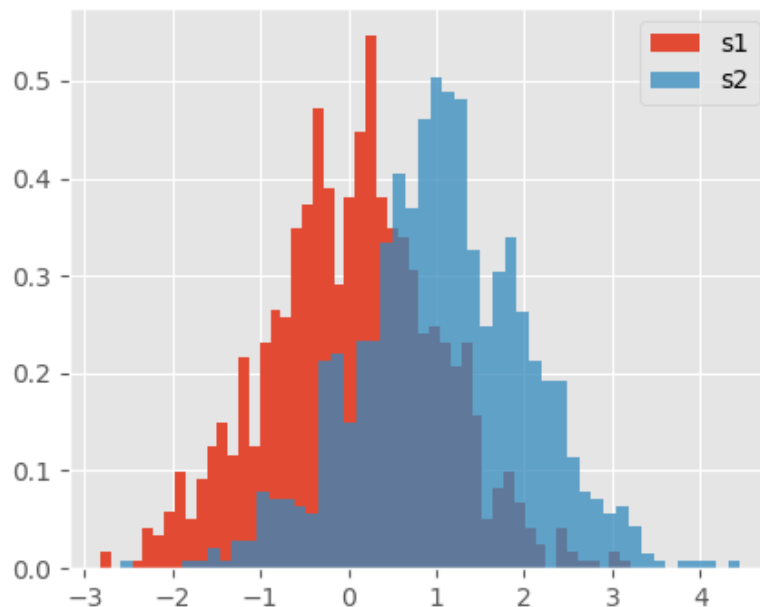
8.3. Contraste de hipótesis

Creamos dos conjunto de datos siguiendo una distribución normal (con distinta media), y posteriormente con `ttest_ind()` realizamos la prueba t de Student (o Test-T) para la media de dos muestras independientes.

Evidentemente el resultado es que ambas muestras son altamente dependientes o apareadas, y por tanto se puede asumir que las dos distribuciones poseen la misma varianza.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.stats as st
4
5 s1 = np.random.normal(0,1,size=1000)
6 s2 = np.random.normal(1,1,size=1000)
7
8 plt.hist(s1, bins=50, density=True, label="s1")
9 plt.hist(s2, bins=50, density=True, label="s2", alpha=0.75)
10 plt.legend()
11 plt.show()
12
13 print(st.ttest_ind(s1, s2))
```

```
statistic=-21.89615660325149
pvalue=6.452442645510087e-86
```



¿Qué probabilidad tenemos de haber obtenido estos valores por casualidad?

- **p-valor** es la probabilidad (determinada bajo un modelo estadístico específico) de obtener los datos observados.
- Esa probabilidad (valor entre 0 y 1) puede interpretarse como una medida del nivel de sorpresa que nos produce ese valor de t observado. El uso de $p=0.05$ es el más extendido.
- Un p -value bajo es bueno. Indican que los datos no ocurrieron por casualidad. Por tanto, rechazaremos el rango de valores de t que tienen una probabilidad muy pequeña de aparecer.

Compatibilidad

Aplicaciones: el p -valor es un **índice de compatibilidad** entre dos elementos: a) una hipótesis (llamada nula) y b) una evidencia empírica.

- Rechazamos la hipótesis nula (H_0) cuando observamos un valor del estadístico de contraste (t), lo suficientemente extremo como para que le corresponda una p menor que el valor que hemos establecido como criterio (por ejemplo 0.05).
- En caso de que ese índice de compatibilidad (el p -valor) sea bajo, habrá que elegir entre la hipótesis y la evidencia empírica (son incompatibles y no pueden coexistir) y, claro, nos quedaremos con la segunda y **rechazaremos la hipótesis nula**.
- En caso de que el índice de compatibilidad (el p -valor) sea alto, no habrá ninguna necesidad de elegir entre la hipótesis y la evidencia empírica y, por tanto, **aceptaremos la hipótesis nula**.
- El p -value **no** es la probabilidad de que la hipótesis nula sea falsa.

¿Cuándo es un p -valor alto o bajo? A dicho umbral se le llama **Nivel de significación**.

- Es práctica habitual situar el nivel de significación en 0.05 (Ronald Fisher “*Statistical Methods for Research Workers, 1925*”), aunque para muchos autores (Jacob Cohen ³) sea un estándar demasiado bajo de evidencia.
- Debido al “abuso” y a falsas interpretaciones en determinados sectores, la American Statistical Association (ASA) publicó en 2016 una declaración oficial (*The ASA’s Statement on p -Values*) sobre qué son (y qué no son) los p -valores, qué se puede afirmar con ellos y para qué se pueden utilizar.

³“Earth is round ($p<0.05$)”, Jacob Cohen

Contraste de hipótesis

Se denomina hipótesis nula (H_0) a la hipótesis que se desea contrastar (la hipótesis que mantendremos a no ser que los datos indiquen su falsedad)

Las hipótesis pueden clasificarse en dos grupos, según:

- Especifiquen un valor concreto o un intervalo para los parámetros del modelo.
Ejemplo: la media de una variable es 10
- Determinen el tipo de distribución de probabilidad que ha generado los datos.
Ejemplos: la distribución de probabilidad es la distribución normal o que dos conjuntos de datos se comportan bajo una misma distribución de probabilidad

Aunque la metodología para afrontar el contraste de hipótesis es similar en ambos casos, el primero es un ejercicio de estimación donde se estima un parámetro asociado a un intervalo de confianza, y el segundo el contraste de hipótesis se utiliza para validar el modelo estadístico del fenómeno aleatorio objeto de estudio.

8.4. Bootstrap

Remuestreo aleatorio:

- Cuando se asume normalidad, se simplifica enormemente la tarea de hallar los IC de los diferentes parámetros estimados, pero cuando no se ajusta a una distribución normal, hay que utilizar métodos empíricos o de remuestreo (resampling) para obtener de forma empírica los intervalos de confianza.
- El bootstrap es una técnica de remuestreo usado para estimar estadísticos en una población por muestreo con reemplazamiento, así como para construir intervalos de confianza o realizar contrastes de hipótesis.
- Partiendo de una muestra a estudiar, se remuestrea con reposiciones B veces, obteniendo B muestras diferentes* a la original. Para cada una de esas muestras, se calcula la media muestral y su distribución empírica, para posteriormente obtener los diferentes percentiles a usar como IC.
- Se puede usar para estimar los principales estadísticos como la media o la desviación estándar del siguiente modo:

```
1  for i in range(1000):  
2      muestra = data.sample(1000, replace=True)  
3      media = muestra.mean()  
4      print(media)
```

- Aunque en 1993, Efron fijó el número de reemplazamientos entre 25 y 200, con la potencia de los actuales ordenadores podemos incrementar a mínimo de 1.000 observaciones:

Ejemplo de remuestreo o bootstrap:

```
1 import pandas as pd
2 import numpy as np
3
4 data = pd.Series([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
5 b = 10 #1000
6 s = 5 #1000
7 oob = pd.Series(np.empty(b))
8 for i in range(b):
9     sample = data.sample(s, replace=True)
10    statistic = sample.mean()
11    oob[i] = statistic
12
13 # Intervalo de confianza al 95%
14 print(oob.mean(), oob.quantile(0.025), oob.quantile(0.975))
```

- La técnica bootstrap puede adaptarse a cada situación en función de la observación de los datos disponibles y el tipo de negocio a estudiar.
- Así, para un modelo lineal (Clásico o Generalizado) para la estimación de la provisión IBNR (como Chain-Ladder) se puede optar entre dos enfoques:
 - Bootstrap vinculado: el remuestreo se realiza directamente desde las observaciones en un modelo de regresión.
 - Bootstrap de residuos: el remuestreo se aplica a los residuos del modelo (residuos estandarizados Pearson o Chi).
- Para la implementación de un modelo Bootstrap en el cálculo de reservas es necesario elegir un modelo, definir un residuo adecuado y los parámetros estimados.
- La librería **scikit-learn** enfocada al aprendizaje automático o *machine learning* posee la función **sklearn.utils.resample()** que simplifica la tarea de realizar reemplazamientos.

Ejercicio 15

Ejercicio_06_02

En el anexo XVII del Reglamento Delegado, figuran los requisitos y demostraciones que deben realizarse para la aprobación de parámetros específicos de empresa. Entre ellos, se debe demostrar que la siniestralidad agregada sigue una distribución logarítmica normal $LN(\mu, \sigma)$.

Se pide, partiendo de los siniestros recogidos en el fichero “*siniestralidad_agregada.csv*”:

- Calcular la media y varianza valiéndose de la técnica de remuestreo (bootstrap) obteniendo 10000 \bar{s} .
- Calcular los principales percentiles (.25, .5, 0.75) de la distribución teórica y compararlos con los datos de la distribución empírica de los siniestros.

AYUDA: siendo s la siniestralidad total y z la variable aleatoria normal, entonces $s = e^z$ y $z = \log(s)$ con media y varianza:

$$E(s) = e^{\mu + \frac{\sigma^2}{2}}$$

$$Var(s) = (e^{\sigma^2} - 1)e^{2\mu + \sigma^2}$$

9. Modelos estadísticos

Todos los modelos se equivocan, pero algunos son útiles. (George Box, estadístico)

Adaptar las distribuciones a los datos es una tarea muy común en estadística. Consiste en elegir la **distribución de probabilidad** que mejor se adapte a la variable aleatoria, así como encontrar estimaciones de parámetros para esa distribución. Es decir, consiste en encontrar la función matemática que mejor represente o explique el comportamiento de la variable aleatoria a modelizar.

Un **modelo estadístico** es una forma simplificada de aproximarse a la realidad.

Los modelos estadísticos nos pueden servir para:

- Revelar hechos y tendencias sobre una población
- Predecir el comportamiento futuro
- Validación de las hipótesis en que se basan los **modelos internos** o **parámetros específicos**.

Esto requiere del uso del **juicio experto**, ya que generalmente es necesario un proceso iterativo de varios pasos:

- Elección de un modelo o distribución
- Estimar los parámetros de la distribución candidata
- Evaluar la calidad del ajuste (bondad del ajuste)

*”Las entidades aseguradoras calcularán el capital de solvencia obligatorio directamente a partir de la **distribución de probabilidad prevista** generada por su modelo interno, utilizando la medida del valor en riesgo establecida en el artículo 74.1 de la Ley 20/2015, de 14 de julio.” (art. 85. RDOSSEAR).*

- No se establece ningún método concreto para calcular la distribución de probabilidad prevista para los modelos internos (art. 121.4 Directiva + art. 84. RDOSSEAR), sino usar prácticas de mercado generalmente aceptadas.
- En el caso de **parámetros propios de empresa**, estos deben ser calculados con los métodos estandarizados usados para el mercado, incluyendo los supuestos de distribución probabilística. Así, para el cálculo de la **desviación típica** específica de la empresa, se usará el MSEPM (Error Cuadrático Medio de Predicción) de una **distribución ajustada a una lognormal**. (anexo XVII Reglamento Delegado).

Modelos de ajustes a los datos

Muchos procedimientos estadísticos suponen que los datos siguen algún tipo de **modelo matemático que se define mediante una ecuación**, donde al desconocerse alguno de sus parámetros, pueden estimarse a partir de la información disponible.

Existen principalmente dos procedimientos para estimar los coeficientes de un modelo de regresión, o para estimar los parámetros de una distribución de probabilidad:

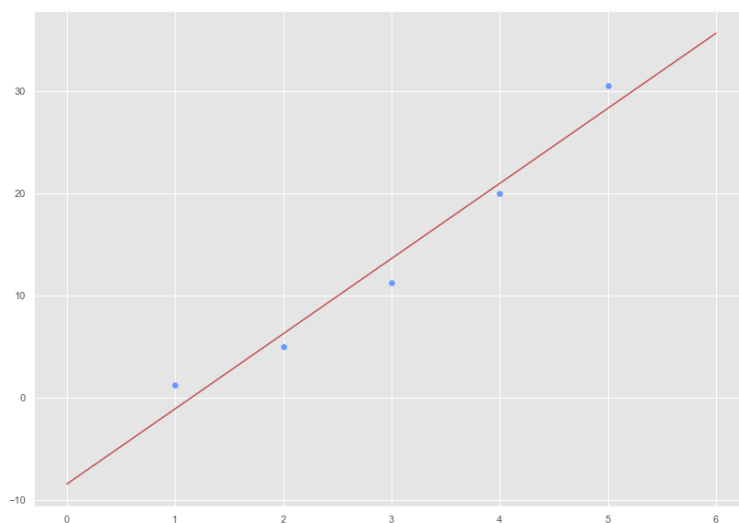
- **Mínimos cuadrados ordinarios (MCO)**, *ordinary least squares* (OLS) o mínimos cuadrados lineales son los nombres que recibe el método para encontrar los parámetros poblacionales en un modelo de regresión lineal obtenidos minimizando los residuos observados (la suma de los cuadrados de las diferencias entre la variable dependiente observada).
- **Estimación por máxima verosimilitud (EMV)**, *maximum likelihood estimation* (MLE), la cual requiere maximizar la función de verosimilitud, o función de probabilidad conjunta de los valores muestrales.

Bajo la hipótesis de distribución normal, las estimaciones por ambos criterios coinciden.

9.1. Regresión lineal

Gráfico de puntos y recta estimada de regresión por mínimos cuadráticos:

	x	y
2014	1	1.25
2015	2	5
2016	3	11.25
2017	4	20
2018	5	30.5




```

1 df = pd.DataFrame(index=range(2014,2019))
2 df['x'] = [1, 2, 3, 4, 5]
3 df['y'] = [1.25, 5, 11.25, 20, 30.5]

```

En el modelo de regresión lineal, conocido también como modelo de regresión simple, o recta poblacional, es un modelo matemático usado para aproximar la relación de dependencia entre una variable dependiente y, las variables independientes x y un término aleatorio e:

$$y = \beta_1 + \beta_2 x + e$$

Siendo:

- **y**: la variable endógena o dependiente (variable explicada o regresando)
- **x**: la variable exógena o independiente (variable explicativa, regresor, covariable o variable de control)
- **e**: el error o perturbación aleatoria. Lo que no explica el modelo de ajuste.
- β_1 : es la ordenada al origen (indica el valor de y cuando x=0).
- β_2 es la pendiente de la recta (cuanto cambia y por cada incremento de x).

El objetivo principal del modelo de regresión es β_1 y β_2 (parámetros fijos y desconocidos) a partir de una muestra dada.

En base a una muestra de tamaño n es posible estimar los parámetros del modelo. Si una recta pasa exactamente por los puntos, el error será cero. El programa de minimización es el siguiente:

$$\min \sum_{i=1}^n (y_i - (\beta_0 + \beta_1 x_i))^2$$

En Python existen funciones específicas cuya finalidad es ajustar los datos tanto a una recta como a una distribución de probabilidad.

Así, para determinar la distribución que sigue un vector de datos en Python contamos con:

- Scipy: **scipy.stats**
- Statsmodels: **statsmodels.api**

Scipy

`SciPy.stats.linregress(x, y)` donde la salida devuelve los siguientes parámetros:

- **slope**: pendiente de la línea de regresión
- **intercept**: intercepción de la línea de regresión
- **r_value**: coeficiente de correlación
- **p_value**: p-value cuya hipótesis nula es que la pendiente es cero.
- **std_err**: Error estándar de la estimación

```
1 import scipy.stats as st
2 x = (1., 2., 3., 4., 5.)
3 y = (1.25, 5, 11.25, 20, 30.5)
4 print(st.linregress(x,y))
```

```
LinregressResult(slope=7.35, intercept=-8.45, rvalue=0.983437109,
pvalue=0.00255244, stderr=0.782091213)
```

Statsmodels

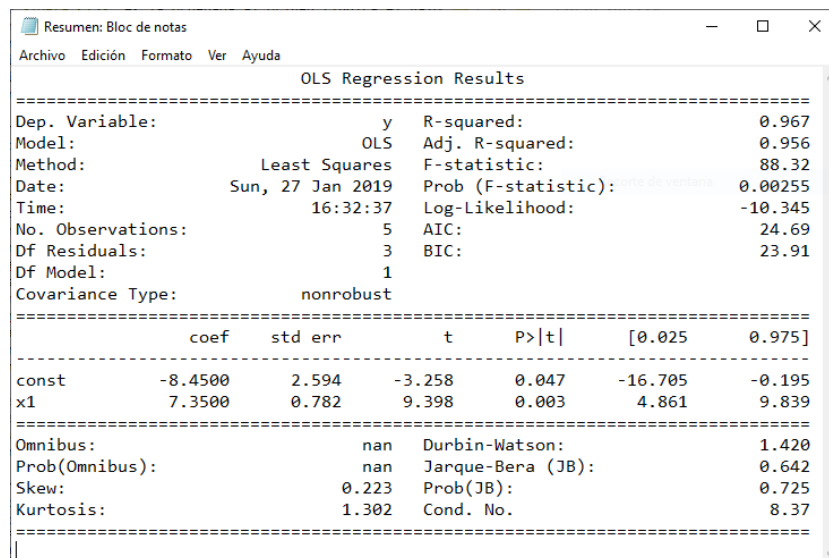
`statsmodels.api` posee la función `.OLS(x, y)` para calcular el MCO (ordinary least squares model).

- `results = sm.OLS(x, y).fit()` Devuelve, entre otros, los siguientes estadísticos:
- `results.params` devuelve los parámetros estimados `[-8.45 7.35]`
- `results.tvalues` devuelve los valores de t `[-3.25763903 9.3978808]`
- `results.mse_resid`
- `results.rsquared`
- `results.fvalue`
- `results.aic`
- `results.bic`
- `results.summary()`: Engloba los anteriores. Útil para exportarlo a txt o pdf.

<https://www.statsmodels.org/dev/regression.html>

Ejemplo de MCO con `statsmodels.api.OLS(x, y)` guardando la salida en un fichero TXT:

```
1 import numpy as np
2 import statsmodels.api as sm
3
4 x = [1., 2., 3., 4., 5.]
5 y = [1.25, 5, 11.25, 20, 30.5]
6
7 x = sm.add_constant(x)
8 model = sm.OLS(y, x)
9 results = model.fit()
10
11 with open("Resumen.txt", "w") as text_file:
12     print(results.summary(), file=text_file)
```



Resumen: Bloc de notas

Archivo Edición Formato Ver Ayuda

OLS Regression Results

Dep. Variable:	y	R-squared:	0.967
Model:	OLS	Adj. R-squared:	0.956
Method:	Least Squares	F-statistic:	88.32
Date:	Sun, 27 Jan 2019	Prob (F-statistic):	0.00255
Time:	16:32:37	Log-Likelihood:	-10.345
No. Observations:	5	AIC:	24.69
Df Residuals:	3	BIC:	23.91
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P> t	[0.025	0.975]
const	-8.4500	2.594	-3.258	0.047	-16.705	-0.195
x1	7.3500	0.782	9.398	0.003	4.861	9.839

Omnibus:	nan	Durbin-Watson:	1.420
Prob(Omnibus):	nan	Jarque-Bera (JB):	0.642
Skew:	0.223	Prob(JB):	0.725
Kurtosis:	1.302	Cond. No.	8.37

Prueba de t de student

`statsmodels` también posee otras funciones útiles. Por ejemplo, la prueba de t de Student:

```
results = st.t_test([1, 0], use_t=False)
```

- `use_t = False` : p-values se basan en la distribución normal (z)
- `use_t = True` : p-values se basan en la distribución t (t)

Devolviendo los siguientes resultados:

- `results.mse_resid`:
- `results.rsquared`:

- `results.fvalue`:
- `results.aic`:
- `results.summary()` Resumen de los anteriores

9.2. Regresiones multiples

`statsmodels` posee la clase `statsmodels.formula.api` con la función `ols`, la cual también puede ser utilizada para ajustar regresiones lineales múltiples:

- `ols(formula, datos).fit()`

Ejemplo de estimación con regresión múltiple de la tasa de empleo en Estados Unidos utilizando el dataset “*U.S. economic time series used by Schotman & Dijk, 1988*”.

```

1 from statsmodels.formula.api import ols
2 import pandas as pd
3
4 datos = pd.read_csv("desempleo.csv", sep=";", decimal=",")
5 formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
6 results = ols(formula, datos).fit()
7 hipotesis = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
8 t_test = results.t_test(hipotesis)
9 print(t_test)

```

9.3. Modelo lineal generalizado (GLM)

- Scikit learn tiene una función llamada Generalized Linear Model, aunque es simplemente un modelo lineal, sin función de enlace, por lo que propiamente no es un GLM.
- En cambio, **Statsmodel** si dispone de una función GLM, la cual realiza dos ajustes (uno con interacción y otro sin ella):

```

1 import statsmodels.api as sm
2 import statsmodels.formula.api as smf
3 import statsmodels.genmod.families.links as llink
4
5 formula='pf ~ cost + output + lf'
6
7 mod = smf.glm(formula=formula, data=data, family=sm.families.
8               Poisson(link=llink.log))
9 res = mod.fit()
10 print(res.summary())

```

En el ejemplo 10.2 se realiza una estimación del precio de la vivienda en Boston utilizando para ello un enfoque basado en GLM, utilizando el RMSE (Raíz del error cuadrático medio) para validar las diferentes regresiones. Previamente se realiza un estudio de las correlaciones de todas las variables, una selección de variables dependientes en base a su correlación y un estudio uni-variable de la base de datos.

Ejercicio 16

Ejercicio_07_01

En el submódulo de riesgo de prima y reservas, determinados parámetros generales pueden sustituirse por parámetros específicos de la empresa (art. 218 del Reglamento Delegado), cumpliendo entre otros aspectos, las hipótesis previstas en el ANEXO XVII:

Los datos se ajustarán a las siguientes hipótesis:

i. la siniestralidad agregada para un determinado segmento y año de accidente es linealmente proporcional con respecto a las primas devengadas en un determinado año de accidente;

Se pide, para los siguientes datos ficticios, validar el cumplimiento de la anterior hipótesis:

Siniestralidad	461210	518830	574390	690200	706580	740550	763660	804950
Primas	482880	546620	591390	690240	707440	751330	791320	848870

9.4. Máxima verosimilitud

- La estimación por máxima verosimilitud (EMV o MLE) es un método habitual para ajustar un modelo y estimar sus parámetros.
- Si suponemos que x_1, x_2, \dots, x_n son fijos y θ puede variar libremente, la estimación máxima verosimilitud de los parámetros θ son los parámetros que maximizan esta función con valores x fijos.

$$f(x, \theta) = \prod_{i=1}^N f(x_i, \theta) \quad (1)$$

9.5. Modelos de Ajuste a una distribución

- Para ajustar datos a una distribución, es decir, determinar la distribución que sigue un vector de datos, todas las distribuciones continuas de **SciPy.Stats** disponen la función **.fit()**, que nos permite estimar parámetros partiendo de los datos disponibles.
- El método utilizado es el de máxima verosimilitud.
- El output de fit dependerá de los parámetros de la distribución a ajustar. Por ejemplo, en las siguientes dos distribuciones el output es distinto:
 - Normal: **st.norm.fit(data)**, devuelve media y std
 - Lognormal: **st.lognorm.fit(data)**, devuelve sigma, mu y std.
- Importante: **.fit()** no da ningún valor sobre la **bondad del ajuste**. Para ello veremos otras funciones.

9.6. Ejercicio 17

Ejercicio_07_02

Con el fichero “siniestros.xls” del Ejercicio 06_01, se pide hallar la media y desviación estándar de los siniestros estimando que se distribuyen como una distribución normal.

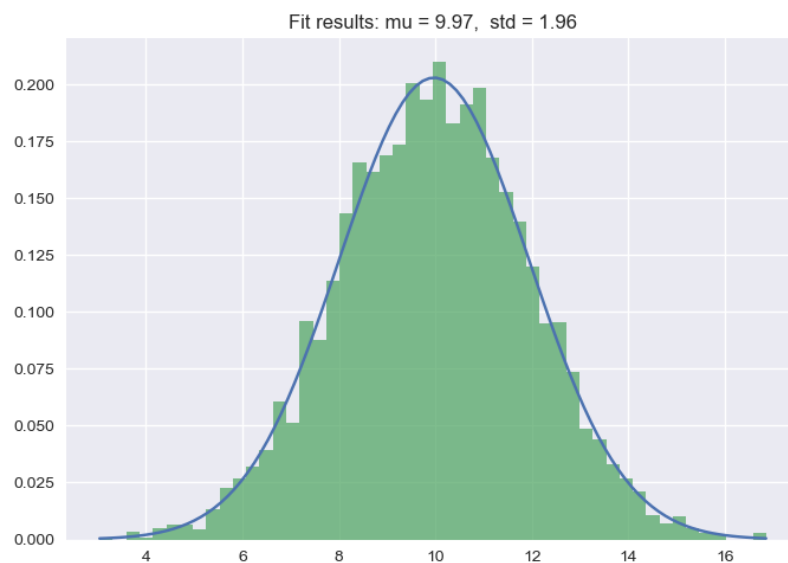
```
1 import pandas as pd
2 import scipy.stats as st
3
4 df = pd.read_excel("siniestros.xls")
5 media, desviacion = st.norm.fit(df['Importe'])
6
7 print(media)
8 print(desviacion)
```

20.0

4.50

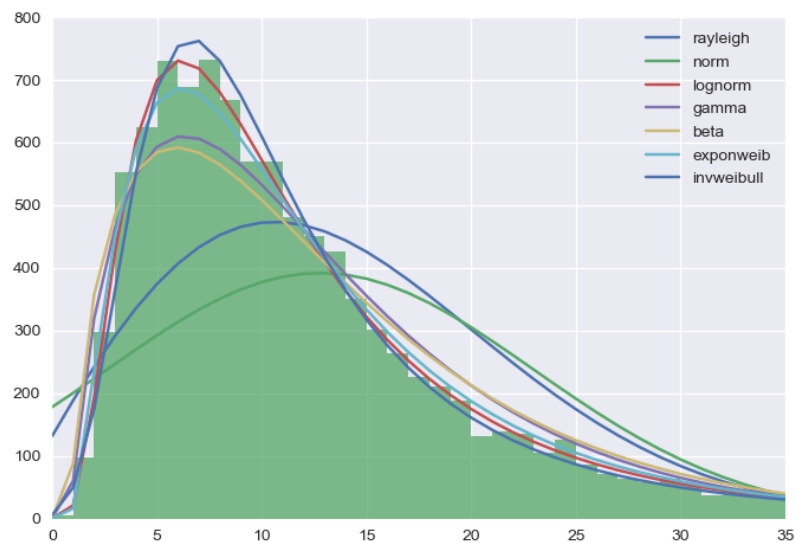
En el siguiente ejemplo, se crean 5.000 datos aleatorios siguiendo una distribución normal, y posteriormente se comprueba cual sería su media y varianza ajustándose a la normal (a mayor número de datos generados mayor precisión). Por último, se dibujan los datos para su visualización.

```
1 import numpy as np
2 import scipy.stats as st
3 import matplotlib.pyplot as plt
4
5 data = np.random.normal(10, 2, 5000)
6 mu, std = st.norm.fit(data)
7 x = np.linspace(data.min(), data.max(), 100)
8 y = st.norm.pdf(x, mu, std)
9
10 plt.style.use('seaborn')
11 plt.plot(x, y)
12 plt.title("Fit results: mu = %.2f, std = %.2f" % (mu, std))
13 plt.hist(data, bins=50, density=True, alpha=0.75)
14 plt.show()
```



Para un mismo set de datos puede realizarse para tantas distribuciones como queramos, y elegir la que mejor se adapte:

```
1 size = 10000
2 x = np.arange(size)
3 y = np.random.lognormal(np.log(10), np.log(2), size)
4
5 distrs = ['rayleigh', 'norm', 'lognorm', 'gamma', 'beta', '
6   exponweib', 'invweibull']
7 for i in distrs:
8     dist = getattr(st, i)
9     param = dist.fit(y)
10    pdf_fitted = dist.pdf(x, *param[:-2], loc=param[-2], scale=
11      param[-1]) * size
12    plt.plot(pdf_fitted, label=i)
13    plt.xlim(0,25)
14 plt.hist(y, bins=range(50), alpha=0.75)
15 plt.show()
```



9.7. Bondad del ajuste: Calidad estadística

¿Qué distribución elegir? ¿Cómo de bueno es el ajuste de cada distribución?

La **bondad de ajuste** de un modelo estadístico describe cómo de bien se ajusta un conjunto de observaciones. Las medidas de bondad resumen la discrepancia entre los valores observados y los valores esperados en el modelo de estudio.

Para poder concluir si una distribución dada se ajusta a un conjunto de datos, y poder dar un valor cuantitativo a dicho ajuste, se suelen utilizar principalmente los siguientes test:

- Test de Kolmogorov-Smirnov (K-S)
- Test de Chi cuadrado o X^2 de Pearson (Chisq)
- Test de Anderson-Darling (AD), Shapiro-Wilk, Jarque-Bera
- Criterio de Información de Akaike (AIC), BIC ...

9.7.1. Test de Kolmogorov-Smirnov

- El test de **Kolmogorov-Smirnov** o K-S permite medir el grado de concordancia existente entre la distribución de un conjunto de datos y una distribución teórica específica.
- Si el valor del test K-S es pequeño o el p-value es alto, no podemos rechazar la hipótesis de que las distribuciones de las dos muestras son las mismas.

En **Scipy.Stats** disponemos de la función **st.kstest()** con los siguientes argumentos:

- **x**: la muestra a validar
- **cdf**: el nombre de la distribución en formato string
- **args**: los parámetros de la distribución (si los disponemos)

Y que devuelven los siguientes 2 valores:

- **statistic**: El valor del estadístico K-S
- **pvalue**

Ejemplo de calculo del estadistico K-S con el fichero "siniestros.xls" del Ejercicio 07_02:

```
1 import pandas as pd
2 import scipy.stats as st
3
4 df = pd.read_excel("siniestros.xls")
5 mu, sigma = st.norm.fit(df['Importe'])
6 kstest = st.kstest(df['Importe'], cdf='norm', args=(mu, sigma))
7 print(mu, sigma)
8 print(kstest)
```

```
20.0 4.50
```

```
KstestResult(statistic=0.048, pvalue=0.00019)
```

9.7.2. Test de Chi cuadrado o X^2 de Pearson

- La Chi-cuadrada o X^2 permite conocer las diferencias entre una distribución de **frecuencias observadas** y otra **esperada o teórica**.
- Se utiliza para realizar contrastes de bondad de ajuste, de homogeneidad y de independencia cuando los tamaños muestrales son grandes.
- Cuanto mayor sea el valor de X^2 , la hipótesis se encuentra más alejada del valor correcto. Por tanto, cuanto más se aproxima a cero el valor de chi-cuadrado, más ajustadas están ambas distribuciones.

En scipy disponemos de la función `st.chisquare(f_obs, f_exp, ddof=0, axis=0)`

```
1 import scipy.stats as st
2 a = [16, 18, 16, 14, 12, 12]
3 b = [16, 16, 16, 16, 16, 8]
4 x2_test = st.chisquare(f_obs=a, f_exp=b)
5 print(x2_test)
```

```
Power_divergenceResult(statistic=3.5, pvalue=0.6233876277495822)
```

9.7.3. Test de Anderson-Darling

- El test **Anderson-Darling** prueba la hipótesis nula de que una muestra se extrae de una población que sigue una distribución determinada.
- Este test funciona para distribuciones normales, exponenciales, logísticas o Gumbel (Extreme Value Type I): *norm*, *expon*, *logistic*, *gumbel*, *gumbel_l*, *gumbel_r*

En SciPy se dispone de la función `st.anderson(x, dist='norm')`

```
1 import scipy.stats as st
2 a = [10, 12, 16, 14, 12, 10]
3 ad_test = st.anderson(a, 'norm')
4 print(ad_test)
```

```
AndersonResult(statistic=0.31071216578486727, critical_values=
array([0.592, 0.675, 0.809, 0.944, 1.123]), significance_level=
array([15. , 10. , 5. , 2.5, 1. ]))
```

Si el estadístico es mayor que estos valores críticos para el nivel de significación correspondiente, se puede rechazar la hipótesis nula de que los datos proceden de la distribución elegida.

9.7.4. Test de Shapiro-Wilk

- La prueba **Shapiro-Wilk** prueba la hipótesis nula de que los datos se extrajeron de una distribución normal.
- **st.shapiro(x)** realiza la prueba de normalidad de Shapiro-Wilk. Devuelve:
 - (valor del estadístico, p-value)
- El test de Shapiro-Wilk posee mayor potencia que el resto de test de ajuste a una normal, especialmente cuando disponemos de pocos datos.

```
1 import scipy.stats as st
2 import numpy as np
3 x = np.random.normal(0, 1, size=1000)
4 w_test = st.shapiro(x)
5 print(w_test)
```

```
(0.9949701428413391, 2.7718315322999842e-06)
```

Con este p-valor no rechazamos la hipótesis nula de normalidad. Esto es, en la muestra hay evidencia de que proceda de una distribución normal.

9.7.5. Test de Jarque-Bera

- La prueba de **Jarque-Bera** comprueba si los datos de la muestra tienen la asimetría y la curtosis correspondientes a una **distribución normal**.
- **st.jarque_bera(x)** la prueba de bondad de ajuste de Jarque-Bera con los datos de la muestra. Devuelve:
 - (valor del estadístico, p-value)
- Esta prueba sólo funciona para un número suficiente de muestras de datos (¡2000), ya que la estadística de la prueba tiene una distribución asintótica de Chi-cuadrado con 2 grados de libertad.

Ejemplo de realizar el test J-B a dos distribuciones (la primera Normal y la segunda Rayleigh)

```
1 import scipy.stats as st
2 import numpy as np
3 np.random.seed(100)
4 x = np.random.normal(0, 1, size=10000)
5 y = np.random.rayleigh(1, size=10000)
6 print(st.jarque_bera(x))
7 print(st.jarque_bera(y))
```

```
(0.720948044268747, 0.6973456904203446)
```

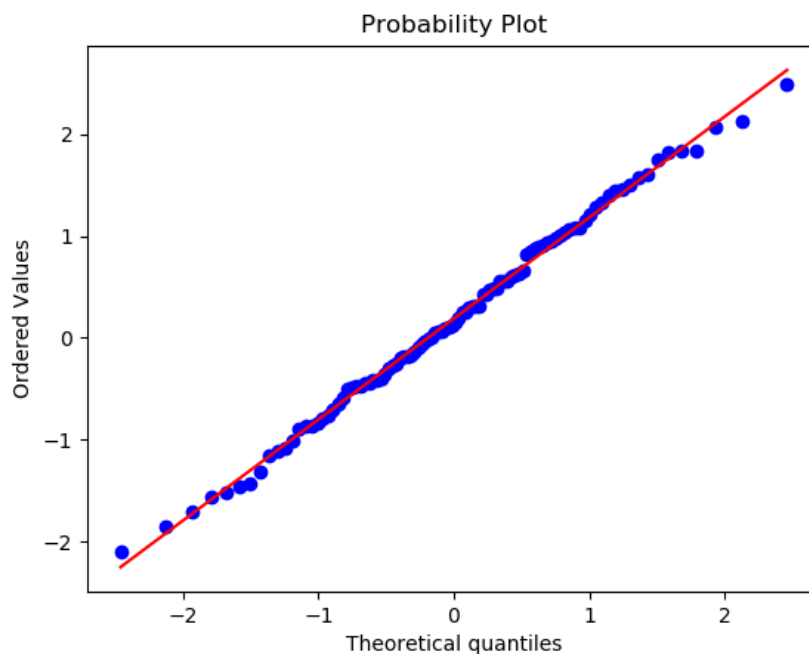
```
(731.844781033357, 0.0)
```

9.8. Gráfica Q-Q

- Una vez seleccionada nuestra distribución candidata a validar, para reforzar nuestra hipótesis y obtener una mayor fiabilidad se puede mostrar la gráfica conocida como **quantil-quantil** (*quantile-quantile plot*, gráfico Q-Q o qqplot).
- Consiste en una comparación visual entre las funciones de distribución teórica (cuantiles teóricos) y los datos extraídos de la muestra ordenados.
- Si los datos están visualmente superpuestos, se puede concluir que ambas distribuciones se comportan de igual manera.
- Tanto `scipy.stats` como `statsmodels` disponen de funciones para realizar, junto con `matplotlib`, los gráficos Q-Q:
`scipy.stats.probplot(sample, dist=st.distribucion, plot=plt)`
`statsmodels.api.qqplot(sample, dist='norm', line='s')`

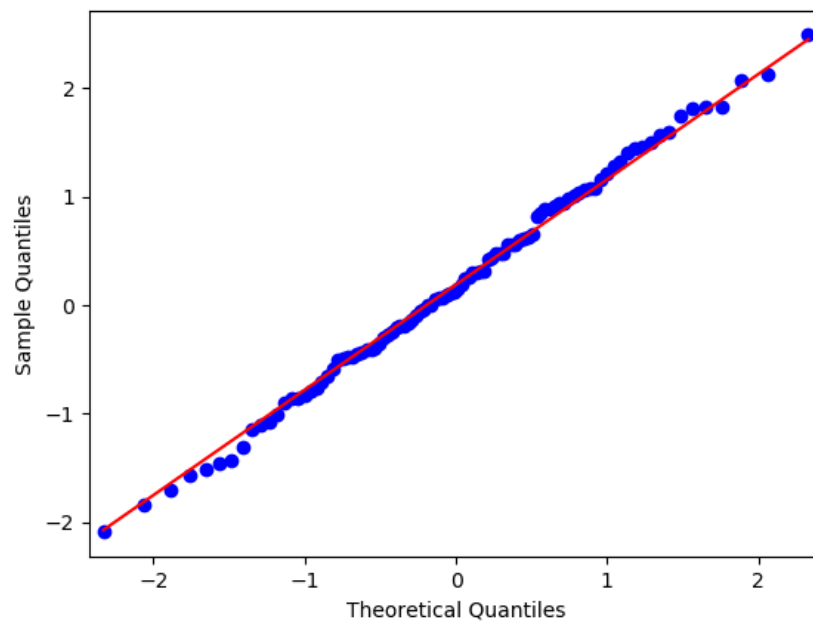
Con `scipy.stats` y la función `.probplot()`:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.stats as st
4
5 np.random.seed(999)
6 s = np.random.normal(0, 1, 100)
7 st.probplot(s, dist=st.norm, plot=plt)
8 plt.show()
```



Con `statsmodels.api` y la función `.qqplot()`:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import statsmodels.api as sms
4
5 np.random.seed(999)
6 s = np.random.normal(0, 1, 100)
7 sms.qqplot(s, dist='norm', line='s')
8 plt.show()
```



9.9. Ejercicio 18

Ejercicio_07_03

Con el df llamado 'danish' (datos de incendios en Dinamarca mayores de 1 DKK entre 1980 a 1990 en millones de coronas danesas) accesible del siguiente modo:

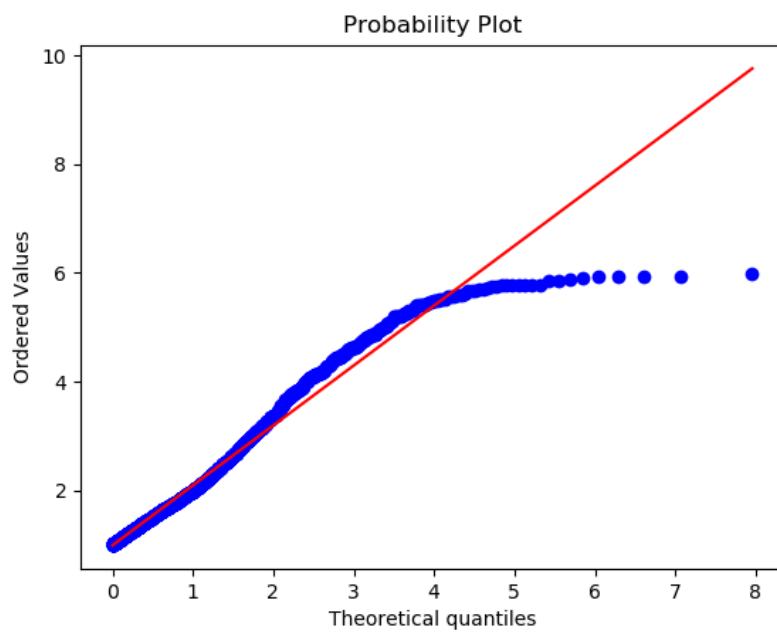
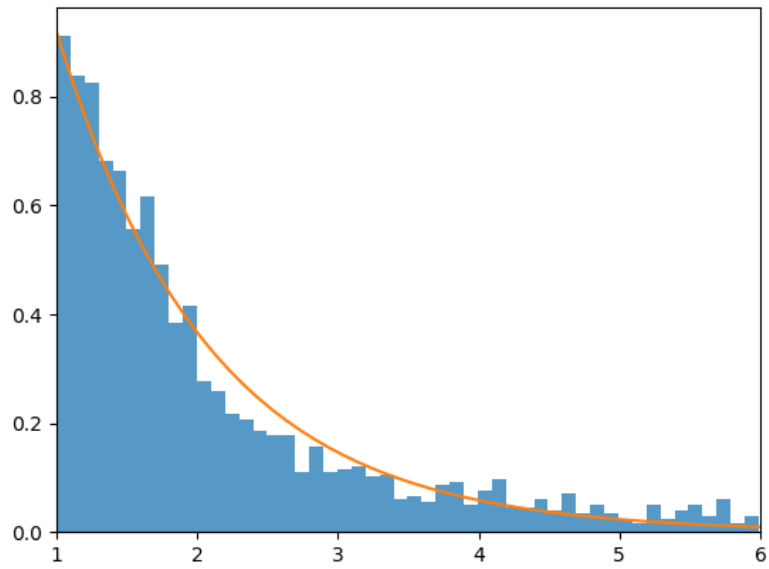
```
1 from pydataset import data
2 danish = data('danish')
```

Para los importes de los siniestros entre los 1 DKK y 6 DKK, se pide:

- Realizar el histograma y ajuste a una distribución exponencial.
- Calcular la bondad del ajuste basado en el test de Kolmogorov-Smirnov
- Realizar la gráfica QQ-Plot frente a una distribución exponencial.

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import scipy.stats as st
4 import numpy as np
5 from pydataset import data
6
7 danish = data('danish')
8 datos = danish[danish.x < 6].copy()
9 datos = pd.Series(datos.x)
10
11 loc, scale = st.expon.fit(datos)
12 exponencial = st.expon(loc=loc, scale=scale)
```

```
1 plt.hist(datos, bins=50, density=True, alpha=0.75)
2 x = np.linspace(datos.min(), datos.max(), 50)
3 y = st.expon.pdf(x, loc=loc, scale=scale)
4 plt.plot(x, y)
5 plt.xlim(1,6)
6 plt.show()
7
8 kstest = st.kstest(datos, cdf='expon', args=(loc, scale))
9 print(kstest)
10
11 st.probplot(datos, dist=st.expon, plot=plt)
12 plt.show()
```



Anexo

Versión utilizada de statsmodels: 0.11.1

10. Modelización de seguros de Vida

Python puede ser una herramienta muy potente para el desarrollo de **herramientas de modelización** para el negocio de Vida, o incluso, como herramienta DFA (Dynamic Financial Analysis). Librerías como **Pandas** y **NumPy** ofrecen facilidad y rapidez en ámbitos como:

- Acceso a bases de datos (SQL, texto plano, etc..)
- Manejo de campos de fechas
- Generación series temporales
- Cálculo de tipos de interés y descuento de flujos
- Uso de interpolaciones en la elaboración de hipótesis
- IFRS17: cálculos en modo “*locked-in assumptions*”
- Generación de escenarios económicos
- Manejo de ficheros temporales (cálculos en esclavos: master-worker) y computación paralela (cloud/grid computing)

Pandas

Aunque el uso **Pandas** reste velocidad en la ejecución, si es una librería muy útil para implementar módulos de tarificación, *profit-testing* e incluso para cálculo de reservas.

- El index del dataframe puede ser utilizado como el periodo de tiempo de la proyección. Del mismo modo, la creación y uso de series temporales es sumamente fácil.
- Si en cada iteración se genera un dataframe, este podrá ser anexado a un dataframe total que podrá ser exportando en csv o guardado en un base de datos externa (txt, sql, access, etc...)
- El output final (el df total) puede agruparse además que por el campo ['t'] por más campos (como por ejemplo: por póliza o producto y por año de emisión) con un simple groupby.
- El límite del dataframe está en la memoria RAM del PC. Si se tiene previsto superar dicha capacidad (el comando `df.info()` informa de la memoria de cada dataframe), pueden crearse ficheros temporales (consultar librería *tempfile*).
- Python ofrece una gran escalabilidad, ya que el código de Python puede ser ejecutado en servidores en la nube, así como en servidores SQL (desde su versión SQL 2016).
- Existen diferentes recursos si queremos reducir el tiempo de cálculo en Python (Cython, Numba, computación en paralelo o distribuida, clusters, multiproceso...)

10.1. Manejo de fechas y edades

El correcto manejo de las fechas es crítico en cualquier cálculo actuarial.

Lo más recomendable cuando utilizamos fechas en Pandas es formatearlas con la instrucción `pd.to_datetime(date, format='%Y-%m-%d')`.

- **date:** en formato *int*, *float*, *str*, *datetime* o *list*.
(en algún caso, puede resultar útil forzarlo a ser *str* con un `.astype(str)` y aplicarle el formato correcto)
- **format:** Formato del dato origen strftime, ej “%d %m %Y”.
Los formatos posibles (%d,%m,%Y,%y,%j...) pueden consultarse en: <https://docs.python.org/2/library/time.html>
- **errors:** En caso de error, que debe hacer **pandas** =‘ignore’ (ignorar) o =‘coerce’ (forzar).
- Ejemplo de lectura de fecha con formato DDMMYYYY:

```
1 data['fnac'] =  
2     pd.to_datetime(data['fnac'].astype(str), format='%d%m%  
    Y')
```

- Solo en el caso de que la fecha en origen tenga el formato 'YYYY-MM-DD', podemos formatearla con un simple `.astype(datetime64[D])`:

```
1 data['fnac'] = data['fnac'].astype('datetime64[D]')
```

apply y lambda

- El uso de **apply** y **lambda** se hace necesario a la hora de aplicar funciones previamente creadas o simplemente aplicar condiciones:

```
1 df['qx'] = df['edad'].apply(lambda x: capital_fall(x))
```

- Ejemplo de apply y lambda aplicando una condición **if**:

```
1 df['qx'] = df['edad'].apply(lambda x: 200 if x<=65 else 100)
```

- En el caso que nuestra función requiera hacer uso de 2 o más variables, debemos aplicar la función **lambda x:** al df, por ejemplo:

```
1 df['qx'] = df.apply(lambda x: PER2012(x['EDAD'],x['ANNAC']), axis  
    =1)
```

- Si queremos añadir limpieza al código, y el nombre de las variables no incluyen espacios, podemos utilizar esta forma:

```
1 df['qx'] = df.apply(lambda x: PER2012(x.EDAD, x.ANNAC), axis=1)
```

datetime

- Pandas incluye de serie las principales funciones de la librería datetime con el alias **dt**.
- Para utilizarla podemos usar sus funciones de varias formas (incluso sin escribir el alias dt). Por ejemplo:

```
1 import pandas as pd
2
3 fnac = '20/03/1975'
4 fnac = pd.to_datetime(fnac, format='%d/%m/%Y')
5 annac = fnac.year
6 print(annac)
```

1975

10.2. Cálculo de edades

Ejemplo con la librería datetime para calcular la **edad actuarial**:

- **np rint()** redondea al entero más próximo
- **np.trunc()** devuelve el valor sin decimales

```
1 import numpy as np
2 import pandas as pd
3
4 def EdAct(fechanac, fval):
5     fechanac = pd.to_datetime(str(fechanac))
6     fval      = pd.to_datetime(str(fval))
7     return np rint((fval - fechanac).days / 365.25).astype
8         (int)
9
10 print(EdAct('1942-06-30', '2019-12-31'))
```

78

Ejemplo para calcular la **edad real** o **edad cumplida**: En este caso, se aprovecha el valor de la expresión lógica <

*En muchos lenguajes como Python, **True** tiene un valor de 1 y **False** de 0.*

```
1 import pandas as pd
2
3 def EdReal(fechanac, fval):
4     fechanac = pd.to_datetime(str(fechanac))
5     fval = pd.to_datetime(str(fval))
6     return fval.year - fechanac.year - ((fval.month, fval.day) < (
7         fechanac.month, fechanac.day))
8
9 print(EdReal('1942-06-30', '2019-12-31'))
```

77

Ejemplo de cálculo de la **fecha de jubilación**: Haciendo uso de la función **date(año, mes, día)** podemos calcular la fecha en alcanzar una determinada edad:

```
1 import pandas as pd
2 from datetime import date
3
4 def DiaJub(fnac):
5     fnac = pd.to_datetime(str(fnac))
6     try:
7         return date(fnac.year + 65, fnac.month, fnac.day)
8     except:
9         return date(fnac.year + 65, fnac.month, fnac.day - 1)
10
11 print(DiaJub('1979-03-19'))
```

2044-03-19

Tablas generacionales

Ejemplo de implementación con pandas de la tabla de mortalidad permf2012c:

```
1 import numpy as np
2 import pandas as pd
3 permf2012c = pd.read_csv('permf2012c.csv', sep=';', decimal=',')
4
5 def per2012c(sex, anno_nac, edad):
6     edad_w = min(edad, 120)
7     t = anno_nac + edad_w
8     if sex == 'V':
9         b = np.exp(-permf2012c['fac_m'][edad_w]*(t-2012))
10        return permf2012c['qx_m'][edad_w] * b / 1000
11    else:
12        b = np.exp(-permf2012c['fac_f'][edad_w]*(t-2012))
13        return permf2012c['qx_f'][edad_w] * b / 1000
14
15 print(per2012c('V', 1979, 41))
```

0.0005679

10.3. Ejercicio 19

Ejercicio_09_01

Utilizando Pandas y NumPy, calcular las edades para cada asegurado del colectivo incluido en el fichero *edades.csv*, a la fecha de valoración 31-12-2019.

AYUDA:

- La fecha de valoración puede fijarse como `pd.to_datetime('2019-12-31')`
- La librería `datetime` puede ser llamada usando `.dt` así como sus funcionalidades.
Ejemplo: `(fecha_1 - fecha_2).dt.days`
- La función `rint` de `numpy` (`np.rint`) redondea al valor entero. Del mismo modo puede usarse `np.round()` sin especificar número de decimales

```
1 import pandas as pd
2 import numpy as np
3
4 f_val = pd.to_datetime('2019-12-31')
5 df = pd.read_csv("edades.csv", sep=";", names=['Nombre', 'Fnac', '
6     Importe'])
7 print(df)
8
9 df['Fnac'] = pd.to_datetime(df['Fnac'].astype(str), format='%d/%m
10     /%Y')
11 df['Edad'] = np.rint((f_val - df['Fnac']).dt.days / 365.2425)
12 print(df)
```

10.4. Generación de series temporales

Generar un rango de datos, por ejemplo desde el momento actual hasta el vencimiento o fallecimiento del asegurado, es una tarea sencilla utilizando la librería Pandas y su función `pd.date_range(args)`, con los siguientes argumentos:

- **start**: Inicio del rango. Límite izquierdo para generar fechas.
- **end**: Fin del rango. Límite derecho para generar fechas.
- **period**: Número de periodos a generar
- **freq**: Frecuencia de las proyecciones.
 - D: Diaria (*por defecto*)
 - Y: Anual
 - M: Mensual
- **closed**: Si queremos excluir el inicio (`closed='right'`) o el final (`closed='left'`)
- **name**: Nombre del DatetimeIndex resultante (*por defecto ninguno*)

Ejemplo:

```
1 import pandas as pd
2 s = pd.date_range(start='31/12/2019', periods=8, freq='Y')
3 df = pd.DataFrame(s, columns=['Fecha'])
4 print(df)
```

	Fecha
0	2019-12-31
1	2020-12-31
2	2021-12-31
3	2022-12-31
4	2023-12-31
5	2024-12-31
6	2025-12-31
7	2026-12-31

10.5. Ejercicio 20

Ejercicio_09_02

Utilizando Pandas y NumPy, calcular el NPV de 100 euros pagaderos en 10 anualidades, desde el 31-12-2019 al 31-12-2028.

- Descontados al 2 %
- Descontados utilizando la curva libre de riesgo (RFR) de EIOPA a 31-12-2019.

NOTA: Para la RFR se puede reutilizar la programación del Ejercicio 04_01.

```
1 import pandas as pd
2 import numpy as np
3 s = pd.date_range(start='31-12-2019', periods=10, freq='Y')
4 df = pd.DataFrame(s, columns=['Fecha'])
5 df['Importe'] = 100.
6 df['factor'] = 1 / (1+0.02) ** df.index
7 print(df)
8 print("\nNPV: %.2f" % (np.vdot(df.Importe, df.factor)))
```

NPV (2%): 916.22

	Fecha	Importe	dto	rfr	dto_rfr
0	2019-12-31	100.0	1.000000	-0.00421	1.004228
1	2020-12-31	100.0	0.980392	-0.00391	1.007866
2	2021-12-31	100.0	0.961169	-0.00338	1.010209
3	2022-12-31	100.0	0.942322	-0.00285	1.011482
4	2023-12-31	100.0	0.923845	-0.00229	1.011529
5	2024-12-31	100.0	0.905731	-0.00164	1.009897
6	2025-12-31	100.0	0.887971	-0.00084	1.005900
7	2026-12-31	100.0	0.870560	-0.00018	1.001441
8	2027-12-31	100.0	0.853490	0.00047	0.995780
9	2028-12-31	100.0	0.836755	0.00113	0.988770

NPV (RFR): 1004.71

10.6. Librería Pyliferisk

Pyliferisk es una biblioteca escrita en Python para cálculo actuarial, basada en la notación actuarial comúnmente usada (International Actuarial Notation).

- **Documentación:** <https://github.com/franciscogarate/pyliferisk>
- **Licencia:** GPL v3.0 (*open source*)
- **Principales utilidades:**
 - Funciones biométricas (q_x , l_x , w , dx , ex)
 - Valor actual actuarial
 - Rentas actuariales
- Los nombres de las fórmulas son fácilmente adivinables (q_x , l_x , p_x , w , dx , ex , A_x , $A_{x:n}$...), con algunas excepciones en cuanto a caracteres especiales.
- No posee dependencias de otras librerías, lo que disminuye el tiempo de ejecución del cálculo en comparación con implementaciones en otras librerías como Pandas.

Las hipótesis de mortalidad pueden ser fijadas fácilmente con la función **.MortalityTable()** con los siguientes parámetros:

- **nt:** La tabla actuarial utilizada para realizar los cálculos de contingencias de vida. Ejemplo: `nt=GKM95`
- **perc:** Variable opcional que indica el porcentaje de mortalidad que debe aplicarse. Ejemplo: `perc=85`. El porcentaje de la variable puede omitirse, en este caso será 100 por defecto.

Una vez tenemos fijadas las hipótesis de mortalidad, podemos llamar (entre otras) a las siguientes funciones:

- **.qx[x]:** Returns the probability that a life aged x dies before 1 year
With the convention: the true probability is $q_x/1000$
- **.lx[x]:** Returns the number of survivors at beginning of age x
- **.w:** ultimate age ($lw = 0$)
- **.dx[x]:** Returns the number of dying at beginning of age x
- **.ex[x]:** Returns the curtate expectation of life. Life expectancy

Ejemplo 1: Calcular la edad límite y la q_x a los 50 años para ambas tablas:

```
1 from pyliferisk import MortalityTable
2 from pyliferisk.mortalitytables import SPAININE2004, GKM95
3 tariff = MortalityTable(nt=SPAININE2004)
4 experience = MortalityTable(nt=GKM95, perc=85)
5 # Print the omega (limiting age) of the both tables:
6 print(tariff.w)
7 print(experience.w)
8 # Print the  $q_x$  at 50 years old:
9 print(tariff.qx[50] / 1000)
10 print(experience.qx[50] / 1000)
```

101

121

0.003113

0.003662395

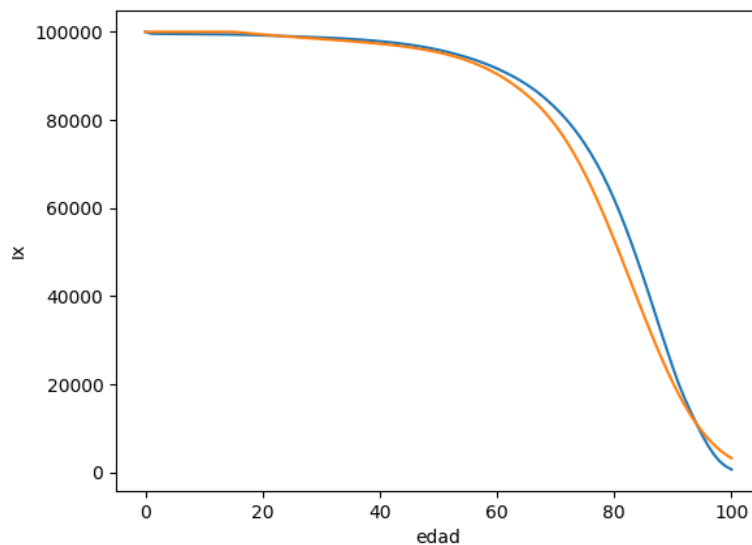
Ejemplo 2. Calcular la esperanza de vida para un varón de 65 años con una mortalidad acorde al 70 % de las PASEM2010:

```
1 from pyliferisk import MortalityTable
2 from pyliferisk.mortalitytables import SPAIN_PASEM2010M
3
4 tabla = MortalityTable(nt=SPAIN_PASEM2010M, perc=70)
5 print("Esperanza vida con 65: %.2f" % (tabla.ex[65]))
```

Esperanza vida con 65: 18.31

Ejemplo 3. Gráfico de la l_x para dos tablas de mortalidad.

```
1 import numpy as np
2 import pyliferisk as life
3 from pyliferisk.mortalitytables import SPAININE2004, GKM95
4 import matplotlib.pyplot as plt
5
6 tarifa = life.MortalityTable(nt=SPAININE2004)
7 experiencia = life.MortalityTable(nt=GKM95, perc=75)
8 x = np.arange(tarifa.w)
9 y = tarifa.lx[:tarifa.w]
10 z = experiencia.lx[:tarifa.w]
11 plt.plot(x, y)
12 plt.plot(x, z)
13 plt.ylabel('lx')
14 plt.xlabel('edad')
15 plt.show()
```



The Present Value of the benefit payment is a function of time of death given a survival model and an interest rate.

- **nt**: La tabla actuarial utilizada para realizar los cálculos de contingencias de vida. Ejemplo: nt=GKM95
- **i**: Tipo de interés. La tasa de interés efectiva. Ejemplo: i=0.02
- **perc**: Variable opcional que indica el porcentaje de mortalidad que debe aplicarse. Ejemplo: perc=85. El porcentaje de la variable puede omitirse, en este caso será 100 por defecto.
- **Ax()**: A_x , Returns the Expected Present Value (EPV) of a whole life insurance (i.e. net single premium). It is also commonly referred to as the Actuarial Value or Actuarial Present Value.
- **Axn()**: $A_{x:\overline{n}|}^1$, Returns the EPV (net single premium) of a term insurance. Sintaxis: Axn(mt, x, n)
- **qAx()**: ${}_qA_x$, This function evaluates the APV of a geometrically increasing annual annuity-due. Sintaxis: Axn(mt, x, q)
- **qAxn()**: ${}_qA_{x:\overline{n}|}$, This function evaluates the APV of a geometrically increasing Term insurance. Sintaxis: qAxn(nt, x, n, q)
- **AExn()**: $A_{x:\overline{n}|}$, Returns the EPV of an endowment insurance. An endowment insurance provides a combination of a term insurance and a pure endowment. Sintaxis: AExn(mt, x, n)

Notation	Description	Syntax
A_x	whole-life death insurance	<code>Ax(nt, x)</code>
$A_{x:\overline{n} }^1$	Term insurance	<code>Axn(nt, x, n)</code>
$A_{x:\overline{n} }$	Endowment insurance	<code>AExn(nt, x, n)</code>
${}_qA_x$	Increasing whole-life	<code>qAx(nt, x, n)</code>
${}_qA_{x:\overline{n} }$	Increasing Term insurance	<code>qAxn(nt, x, n, q)</code>

Ejemplo 1: A whole-life single premium:

```

1 from pyliferisk import Actuarial, Ax
2 from pyliferisk.mortalitytables import GKM95
3
4 mt = Actuarial(nt=GKM95, i=0.02)
5 x = 50          #age
6 C = 1000        #capital
7
8 print(Ax(mt, x) * C)
```

589.0804423991423

Ejemplo 2: A term insurance single premium:

```

1 from pyliferisk import Actuarial, Axn
2 from pyliferisk.mortalitytables import GKM95
3
4 mt = Actuarial(nt=GKM95, i=0.03)
5 x = 40          #age
6 n = 20          #horizon
7 C = 10000       #capital
8
9 print(Axn(mt, x, n) * C)
```

646.1486398262324

10.7. Ejercicio 21

Ejercicio_09_03

Calcular la prima única del seguro temporal del anterior ejemplo 2 siguiendo una aproximación de proyección y descuento de flujos de caja (cash flow approach)

Solución con Numpy:

```
1 from pyliferisk import MortalityTable
2 from pyliferisk.mortalitytables import GKM95
3 import numpy as np
4 mt = MortalityTable(nt=GKM95)
5 x, n, C, i = 40, 20, 10000, 0.03
6
7 payments = np.zeros(n)
8 for t in range(0, n):
9     payments[t] = (mt.lx[x+t] - mt.lx[x+t+1]) / mt.lx[x] * C
10
11 discount_factor = np.zeros(n)
12 for y in range(0, n):
13     discount_factor[y] = 1 / (1 + i) ** (y + 0.5)
14
15 print(np.dot(discount_factor, payments).round(2))
```

646.15

Solución con Pandas:

```
1 from pyliferisk import MortalityTable
2 from pyliferisk.mortalitytables import GKM95
3 import numpy as np
4 import pandas as pd
5 mt = MortalityTable(nt=GKM95)
6 x, n, C, i = 40, 20, 10000, 0.03
7
8 df = pd.DataFrame(np.arange(x, x+n+1), columns=['edad'])
9 df['t'] = np.arange(0, n+1)
10 df['lx'] = df['t'].apply(lambda t : mt.lx[x+t])
11 df['qx'] = df['lx'].diff(-1).fillna(0)/df['lx'][0]
12 df['payments'] = df['qx'] * C
13 df['disc'] = df['t'].apply(lambda t : 1 / (1 + i) ** (t + 0.5))
14 print(np.dot(df.payments, df.disc).round(2))
```

646.15

10.8. Rentas actuariales:

- Las rentas pueden ser temporales o vitalicias.
- Una anualidad vitalicia se refiere a una serie de pagos a un individuo siempre y cuando el individuo esté vivo a la fecha de pago.
- Los intervalos de pago pueden comenzar inmediatamente o diferidos.
- El pago puede ser pagadero al principio de los intervalos (anualidad prepagable) o al final (anualidad pospagable)
- Adicionalmente, pueden ser rentas inmediatas o diferidas, y con crecimiento aritmético o geométrico.
- La librería `pyliferisk` ofrece la función **`annuity()`** donde pueden especificarse los anteriores parámetros.

`annuity(mt, x, n, p, m, ['a/g', q], -d)`

- **`mt`**: la tabla de mortalidad
- **`x`**: la edad como número entero.
- **`n`**: Un número entero (duración del seguro en años) o `'w'` = toda la vida.
- **`p`**: Momento de pago. Sintaxis: 0 = inicio de cada período (prepagable), 1 = final de cada período (postpagable),

Parámetros opcionales:

- **`m`**: A pagar `'m'` por año (pagos fraccionados). Por defecto = 1 (anualmente)
- **`'a' / 'g'`** = Aritmética / Geométrica.
- **`q`**: La tasa de incremento. Sintaxis: `['g', q]` o `['a', q]`. Por ejemplo, `['g', 0.03]`
- **`-d`**: El período de diferimiento en n-años en número negativo.

Notation	Description	Syntax
$\ddot{a}_{x:\overline{n} }$	n-year temporary life annuity-due	<code>annuity(nt,x,n,0)</code>
$a_{x:\overline{n} }$	n-year temporary life annuity	<code>annuity(nt,x,n,1)</code>
$\ddot{a}_{x:\overline{n} }^{(m)}$	n-year annuity-due m-monthly payments	<code>annuity(nt,x,n,0,m)</code>
\ddot{a}_x	whole life annuity-due	<code>annuity(nt,x,'w',0)</code>
a_x	whole life annuity	<code>annuity(nt,x,'w',1)</code>
$\ddot{a}_x^{(m)}$	whole life annuity-due m-monthly	<code>annuity(nt,x,'w',0,m)</code>
$a_x^{(m)}$	whole life annuity m-monthly	<code>annuity(nt,x,'w',1,m)</code>
${}_n \ddot{a}_x$	d-year deferred whole life annuity-due	<code>annuity(nt,x,n,0,-d)</code>
${}_n a_x$	d-year deferred whole life annuity	<code>annuity(nt,x,n,1,-d)</code>
${}_n \ddot{a}_{x:\overline{n} }^{(m)}$	d-year deferred n-year temporary annuity-due m-monthly payments	<code>annuity(nt,x,n,0,m,-d)</code>
${}_n \ddot{a}_x$	d-year deferred whole life annuity-due	<code>annuity(nt,x,'w',0,-d)</code>
${}_n a_x$	d-year deferred whole life annuity	<code>annuity(nt,x,'w',1,-d)</code>

Ejemplo 1 renta financiera: The present value of a 5-year (financial) annuity with nominal annual interest rate 12 % and monthly payments of \$100:

```

1 from pyliferisk import *
2 import numpy as np
3
4 mt = Actuarial(nt=FIN, i=0.12/12) #FIN = Financial table
5
6 n = 5 * 12
7 C = 100
8
9 print(annuity(mt, 0, n, 1) * C) #replace age 'x' by 0

```

4495.503840622397

Ejemplo 2: Cálculo de prima: A Life Temporal insurance for a male, 30 years-old and a horizon for 10 years, fixed annual premium (GKM95, interest 6 %):

Actuarial equivalence: $\pi_{30:\overline{10}|}^1 = 1000 \cdot \frac{A_{30:\overline{10}|}^1}{\ddot{a}_{30:\overline{10}|}}$

```

1 from pyliferisk import *
2 from pyliferisk.mortalitytables import GKM95
3
4 nt = Actuarial(nt=GKM95, i=0.06)
5 x = 30
6 n = 10
7 C = 1000
8
9 print(C * (Axn(nt, x, n) / annuity(nt, x, n, 0)))

```

1.398266715155939

10.9. Ejercicio 22

Ejercicio_09_04

Calcular la reserva de una seguro temporal de vida-riesgo a prima periódica aplicando el principio de equilibrio, para una edad de 40, duración 20, capital 100.000 y base técnica: GKM95 al 3 %.

$$\begin{aligned} {}_0V_x &= A_x - \pi \cdot \ddot{a}_x = 0 \\ \pi &= \frac{A_{x:\overline{n}|}^1}{\ddot{a}_{x:\overline{n}|}} \\ {}_tV_x &= A_{40+t:\overline{10-t}|}^1 - \pi \cdot \ddot{a}_{40+t:\overline{10-t}|} \end{aligned}$$

```
1 from pyliferisk import *
2 from pyliferisk.mortalitytables import GKM95
3
4 nt = Actuarial(nt=GKM95, i=0.03)
5 x, n, Cm = 40, 20, 100000
6
7 Premium = Cm * Axn(nt, x, n) / annuity(nt, x, n, 0)
8
9 def Reserve(t):
10     return round(Cm * Axn(nt, x+t, n-t)
11                  - Premium * annuity(nt, x+t, n-t,
12                  0), 2)
13
14 for t in range(0, n+1):
15     print(t, Reserve(t))
```


10.10. Flujos de caja con Pandas

Cálculos para una cartera de pólizas en Pandas:

- Cada iteración generará un dataframe, y la última instrucción de la iteración deberá ser anexar dicho df a una lista vacía (`lista=[]`) que previamente se ha creado.
- Una vez acabado el for, tendremos una lista con n DataFrame que juntaremos con **`pd.concat(lista, ignore_index=True)`**.
- El output final (el df total) puede agruparse además que por el campo `'t'` por más campos (como por ejemplo: por póliza o producto y por año de emisión) con un simple `groupby`. Igualmente puede crearse previamente un campo calculado como la concatenación de varios campos y hacer un `groupby` final por dicho campo.
- El límite del dataframe está en la memoria RAM del PC. Si se tiene previsto superar dicha capacidad (el comando `df.info()` informa de la memoria de cada dataframe), pueden crearse ficheros temporales (se trata más adelante).
- Existen diferentes recursos si queremos reducir el tiempo de cálculo en Python (Cython, Numba, computación en paralelo o distribuida, clusters, multiproceso...)

```
1 import pandas as pd
2
3 polizas = []
4 for i in range(5):
5     temp = pd.DataFrame([i], columns=['t'])
6     temp['Importe'] = i * 10
7     polizas.append(temp)
8
9 total = pd.concat(polizas, ignore_index=True)
10 agg = total.groupby(['t']).agg({'Importe': 'sum'})
11 print(agg)
```

Importe	
t	
0	0
1	10
2	20
3	30
4	40

Ejemplo sencillo de concatenar dos df, y posteriormente calcular la suma de los importes:

```

1 import pandas as pd
2
3 a = pd.date_range('2019-01-01', periods=12, freq='M', name='t')
4 df = a.to_frame()
5 df['Importe'] = 1.
6
7 b = pd.date_range('2019-01-01', periods=11, freq='M', name='t')
8 df2 = b.to_frame()
9 df2['Importe'] = 2.
10
11 total = pd.concat([df, df2], ignore_index=True)
12 agg = total.groupby(['t']).agg({'Importe': 'sum'})
13 print(agg)

```

	Importe
t	
2019-01-31	3.0
2019-02-28	3.0
2019-03-31	3.0
2019-04-30	3.0
2019-05-31	3.0
2019-06-30	3.0
2019-07-31	3.0
2019-08-31	3.0
2019-09-30	3.0
2019-10-31	3.0
2019-11-30	3.0
2019-12-31	1.0

Cotizaciones de colectivos en Pandas sin utilizar proyecciones:

```

1 import pandas as pd
2 from pyliferisk import *
3
4 mt = Actuarial(nt=GKM80, i=0.04)
5 def single_risk_premium(x, n):
6     x = int(x); n = int(n)
7     return nEx(mt, x, n) + Axn(mt, x, n)
8 def annual_risk_premium(x, n):
9     x = int(x); n = int(n)
10    return(single_risk_premium(x, n) / annuity(mt, x, n, 0))
11
12 df = pd.read_csv('colective.csv', sep=';', decimal=',')
13 df['Single Premium'] = df.apply(lambda x: single_risk_premium(x['age'], x['duration']), axis=1) * df['capital']
14 df['Annual Premium'] = df.apply(lambda x: annual_risk_premium(x['age'], x['duration']), axis=1) * df['capital']

```

```
6670884.68 998828.42
```

10.11. Ejercicio 23

Ejercicio_09_05

Calcular la reserva siguiendo un método prospectivo y proyecciones anuales para un seguro de Decesos (Vida Entera), con los siguientes datos:

- Edad actual: 31
- Capital constante: 3.000
- Prima nivelada calculada a la edad de 25 años con las INM05 al 5 % ($\pi = C \cdot A_{25}/\ddot{a}_{25}$)
- Hipótesis mortalidad: INM05
- Tasa de descuento a utilizar: 1.39 %

Aviso

Versión utilizada de pyliferisk: 1.11

Implementar un calendario de pagos teniendo en cuenta festivos:

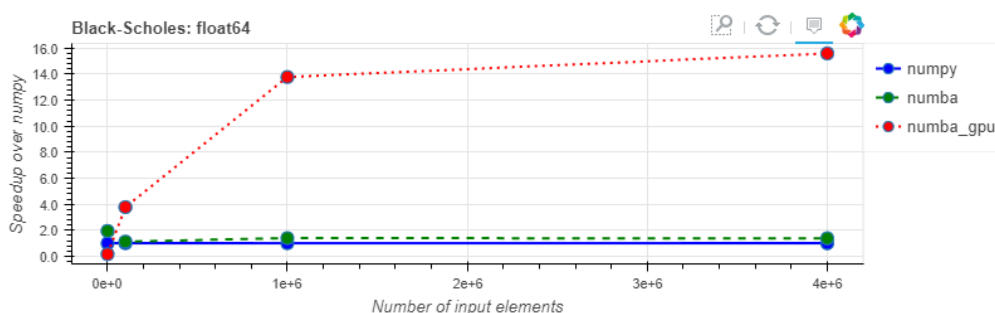
<https://towardsdatascience.com/holiday-calendars-with-pandas-9c01f1ee5fee>

11. Aumento de velocidad de cálculo

11.1. Cython: Compilar código Python en C

Python dispone, entre otras, de la librería **Cython** para convertir y compilar el código de Python a C (con algunas modificaciones en el código). Así, puede utilizarse directamente el programa en C o compilarse únicamente el motor de cálculo y llamar a dicho código en C desde Python con la librería CPython.

Numba, es un compilador de bajo nivel (LLVM) que sin modificar nuestro código en Python, permite mayor velocidad de cálculo.



Cython es Python con tipos de datos C.

- Casi cualquier código Python es también un código Cython válido. El compilador de Cython lo convertirá en código C que hace llamadas equivalentes a la API de Python/C.
- Ejemplo. Hacemos nuestro código en Python y guardamos el fichero con la extensión `.pyx`:

```
1 def fib(n):
2     a, b = 1, 1
3     for i in range(n):
4         a, b = a+b, a
5     return a
```

- Situamos en la misma carpeta el siguiente fichero **setup.py** apuntando la función **cythonize** a nuestro fichero `.pyx`

```
1 from distutils.core import setup
2 from Cython.Build import cythonize
3
4 setup(
5     ext_modules = cythonize("fib.pyx")
6 )
```

- Ejecutamos desde consola:

```
python setup.py build_ext --inplace
```

o

```
python setup.py build_ext --inplace --force --compiler=mingw32
```

y obtenemos nuestro fichero **fib.c**

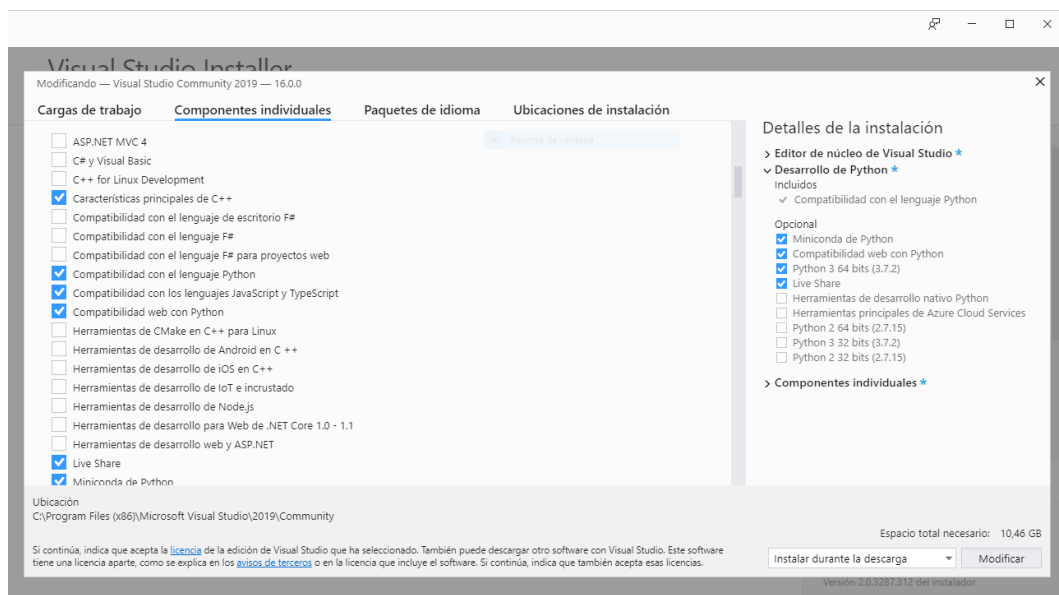
Con nuestro código en C (fib.c), tenemos dos opciones:

- Utilizarlo para ser llamado desde cualquier código de Python con **CPython**.
- Compilarlo en la misma máquina que lo vamos a utilizar para generar un fichero .exe (en windows) o .so en (Linux/OS X).

Para ello necesitamos un compilador de C o C++. En Mac/Linux disponemos de gcc:

```
gcc fib.c -o fib.exe
```

En caso de Windows puede instalarse **mingw32** o el paquete Visual Studio 17 (o 19) que posee soporte para Python (con la librería Python.h)



11.2. Cloud computing

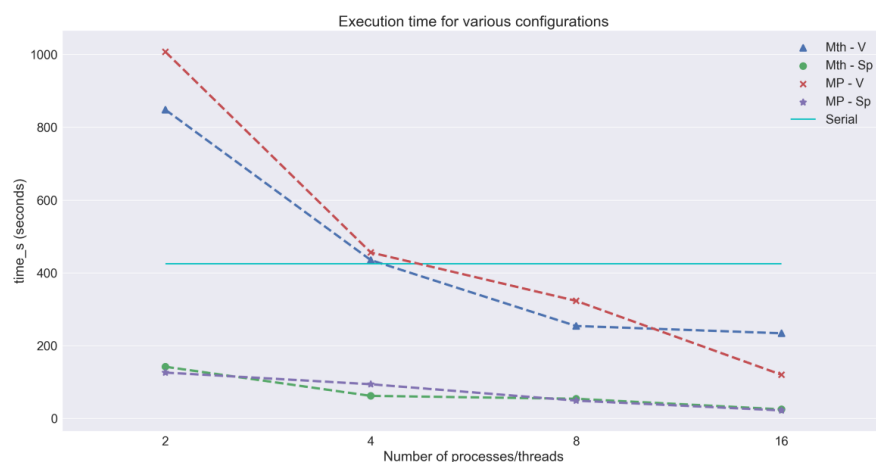
Con el objeto de reducir el tiempo de cálculo, Python dispone de muchos recursos para la **computación en paralelo o distribuida**:

- Symmetric Multiprocessing
- Cluster Computing
- Cloud Computing
- Grid Computing

Computación paralela o distribuida

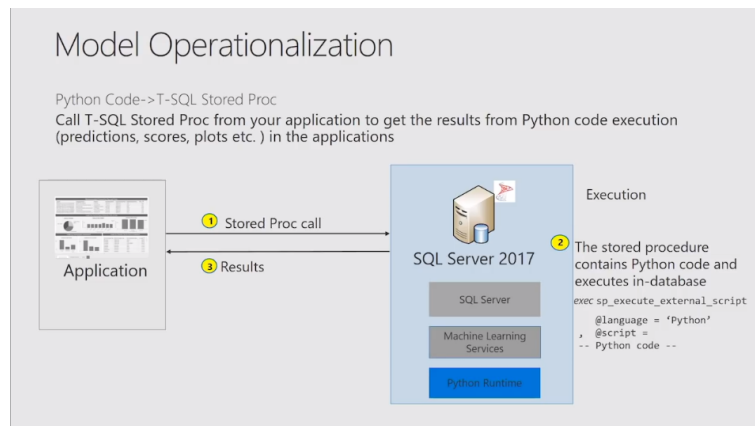
En caso de disponer de un **cluster** de ordenadores o una **granja de computadoras** (grid) podemos hacer uso de herramientas de computación distribuida. Dependiendo de los cálculos a efectuar, la solución puede cambiar. *(En este ejemplo el multiproceso es más lento).*

```
1 import multiprocessing as mp
2 import numpy as np
3 num_process_mi_pc = mp.cpu_count()
4
5 def f(x):
6     return(x ** 2)
7
8 s = np.arange(100000)
9 #f(s)
10 if __name__ == '__main__':
11     pool = mp.Pool(processes=num_process_mi_pc) #workers pool
12     pool.map(f, s)
```



In-database computing

A partir de la versión de SQL Server 2017, Microsoft ha dado soporte al controlador de Python para conexiones SQL (pyodbc), ofreciendo desde su plataforma en la nube (Microsoft Azure) servicios de *Machine Learning* con Python y R.



Esto permite correr en SQL Server nuestro código con la instrucción `sp_execute_external_script`, y evitar así el movimiento o lectura remota de los datos:

```
exec sp_execute_external_script
@language = N'Python',
@script = N'
import sys
import os
print("*****")
print(sys.version)
print("!!!Hello World!!!")
print(os.getcwd())
print("*****")
'
```

Message

STDOUT message(s) from external script:

C:\Program Files\Microsoft SQL Server\MSSQL14.CTP20\PYTHON_SERVICES\lib\site-packages\revoscalepy

3.5.2 [Anaconda 4.2.0 (64-bit)] (default, Jul 5 2016, 11:41:13) [MSC v.1900 64 bit (AMD64)]

!!!Hello World!!!

C:\PROGRAM~1\MICROS~1\MSSQL~1\1.CTP\MSSQL\EXTENS~1\CTP2001\05C77908-592C-4AE0-9946-577505A4B0C5
