

The slide features a large title 'PYTHON PARA FINANZAS QUANT' at the top. Below it is a block of Python code for financial calculations, including functions for calculating option prices and implied volatilities. In the center, there is a blue box containing the text 'Primeros Scripts'. To the right of the box, the text 't[0]' is displayed. In the bottom left corner, there is a white Python logo icon.

```
• def normalInv(x):
    return ((1/math.sqrt(2*math.pi)) * math.exp(-x*x*0.5))

• def callPrice(S0, K, r, T, sigma, q=0):
    d1 = (math.log(S0/K) + (r-q+sigma*sigma/2)*T) / (sigma * math.sqrt(T))
    d2 = d1 - sigma*math.sqrt(T)
    ret['call'] = S0 * math.exp(-q*T) * normalInv(d1) - K * math.exp(-r*T) * normalInv(d2)
    ret['vega'] = 0.01 * S0 * math.exp(-q*T) * normalInv(d1) * math.sqrt(T)
    ret['theta'] = -(0.01*sigma*S0) * (-((S0*sigma*math.exp(-q*T))/(2*math.sqrt(T))) * normalInv(d1))
    ret['rho'] = 0.01 * K * T * math.exp(-r*T) * d1*d2

    if d1 < -5:
        ret['errors'] = "Se ingresaron valores incorrectos"
    return ret

• def input(S0, K, r, T, sigma, q=0):
    ret = {}
    if (S0 > 0 and K > 0 and r >= 0 and T > 0 and sigma > 0):
        d1 = (math.log(S0/K) + (r-q+sigma*sigma/2)*T) / (sigma * math.sqrt(T))
        d2 = d1 - sigma*math.sqrt(T)
        ret['put'] = K * math.exp(-r*T) * normalInv(d2) - S0 * math.exp(-q*T) * normalInv(d1)
        ret['delta'] = - math.exp(-q*T) * normalInv(d1)
        ret['gamma'] = math.exp(-q*T) * normalInv(d1) / (S0 * sigma * math.sqrt(T))
        ret['vega'] = 0.01 * S0 * math.exp(-q*T) * normalInv(d1) * math.sqrt(T)
        ret['theta'] = -(0.01*sigma*S0) * (-((S0*sigma*math.exp(-q*T))/(2*math.sqrt(T))) * normalInv(d1))
        ret['rho'] = 0.01 * K * T * math.exp(-r*T) * d1*d2

    else:
        ret['errors'] = "Se ingresaron valores incorrectos"
    return ret

• def callPrima(S0, K, r, T, prima, q=0):
    if (S0 > 0 and K > 0 and r >= 0 and T > 0):
        maximasIteraciones = 300
        techo = prima
        S0 = prima
        while maximasIteraciones > 0:
            maximasIteraciones -= 1
            for i in range(1,maximasIteraciones):
                S0 = number(71000)
                nCall(S0, K, r, T, sigma, q)[i][0]
            S0 = nCall(S0, K, r, T, sigma, q)[0][0]
            if S0 > techo:
                techo = S0
            else:
                break
        print("Created on Sun Mar 30 12:00:20 2020")
        print("Author: Juan Pablo Testler UC3NA")
        print("Python Version: 3.8.5")
        print("NumPy Version: 1.19.2")
        print("Pandas Version: 1.1.3")
        print("SciPy Version: 1.5.2")
        print("Matplotlib Version: 3.3.2")
        print("Math Version: 1.8.2")
    else:
        print("Se ingresaron valores incorrectos")
```

Primeros Scripts

t[0]



 Juan Pablo Pisano
@johnGalt_is www.

Impreso en:
laimprentadigital.com.ar

Pisano, Juan Pablo

Python para finanzas quant : primeros scripts / Juan Pablo Pisano. - 2a ed. --
Florida Oeste : Logikamente, 2021.

Libro digital, PDF

ISBN 978-987-47669-4-6

1. Lenguaje de Programación. 2. Finanzas. 3. Mercado de Capitales. I. Título.

CDD 005.284

Fecha de catalogación: 10/2021

Segunda edición: noviembre 2021

@JohnGalt_is_www

La Imprenta Digital SRL, Florida, Bs As. www.laimprentadigital.com.ar

Edición digital de distribución gratuita, prohibida su venta

ISBN 978-987-47669-4-6

Queda hecho el depósito que marca la ley 11.723

Obra de distribución gratuita a partir de noviembre de 2021 😊

Indice t[0] Primeros scripts

Introducción	11
Consideraciones generales	11
El Open Source	14
GitHub.....	15
StackOverFlow	16
Google	16
Lenguajes	16
La programación y la era de la Big Data	19
El futuro de la programación en el ámbito financiero	20
Las herramientas estadísticas y matemáticas	21
¿Por qué Python?	22
Las 8 premisas a evaluar en un programa	24
Diferentes entornos para programar	25
Instalaciones.....	27
Primer Hola Mundo en diferentes entornos	28
Consola básica de python	28
IDE con intérprete online	29
Spyder	29
Jupyter Notebooks.....	32
Google Colab.....	35
Ventajas respecto a los Jupyter Notebooks.....	35
Desventajas respecto a los Jupyter Notebooks	35
Primeros pasos en Google Colab	35
Editores de código, Atom, SublimeText	37
Anaconda Navigator	38
Básicos de Jupyter Notebook	39
Atajos de Teclado de Jupyter	41
Inputs, Outputs y Kernel	42
Outputs versus impresiones de variables	44

Variables.....	44
Asignación de valores a variables	46
Asignación de variables por consola o interfaz de usuario	46
Python es Case Sensitive.....	47
Tipos de datos	48
Otros tipos de Variables	50
Errores por operar tipos de datos diferentes	51
Operaciones permitidas con diferentes tipos	51
Python es un lenguaje interpretado, dinámicamente tipado	52
Operaciones Básicas.....	53
Métodos básicos para trabajar con strings.....	54
Ejercicios con Strings 1	56
Respuestas, ejercicios con strings 1	56
.....	56
Palabras Reservadas	58
Números al Azar	59
Valor flotante al azar entre 0 y 1	59
Valor flotante entre min y max.....	59
Valor entre min y max definiendo intervalo o "step"	59
Número aleatorio en una Distribución.....	60
Semilla	61
Operaciones Matemáticas. Librería Math	63
Constantes	64
Truncado, piso y techo de un flotante	64
Logaritmos	65
Máximo común divisor	65
Factorial	65
isClose	65
Ejercicios con random	66
Respuestas.....	67

Trabajando con Fechas.....	69
La librería datetime	69
Atributos del objeto datetime	70
¿Y si quiero solo la fecha y no me importa la hora?	71
Fechas en formato string	71
Convertir un string en un Objeto de DateTime	72
Directivas de día, mes y año	73
Directivas de zonas horarias	73
Otras Directivas.....	74
Generar una fecha como objeto de datetime	77
Convertir un objeto Datetime a string	77
Distintos formatos para mostrar fechas	78
Seteo de parametrización local	78
¿Qué es un timestamp?	79
Pasando a Otro huso horario	80
Librería Calendar	81
Impresión en pantalla de un calendario	81
Años bisiestos	82
Funciones básicas de calendar.....	82
Función monthrange.....	83
Ejercicios.....	84
Respuestas.....	85
Colecciones de datos en python.....	87
Listas	87
Slicing de una lista.....	89
Tuplas	94
Métodos de Tuplas y Listas.....	96
Métodos de las listas.....	97
Diccionarios	100
Acceso a valores puntuales de un diccionario	101
Inicialización de Listas y Diccionarios	104

Asignación ligada	105
Mas ejemplos de asignaciones ligadas	106
El método copy().....	106
Generadores.....	107
Colecciones por comprensión	108
Ejercicios de colecciones I	109
Respuestas ejercicios de colecciones I.....	111
Decisiones.....	115
Sentencia IF	118
Concepto de la identación	118
Evaluación de True o False en distintos tipos de Variables.....	119
Decisiones concatenadas y anidadas	123
Decisiones Consecutivas	123
Decisiones Anidadas	124
Operadores Lógicos.....	125
Tablas de VERDAD	125
Manejo básico de errores, Try/Except	127
Catching, capturando el tipo de error.....	129
Ejercicios de condiciones.....	131
Respuestas ejercicios de condiciones	134
Archivos necesarios.....	139
Instalación Necesaria	139
Ciclos Finitos e Infinitos	139
Ciclos Definidos.....	140
Iterando una lista	140
Iterando el índice	141
Iterando el índice y el elemento	141
Iterando el elemento y generando el índice dentro de la iteración	142
Iterando una Tupla	142
Tuplas de 1 elemento en Python.....	142
Iterando una lista de tuplas	144

Iterando dentro de una iteración.....	144
Iterando un Diccionario.....	145
Iterar diccionario por sus claves	145
Iterar diccionario por sus valores.....	145
Iterar diccionario por sus ítems	146
Operadores de asignación	146
Lista de operadores de asignación.....	148
Ciclos definidos usando la función range.....	149
Creando listas con ciclos definidos	150
Ciclos Indefinidos Finitos	151
Ciclos Infinitos	152
Corte de ciclos infinitos con corte por TimeOut.....	153
Corte de Ciclos infinitos con Centinela	154
Importación de datos de una planilla.....	155
Que es un archivo CSV	155
Librería CSV	155
Generamos una columna nueva a partir de los datos dados	159
Lectura de balances desde un CSV	161
Bucles Anidados.....	163
Ejercicios con CSVs	167
Respuestas ejercicios con CSVs.....	169
Ejercicios de cálculo con ciclos	177
Respuestas a ejercicios de cálculo con ciclos	179
Palabras Finales	182

Introducción

Consideraciones generales

Si son personas pragmáticas y les gusta ir bien al grano directo en todo, salteen estas páginas introductorias que es todo sarasa, listo lo dije, al que quiera relajarse un rato y meterse en esta intro de a poco antes de ir al grano a continuación, mis palabras.

Esta obra nació de la idea de empezar a compilar material para una versión online o digital del curso preQuant que doy en la Ucema, me preguntaron infinidad de veces si pensaba hacer el curso en formato digital y lo primero que se me vino a la mente con esa pregunta es ¿cómo sería ese curso digital? ¿Qué recursos tendría aparte de unos videítos mostrándome a mi codeando?, y no es un detalle menor, porque aprender las herramientas de análisis financiero cuantitativo requiere en realidad de la conjunción de 3 disciplinas: Programación, Finanzas y Matemática (Álgebra y Estadística, sobre todo).

Partí de la base que la mayoría de interesados tiene una base de conocimiento de instrumentos financieros, quizá sea porque me desempeño en este rubro y por eso veo ese background común entre los interesados, pero a su vez la gran mayoría tiene nulos conocimientos de programación y una base de matemática bastante pobre también, con lo cual el desafío es grande porque son todas disciplinas con una base teórica pero también con mucha base práctica. Siempre digo que aprender a programar es como aprender a manejar o a andar en bici, te pueden explicar la teoría en dos patadas y la verdad que no hay gran ciencia detrás pero de ahí a salir andando en bici por primera vez o a manejar en una ruta hay un gap que solo se resuelve con práctica no se necesita dedicar una vida ni mucho menos pero si muchas horas de práctica para dominar bien las herramientas de programación.

Volviendo al tema del nacimiento de la idea de esta serie de libros, empecé con la idea de armar guías de prácticas, sobre todo de las herramientas de python en forma gradual como las vamos viendo en el curso, y de a poco fue tomando forma de libro, el detalle es que son muchísimos temas y muchas herramientas y se hace medio denso y no iba a terminarlo jamás, así que decidí partirla en tomos o capítulos, vamos a arrancar con el tomo 0, ya que en programación siempre se arrancan los índices con un 0, y si no fallan mis cálculos preliminares vamos a llegar al tomo 15 en los últimos temas en donde nos enfocamos en herramientas de machine learning y Deep learning o como lo llaman ahora que queda más cool a un “Inteligencia Artificial”

Los tomos siempre van a estar centrados en las herramientas informáticas en sí. Mientras escribo estas líneas ya tengo escritos los tres primeros tomos, el primero el T0 es un tomo de scripts iniciales, es decir las primeras líneas de código donde vemos variables flujo de un condicional y ciclos infinitos y finitos, etc, el tomo siguiente T1, se basa en una librería muy conocida de Python que se usa para trabajar con matrices o arreglos bidimensionales de datos (tablas de filas y columnas como las de Excel pero más estructuradas por decirlo de algún modo), luego el tercer tomo se basa en el uso de herramientas gráficas y el uso de funciones para no reescribir código, el cuarto tomo tengo pensado ya dedicarlo entero al uso de APIs y data feed, es decir a la recolección automática de datos en tiempo real en forma online y automatizada. Ya para el quinto tomo se empieza a poner más interesante porque con todas las herramientas vistas hasta ahí ya se pueden armar

backtestings y screeners con buenos reportes gráficos. Luego vendrá seguramente un tomo entero dedicado a bases de datos, es un tema central este para todo lo que es data science profesional y es un tema al que no se le da mucha bola en los cursos.

Y así la idea es como les dije antes terminar con algos de machine learning y siempre pero siempre todos los tomos, la idea es que tengan toda ejercitación bien aplicada al campo financiero

¿A quién o qué público están apuntados los libros?

Me han preguntado mucho esto en mi cuenta de twitter, generalmente interesados en las herramientas mas cuantitativas y tecnológicas del campo financiero. Y es una pregunta difícil de responder, voy a separar el público objetivo en 3 tipos de interesados

1. Jóvenes, que trabajan en bancos, estudios contables, consultoría, o que se sienten atraídos por temas económicos y/o financieros, que quieren potenciar su desarrollo laboral.

Para estos jóvenes siempre el consejo es el mismo, más allá del tema financiero en sí, si estudiases abogacía o profesorado de educación física igualmente les diría que herramientas como python en 10 años van a ser como Excel hoy, y hoy mismo, saber python en sus carreras les va a dar un plus seguramente con lo cual ni lo dudaría en tomarlo como una capacitación más, si les gusta y se enganchan mejor, obviamente no es para cualquiera la programación pero si le ponen onda al menos a lo básico tampoco es algo con mayor dificultad

2. Profesionales de carreras de ciencias económicas, licenciatura en administración, carrera de actuario, administradores de carteras de terceros o portfolio managers de fondos importantes. A todos ellos les digo que no se preocupen tanto por el tema técnico de python en sí, seguramente en su carrera laboral les tocará dirigir a jóvenes programadores que estudiaron a fondo este y otros muchos lenguajes de programación.

Pero también les digo que no pueden seguir capacitándose en otros temas dejando de lado esto por completo porque la tendencia al uso de big data, al uso de machine learning, al uso de herramientas cuantitativas que exploten cualquier mínimo sesgo estadístico para mejorar la eficiencia de un proceso es algo que llegó para quedarse. Yo soy el primero en estallar de la risa cuando escucho la preocupación de “¿nos remplazarán los bots?” porque es absurdo ese planteo, no no pueden, pero también es igual de absurdo creer que los bots son solo una moda y que podemos seguir trabajando en el futuro como lo hacíamos hace 10 años o lo seguimos haciendo ahora.. Ya está, les guste o no, la disruptión está instalada y falta la etapa de adaptación, competencia y mejora continua hasta la próxima disruptión, no tiene sentido resistirse

A todo este grupo de interesados, les diría que se lo tomen mas light, pero que le den bola, que entiendan que se puede hacer y que no, que demanda de trabajo y conocimientos conlleva, que velocidades y capacidades tienen estos sistemas, cuales son sus puntos débiles, y un sinfín de etc.

3. Por último hay un grupo no menor de interesados que son traders o inversores independientes, más allá de su trabajo profesional que puede no tener absolutamente nada que ver ni con las finanzas la economía ni incluso con los números pero que en sus finanzas personales, le dedican un tiempo importante a investigar cómo mejorar el rendimiento de sus inversiones personales. Hay de todo en esta selva, están los fundamentalistas del análisis técnico y los del análisis fundamental y también los que usan ambos tipos de análisis, también están los traders que hacen swings semanales o que operan con una frecuencia baja y los scalpers que hacen muchas operaciones diarias, y a su vez están los operadores de futuros, de acciones, de renta fija los loteros u operadores de opciones y dentro de estos los que son de armar estrategias sofisticadas, los arbitrajistas que adoran los sintéticos y los fanáticos que aman ir de frente a una posición especulativa, en fin, la selva es inmensa, pero les soy sincero, no se me ocurre un solo ejemplo al que este tipo de herramientas no le sume, en algunos casos como los más especulativos creo que el uso de este tipo de herramientas es casi un salvavidas que puede salvarles el pellejo más de una vez, en otros casos más de inversores del estilo “value” esto lo veo más como un complemento que les va a agilizar los screeners de activos y les dará un panorama más amplio de datos, comparadores y reportes pero a todos sin excepción este tipo de cosas les va a sumar un montón. Yo lo resumo así: Hoy operar/invertir sin python es como en los 90s hacerlo sin Excel.

¿Por qué vemos más de una vez lo mismo con herramientas diferentes?

¿por qué se dedica un capítulo entero a listas y diccionarios si después se dedica un libro entero a dataFrames de pandas?

Es algo que me pregunté antes de empezar a escribir la primera oración del libro ¿Qué herramientas muestro? Lo primero que se me vino a la mente es como programador más de base, mostrar las herramientas de base del lenguaje (por ejemplo, uso de listas y diccionarios) y después simplemente hacer una mención a Pandas y librerías super útiles como esas para trabajar con matrices de datos.

Pero después dije “no, al revés, si tengo herramientas tan completas como Pandas, para que los voy a torturar con listas llanas y recorrerlas así <peladas>, no tiene sentido” Entonces me autoconvencí de usar la mayor cantidad de librerías con el argumento que mejora mucho la curva de aprendizaje y hace la vida más fácil y la programación más entretenida.. peeeeero, ahí apareció el primer problema

¿Cuál es ese problema? El tema es que en programación por más copada que esté una herramienta (librería, paquete o lo que fuera que alguien pre-armó) siempre, pero siempre le va a faltar algo que queremos personalizar, y para eso vamos a tener que recurrir a la base <pelada> del lenguaje.

Es lo típico que nos pasa con cualquier solución “enlatada” por lo general están buenísimas para ese uso “enlatado” que pensó quien diseñó la solución, pero en donde necesitamos alguna cosita específica, ahí empezamos a decir “que lástima que no se puede hacer..” y si esas cositas específicas empiezan a ser muchas terminamos diciendo “esto solo sirve para...”

Así que si se preguntan en algún momento

“¿para que me hizo dar tantas vueltas con esto, si después me enseñó, unos capítulos después, que se podía hacer mucho más sencillo con tal o cual paquete?”

Bueno, en los párrafos precedentes está la respuesta. Es así, los paquetes y librerías están buenísimas, nos simplifican la vida, nos limpian el código, generalmente son más eficientes en uso de memoria y procesador para el uso típico, pero siempre tendrá alguna customización que no podremos hacer y tendremos que recurrir a “emparcharla” usando herramientas más de base, así que no queda otra que aprenderlas y si enseño solo las herramientas de base, me van a putear en todos los idiomas porque no les enseñé otras formas mucho más sencillas de resolver lo mismo con librerías

El Open Source

Yo particularmente siempre fui muy desconfiado de las soluciones enlatadas, me daba cosa no saber como se resolvían las cosas “por detrás” por eso siempre fui a las bases, pero en python es todo “open source” ¿Qué significa esto?, que el código de base de todos los paquetes es abierto a la comunidad, es decir que cualquiera puede ver “la cocina” del paquete

Por ejemplo si usan una librería para calcular por ejemplo las primas teóricas de una opción según el modelo Black&Scholes o las griegas de las opciones, podrían decir “¿y yo que se si el que hizo esto consideró o no los dividendos o tal o cual cosa..?” bueno, esto no pasa en python porque al ser todo open source, si tienen dudas de cómo se calcula algo en una librería solo tienen que ver el código fuente de esa librería y está todo ahí, muchas veces hasta explicado con lujo de detalles, comentarios y ejemplos que puso el autor para mejorar la experiencia de los usuarios..

Me preguntarán “¿Dónde está la trampa de esto del open source?” y si, es lo primero que se nos viene a la cabeza, ¿por qué alguien haría algo gratuitamente abierto y lo compartiría con la comunidad dejando su trabajo así nomás expuesto?

Hay un sinfín de explicaciones, para un programador profesional las librerías que desarrolla son como la vidriera de un negocio, mientras más gente use sus librerías, más importante se considera su trabajo, así que para los desarrolladores profesionales esta puede ser una opción

Pero no es la única, muchos proyectos open source surgen de una bifurcación de proyectos privados donde el equipo de desarrollo tiene visiones diferentes de cómo mejorar el software, y allí por lo general la solución es separarse en dos proyectos y uno encara para un lado comercial no-open y el otro para el lado mas open source.

Y por supuesto que están también los casos mixtos donde un equipo de desarrolladores arma un paquete open source con la finalidad de darle una vuelta de tuerca en algún momento para una versión comercial o para vender el proyecto cuando tenga muchos usuarios a una empresa grande interesada por el product market fit del servicio o paquete, tal podría ser el caso por ejemplo de YahooFinance que en principio tuvo una API muy usada, que luego quiso limitar, pero que en definitiva varios años después todavía sabe

que es lo que quieren hacer, siguen manteniendo un feed de datos muy completo a nivel global y aun hoy no se decidieron si privatizar el proyecto o que.

Como sea, lo bueno de todas las librerías que seguramente veamos de python es que al ser open source pueden meterse en el código fuente y ver cada línea de código como está hecho cada cálculo, ver los argumentos que toma cada función y ver la documentación completa en el caso que la tenga.

Es más, no solo eso, los proyectos open source no solo permiten que la gente y usuarios sugieran cambios y/o mejoras sino que lo incentivan y agradecen.

Esto nos lleva a saber que en este mundo de los paquetes y librerías open source existen repositorios públicos de código y entre ellos el más conocido del mundo que es GitHub

GitHub

Como dije antes GitHub es el repositorio de código más grande del mundo, pero no es solo un lugar donde se guardan y se pueden descargar los archivos (que además lo es, obvio), sino que los repositorios son además sitios dinámicos donde se guardan todas las versiones y cambios y documentaciones y además son participativos ya que permiten que varios desarrolladores mejoren el mismo repositorio al mismo tiempo e incluso permiten que cualquiera de nosotros hagamos una pequeña modificación en una “bifurcación del código” y les podamos sugerir a los dueños del repositorio que acepten nuestro cambio o sugerencia con solo un clic.. Es largo de explicar pero esto funciona de maravilla, se arman como “ramas” del código original en donde cada programador va haciendo modificaciones y esas ramas se van uniendo a la rama principal a medida que los dueños van aceptando los cambios y actualizaciones sugeridos.

Este tipo de repositorios dinámicos permiten que siempre esté online la versión más reciente y permiten además el trabajo colaborativo, lo que hace que estos proyectos sean monumentales a veces. Es muy loco, pero muchas veces uno usa una librería de estas y no se da una idea el inmenso trabajo que hay detrás y encima es todo gratis, con lo cual muchas veces la explicación a este lujo que nos podemos dar en estas épocas viene por el lado del trabajo colaborativo y la explosión de productividad que eso implica.

Me pasa mucho cuando le explico a alguien muy joven lo hermoso de esta época para aprender a programar comparado con lo que fue cuando aprendí yo, antes para lograr ver una tabla (ni hablar de un gráfico) de un análisis estadístico completo de una serie de datos temporales podría llegar a ser el trabajo de semanas de un programador de los años 90s, pero hoy quizás sean 10 líneas de código de un programador junior de python, sí, el mismo análisis.. esto es gracias a los paquetes open source, así que no sean quejoso y no se pongan a putear cuando algo no sale, valórenlo jaja

Bueno, no les digo más nada, averigüen el resto por su cuenta, vayan a github.com y busquen ahí, tienen repositorios de todos los lenguajes que se imaginen, entre ellos python y de todas las temáticas que se les ocurran, desde temas financieros, climáticos, de astronomía, deportes hasta fitness o arte.

StackOverFLow

No podía hacer una intro a temas de programación sin mencionar stackoverflow.com es el Google de los programadores, es un sitio que al principio medio que cuesta entender como funciona, es como una red social en realidad, hay usuarios expertos programadores en cada lenguaje y usuarios que solo preguntan y es como un inmenso foro de preguntas y respuestas de programación.

Lo bueno de stackoverflow es el formato e interfaz de usuario ya que los espacios para hacer preguntas y respuestas tienen un formato que permiten copiar y pegar código con formato de texto enriquecido para código, se van a dar cuenta lo que digo con la práctica, es muy útil porque a simple vista uno identifica muy rápidamente la respuesta que estaba buscando

Por otro lado, otra gran ventaja es que permite que los usuarios califiquen las respuestas o veten por buena o mala digamos y esto le da un ranking a las respuestas haciendo que cuando uno busca algo que otro usuario ya preguntó (es lo que pasa siempre les aseguro) ve siempre primero las respuestas que mayor cantidad de usuarios calificó como buenas

Como les decía por lo general la duda de uno fue duda de otros en otro momento, solo es cuestión de buscar bien lo que uno necesita responder, a veces en forma de pregunta a veces simplemente copiando y pegando el error que nos tira el programa o la consola

Google

De más está decir que la herramienta por excelencia es siempre Google, aunque en programación les diría que pelea cabeza a cabeza con stackoverflow, porque stackoverflow como les decía antes tiene un formato que lo hace muy piola para programadores y Google es más general, buscando en Google van a encontrar blogs, sitios oficiales de desarrolladores de paquetes, sitios de capacitación, en fin, de todo un poco, mientras quien en stackoverflow van a encontrar más interacción entre programadores directamente.

No se dejen llevar por la frustración de que algo no les sale, me pasó mil veces eso, les aseguro, siempre hay días que uno se traba con algo y está 3 horas sin avanzar una sola línea, sobre todo al principio, pero insistan, googleen e insistan que siempre se encuentra una respuesta

Lenguajes

Muchas veces me han preguntado **por que lenguaje empezar a aprender a programar**, y es una pregunta muy difícil, **es como que te pregunten con que auto aprender a manejar..** y que se yo, si, no es lo mismo la caja de cambios de un fierro modelo 80 que de un auto modelo 2020 y ni que hablar comparado con uno de caja automática, hay muchas diferencias obviamente, pero sinceramente ¿conocen a alguien que haya aprendido con un determinado modelo y no haya podido manejar otro luego? Claro que al principio habrá una

adaptación al nuevo auto, pero la base es la misma, luego de un tiempo de adaptación no hay que aprender a manejar de nuevo el otro auto sino adaptarse nomás..

Con la programación es lo mismo, si aprenden con Python que es un lenguaje de mas alto nivel que C++ si luego se pasan a C++ les va a costar mas que si aprenden con C++ y se pasan a Python, pero la base es siempre la misma, lo que hay que tener más en cuenta a la hora de elegir el primer lenguaje, es más bien el uso típico que le quieren dar, en el primer tema de este libro es lo primero que explico, pero sigo con la analogía con los autos.

Si quiero aprender a manejar porque necesito manejar una camioneta en el campo con caminos de ripio etc no me conviene elegir aprender a manejar con un Smart en Palermo en pleno tránsito y si necesito manejar para ir a ver clientes en medio de la capital todos los días no me conviene tanto practicar con un Falcon en las calles de tierra de la quinta de un amigo, ya que es mas práctico para ese uso, practicar con un auto mas parecido al que me vaya a comprar, en un lugar mas parecido al que me voy a mover.

O sea a lo que voy a la hora de elegir el auto (lenguaje) tiene mas sentido pensar en el uso que le voy a dar que en las características del auto (lenguaje) en si

Aprender a aprender

Bueno, ya cerrando esta breve intro general al tema de programación, lesuento que todo lo que aprendan este año y el año que viene, en 10/15 años no va a servir mas, o algo así

Me dejo de joder un rato con el tema de los autos y les pongo el ejemplo con el famoso y querido Excel. Imagínense que hace 20/25 años hacían un super curso de “planillas de cálculo”

A principios de los 90s el Excel era toda una novedad, en épocas donde se usaban mucho el MsDOS y no tanto los entornos gráficos como Windows 3.11 que recién se empezaban a popularizar, en esa época las principales planillas de cálculo eran las del programa Lotus 1-2-3 que corría sobre MsDOS que no tenía entorno gráfico como windows, se veía como una consola y se usaban atajos de teclado difíciles de recordar, no tenían ni la mitad de las funciones que hoy tiene el Excel

Excel - ENP									
File		Edit		Range		Copy		Move	
New		Open		Delete		Column		Erase	
Global		Insert		Delete		Column		Titles	
A		B		C		D		E	
1	Sales	EMP	NAME	DEPTNO	JOB	YEARS	SALARY	BONUS	TOTAL
2	3777	Azizah		4000	Sales	2	40000	10000	
3	81044	Brown		6000	Sales	3	45000	10000	
4	48378	Burns		6000	Mgr	4	75000	25000	
5	58706	Casper		7000	Mgr	3	65000	25000	
6	49692	Curly		3000	Mgr	5	65000	20000	
7	34791	Dabarett		7000	Sales	2	45000	10000	
8	84994	Daniels		1000	President	8	150000	100000	
9	59937	Dempsey		3000	Sales	3	40000	10000	
10	51515	Donovan		3000	Sales	2	30000	5000	
11	48338	Fields		4000	Mgr	5	70000	25000	
12	91574	Fiklore		1000	Admin	8	35000	---	
13	64596	Fine		5000	Mgr	3	75000	25000	
14	13729	Green		1000	Mgr	5	90000	25000	
15	55957	Hermann		4000	Sales	4	50000	10000	
16	31619	Hodgeson		5000	Sales	2	40000	10000	
17	1773	Howard		2000	Mgr	3	80000	25000	
18	2165	Hugh		1000	Admin	5	30000	---	
19	23907	Johnson		1000	VP	1	100000	50000	
20	7166	Lafaire		2000	Sales	2	35000	5000	

Ahora, si alguien aprendió a usar esas planillas de cálculo, seguramente se adaptó al Excel muy rápido porque aprendió a aprender.. y ¿a que me refiero entonces con aprender a aprender?

Me explico, seguramente muchos de ustedes hayan aprendido a usar Excel “probando” arrastrando fórmulas hasta que se dieron cuenta de usar el \$ para fijar una fila o columna o tipeando varias veces la misma fórmula hasta que descubrieron la función mágica “desref” o buscando valores a mano de otras tablas hasta que alguien les tiró el dato de usar “BuscarV” o “Index” y así.. o quizás hicieron un curso de Excel o quizás leyeron algún libro, pero la magia no está ni en el libro ni en el curso ni en nada de eso, la magia fue ir probando y las horas que estuvieron sentados frente a sus Excels probando para mejorar sus planillas.. Ojo lo mismo les

pasó a los que aprendieron con el lotus 1.2.3 es decir lo valioso que aprendieron no fueron las herramientas en si, sino que “aprendieron a aprender” como funciona una planilla de cálculo.

Espero me hayan seguido el concepto, lo importante cuando vayamos viendo las herramientas de python no son las funciones que veamos sino digamos “la esencia de los algoritmos” y esa esencia es pensar de modo sistemático, es realmente difícil describir esto con palabras, seguramente a medida que avancen en este mundo de la programación me van a ir entendiendo mejor este concepto

Por ahora solo quiero dejarles esta idea de que se relajen, disfruten del camino y no se preocupen porque se pasaron un día entero renegando con una sola línea de código, es muy común que pase eso, me pasó mil veces y me sigue pasando, pero cuando pasa eso no son horas perdidas para nada, porque en definitiva esa línea de código que no les salía es irrelevante, dentro de unos años no va a existir más ese comando o quizás ni se use python ya porque lo remplazará otro lenguaje más interesante vaya uno a saber, pero lo que sí sé, es que esas horas que perdieron con esa maldita línea de código les enseñó a aprender y aunque no se hayan dado cuenta los fue forjando para pensar sistemáticamente en “modo algorítmico” como suelo decir siempre.

El futuro de la programación

Voy cerrando con una pequeña reflexión personal, hoy se estima que hay unos 20 millones de programadores en el mundo, cifra que era insignificante hace 30 años, es una carrera que no deja de crecer en el mundo, y que incluso tiene desempleo 0, ya hace años que la demanda crece mucho más rápido de lo que las universidades forman a los nuevos profesionales lo que hace que muchos nunca lleguen a terminar sus carreras de grado, de hecho es casi inédito encontrar programadores con carreras de posgrado porque la demanda laboral es tan grande que hace irresistible para un estudiante de estas carreras abstraerse del mundo laboral para seguir capacitándose

Esto hizo que en los últimos 10 años surjan las famosas coding schools o bootcamps de programación que son como cursos muy prácticos e intensivos para formar programadores en tiempos records, ya que el mercado laboral los demanda como dije antes a mayor velocidad de lo que los forman las universidades tradicionales.

O sea que si pasamos de un número marginal de programadores en el mundo a 20 millones (de profesionales que viven de esto) en 30 años, imaginen ahora con el acceso a internet masivo, con la explosión de los cursos online, con la creciente demanda insatisfecha en casi todos los países del mundo de programadores, no es de extrañar que se proyecten unos 100 millones de programadores para 2025, 400 millones para 2030 y cerca de 1000 millones para 2035 o sea que en 15 años puede que más del 10% de la población haya adquirido capacitación en programación (no hablo de graduados universitarios)

A lo que voy con esto, imaginen en ese mundo de dentro de 15 años lo que va a influir herramientas como python en el trabajo cotidiano.

La programación y la era de la Big Data

Implícitamente dentro de la programación hay un campo de trabajo que no para de tomar relevancia que es el campo de los “data scientists”

Voy a dedicar los siguientes párrafos para responder al final la pregunta de ¿qué demonios es un data scientist?, ya que este es uno de los principales campos de aplicación del lenguaje que veremos en esta obra, que es Python.

Si hay algo que crece a un ritmo super descontrolado son los datos, la cantidad de datos diarios generados por nuestra actividad cada vez que hacemos algo con nuestro teléfono o nuestra compu es increíble, cada clic o no-clic es un dato que se guarda en algún lado y muchas veces son datos valiosos si se saben cruzar e interrelacionar, pero ahí justamente está la clave, el dato suelto no sirve, ahora si te dicen que ese clic:

“es un clic de fulano en un anuncio de tal marca que compite con tal otra en tal segmento de población y que el usuario tiene determinado perfil e hizo clic después de ver 4 anuncios de X competidores en los que no hizo clic en ninguno”

¿Ahí parece un dato mas valioso no?

La realidad es que tampoco nadie guarda los datos de esa forma, sino en bases de datos ordenadas y organizadas de tal forma que luego se puedan extraer para armar gráficos de correlación que permitan armar ese tipo de relatos de “fulanos genéricos”, y digo “fulanos” a propósito, dirigido a los conspiranoicos que siempre están pensando que Google les quiere leer la mente.

En realidad la big data no son datos nominados, no pasa por ahí su valor, imaginen que hoy solo la parte de internet que se ve, alberga más de 15 zettabytes, los zettabytes son mil hexabytes que a su vez son mil petabytes que a su vez son mil terabytes que a su vez son mil gibabytes, un disco rígido de su compu lleno de información puede tener 100 gibabytes de data pura si hablamos de fotos o videos, pero si son solo datos numéricos, booleanos (Verdadero o Falso) o de texto, raramente supere los 2 gigabytes toda la info que uno guarda. Un Zettabyte son 10^{21} bytes, es decir es un número con 21 ceros!

No parece tan grande? Veamos, supongamos que cada byte es un dato y que cada uno vale un dólar, en ese caso la fortuna de Jeff Bezos 100mil millones de dólares equivalen a 11 ceros, serían como 100 gigabytes como su disco rígido lleno de datos. El Sp500 todas las empresas juntas valen 30 millones de millones de dólares un número de 13 ceros el equivalente en bytes a 30 Terabytes

O sea que si con esta comparación de un dato un dólar, quisieramos alcanzar el valor de los datos de la internet visible necesitaríamos 500.000 veces las empresas del sp500, y les doy un dato mas, la internet visible es solo el 0,03% de la cantidad de datos de toda la internet profunda +privada +oculta.

Y pensemos una cosita más, esta cantidad infernal de datos se empezó a generar hace relativamente poco, piensen que Google existe hace solo 20 años, Facebook hace 15, es decir, en los próximos 15 años la cantidad de datos que generaremos será muchísimo mayor aún.

Volviendo al hilo de lo que quería decir, es irrelevante la comparación de 1 byte = 1 dólar obviamente, solo fue a modo didáctico y justamente lo que quería resaltar es la inmensidad del volumen de datos, pero ahí viene la siguiente pregunta *¿Qué vale mas, los datos o el análisis de los mismos?*

Ese análisis es lo que transforma los datos en información, y la calidad de ese análisis lo que transforma esa información en información útil. **Y a eso apunta la función de los data scientists.**

El futuro de los programadores

Digo todo esto, no como un incentivo a estudiar este tipo de carrera desde el punto de vista de perspectiva laboral, bien podría hacerlo y tengo infinidad de motivos, pero no es el punto al que quiero ir, lo que quiero remarcar es que todos estos programadores y científicos de datos, están dispersos en las áreas más heterogéneas que se pueden imaginar, desde la ciencia, el arte, el deporte, el ámbito administrativo la gestión pública, la banca, la industria logística, la salud, la manufactura, el ámbito educativo, los encuestadores, consultores, el marketing y la industria creativa, cinematográfica los videojuegos la industria del entretenimiento y podría dedicarle páginas enteras a la descripción.

¿Y a que voy con esto? A que hace 30 años todas esas industrias funcionaban perfectamente sin programadores ni científicos de datos agregando valor a sus actividades, y ahora ya ninguna es competitiva sin ellos.

¿Y? y que claramente la tendencia no se detiene pero recién estamos en muchos países en una etapa de “punto de inflexión” esos puntos en donde cambia la pendiente de la curva,

Lo que quiero decir que hasta ahora (+/- un par de años) dependiendo la industria, muchas empresas no se adaptaron aún a la disrupción tecnológica que significa la era de la big data, o recién están empezando tibiamente a dar los primeros pasos, pero dentro de muy poco cambiará muy rápido el escenario y con ello la forma de tomar las decisiones o de implementarlas o testearlas antes. Y hay que estar preparado para ello cada uno desde su lugar, no importa cual fuera, mientras mayor grado de responsabilidades mayor es la urgencia en entender esta disrupción mas allá de la técnica y las herramientas en si, y a menores escalas en la toma de decisiones mayor urgencia en entender las herramientas en si y aceptar la disrupción.

El futuro de la programación en el ámbito financiero

En nuestro pequeño mundillo de las finanzas el tema del análisis de datos, la big data, python, c++, machine learning y la “sea in car” (expresión tuitera de hace unos días q me hizo mucha gracia) es un hecho consumado. Quiero decir, es una de las actividades pioneras en adoptar esta disrupción

Y no hablo solo del “trading” y los bancos de inversión como muchos se imaginan, el HFT y todo ese tema, sino que todo este rollo ya está presente en infinidad de ámbitos de la industria financiera, tanto en la banca, como en los seguros, y ni hablar en el mundo de las Fintech que vienen a patear el tablero con la competitividad que les da esta disrupción para desplazar a la banca tradicional en infinidad de rubros.

En fin, el mundo de la programación llegó para quedarse, y no es una disciplina oscura y super técnica de “pibes medio raros”, surge mas que nada por el boom y potencia de los ordenadores combinados con el boom de los datos, combinado con la potencia del open source que les hablaba párrafos atrás. Todo junto está provocando una explosión de productividad en infinidad de tareas.

Créanme que el viaje de meterse en este mundo vale la pena, y si llegan a la conclusión que no lo vale es porque no es lo suyo y todo bien, pero seguramente les sume un montón porque ya nada va a volver a hacerse como se hacía antes de que estas herramientas sean tan masivas y populares.

Las herramientas estadísticas y matemáticas

Si bien esta serie de libros son libros de programación para aprender a programar, son necesarios, para muchas de las cosas puntuales de análisis cuantitativo que veremos, algunos conocimientos previos de análisis matemáticos, estadística descriptiva, álgebra, etc. Pensé en hacer un libro introductorio a estos temas como nivelación, pero preferí ir mechando de a poco a medida que vamos avanzando en la profundidad de los análisis este tipo de conocimientos.

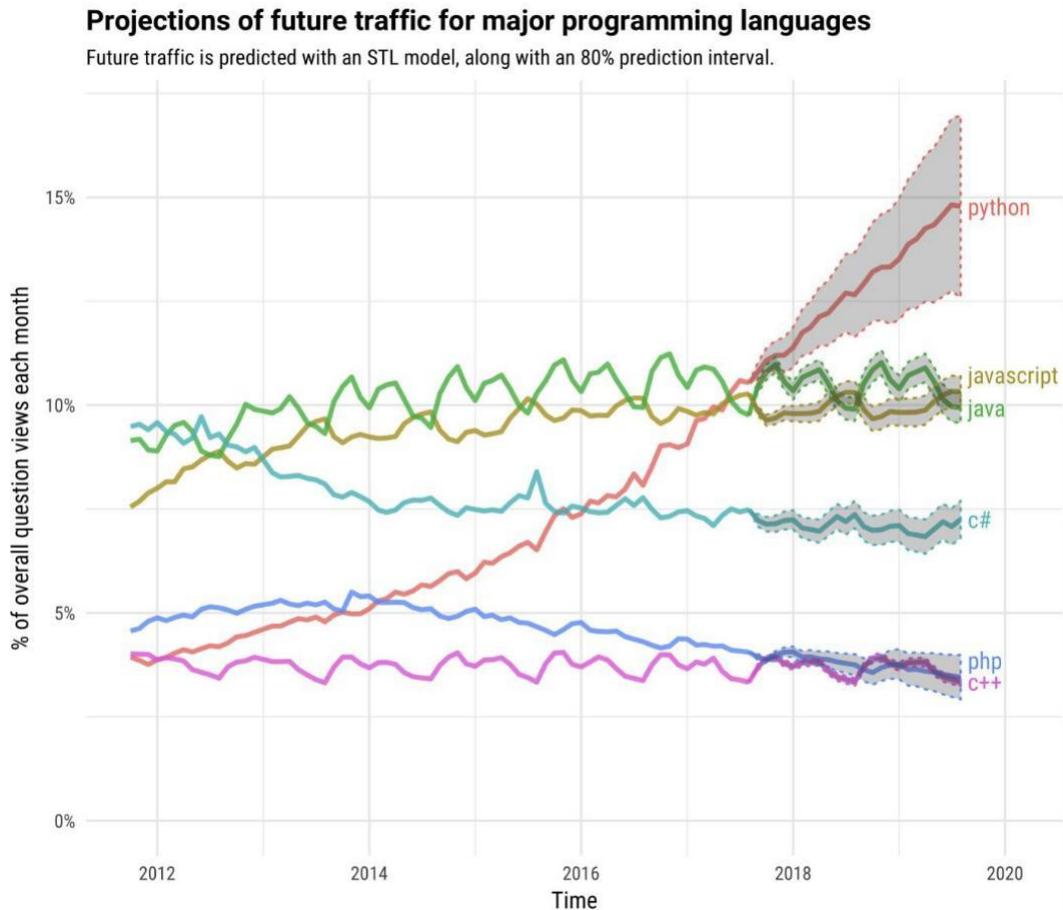
Me contuve de no poner mucha fórmula ni nada que no sea estrictamente necesario para presentar el tema, así que dejo solo el comentario aquí mencionado que luego del tomo 10 calculo, vendrá un libro específico donde me meta mas en profundidad en temas matemáticos y de modelización para los que están mas interesados en esa rama del análisis cuantitativo.

Por ahora los primeros tomos son para aprender Python desde cero, es decir son libros mas bien de programación y no de análisis cuantitativo, me han preguntado muchos programadores colegas que vienen mas del palo programación y les advertí que en los primeros tomos se van a morir de embole porque los pensé para gente que arranca de CERO en programación, de hecho, insisto, son libros para aprender a programar con el trasfondo de temas financieros como motivación si querés, pero el fuerte es aprender a programar.

No obstante, como les decía, a medida que se vayan desarrollando los tomos y se vaya poniendo interesante desde el punto de vista de las aplicaciones, vamos a ir viendo temas de estadística descriptiva, estadística inferencia, probabilidades, análisis matemático, álgebra lineal, etc, pero no se asusten, como les dije, no voy a usar mucho simbolismo, solo cuando sea super necesario y lo voy a explicar paso a paso para que nadie se quede relegado por una base débil en matemática.

¿Por qué Python?

En términos generales está de moda, es decir de todos los lenguajes para todos los usos, es el lenguaje del momento, esto quiere decir que si armamos algo, vamos a poder compartirlo con más gente, y a su vez vamos a poder ver más cosas compartidas por otros desarrolladores si usamos este lenguaje, esto incluye librerías, SDKs, ejemplitos, tutoriales, ayuda en solución de problemas, lo que en el ambiente llamamos "la comunidad"

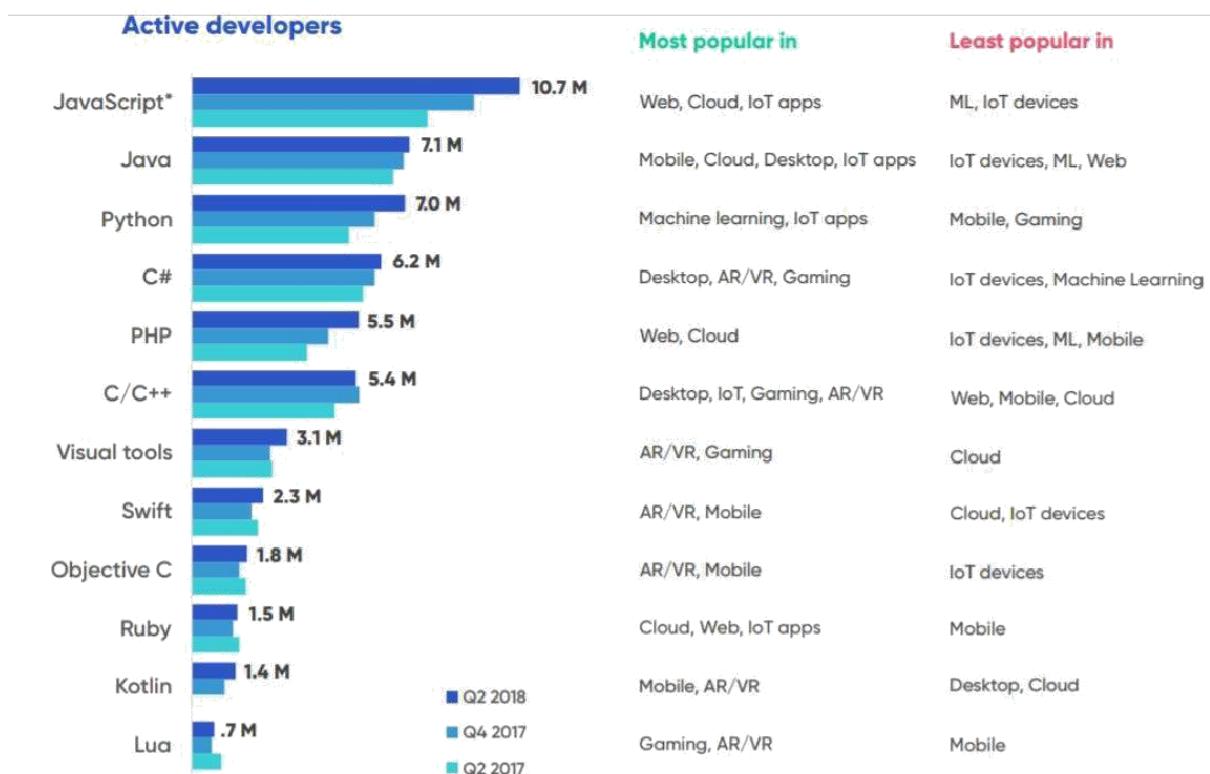


Por otro lado cada lenguaje tiene su uso más típico o digamos para lo que es ideal y usos para lo que no conviene ni pensar

Es como todo cada cosa tiene sus pro y sus contras y eso lo hace mas o menos útil para determinados usos específicos

En el cuadro de la página siguiente les muestro un resumen muy interesante publicado en el sitio stackoverflow.com (uno de los sitios mas consultados por la comunidad mundial de programadores) en base a su encuesta anual de lenguajes de programación

WORLDWIDE PROGRAMMING LANGUAGE STATISTICS



- Ejemplo, el lenguaje **C** es un lenguaje de bajo nivel, es decir, un lenguaje más cercano al hardware, por eso se los llama lenguaje de máquina, por lo tanto es utilizado en proyectos en los que se requiera más cercanía entre el código y el hardware, generalmente proyectos en donde la performance es vital (Por ejemplo HFT)

 Ojo que **C++** no es lo mismo que C, es una evolución del mismo, a su vez **C#** es una evolución de C++ desarrollada por Microsoft, y que compite con Java o sea que es de mucho más alto nivel
- Otro ejemplo de lenguaje muy utilizado es **Java** que es compilado, es el referente en programación orientada a objetos y muy utilizado para interactuar con todo tipo de dispositivos, es muy usado para apps nativas para mobile por ejemplo o para aplicaciones de escritorio.
- En cambio, **PHP** por ejemplo es el lenguaje del lado del servidor por excelencia para páginas y aplicaciones web, hoy en día se dice que el 80% de la web está hecha en PHP
- Sin embargo, **Javascript** está ganando terreno gracias a NodeJs que permite utilizar este lenguaje para el backend (del lado del servidor)
- Python**, es un lenguaje de muy alto nivel, es decir muy fácil para escribir código por la gran cantidad de librerías y código prescrito que tienen, su uso más típico es para Inteligencia artificial, machine learning, data science (en donde compite con R) pero también se usa para aplicaciones web en menor medida y es un lenguaje muy versátil y con una curva de aprendizaje muy amigable con lo cual es genial para aprender

De todos modos hay infinidad de lenguajes, todos tienen alguna ventaja y algún uso para los que son geniales y otros para los que no son lo ideal, la verdad que para aplicaciones financieras, backtesting, screenings, recolección da datos, etc es muy muy recomendado mas aun teniendo en cuenta que consta con una comunidad que no deja de crecer

Las 8 premisas a evaluar en un programa

Antes que nada tiro unos tips muy importantes:

- Siempre pero siempre hay una solución al problema planteado
- Siempre que hay una solución a un algoritmo hay muchas maneras de llegar al mismo resultado
- Cuando el programa no funciona no es nunca la máquina, mucho menos una conspiración mundial, somos nosotros que pensamos mal la lógica

Dicho esto, como siempre hay muchas maneras de llegar al mismo resultado, es decir que no hay una solución única, lo importante en programación no va a ser llegar a la solución correcta del problema, sino llegar de la forma mas razonable posible dependiendo de las necesidades del caso, para exemplificar veamos las premisas con las que siempre vamos a lidiar

1. Confiabilidad: ¿Resuelve siempre sin fallas? ¿De qué depende? ¿Cuál es el % de fallas?
2. Testeabilidad: ¿Hay maneras de probar el programa antes de sacarlo a producción?
3. Performance: ¿Se podría hacer que sea más rápido ejecutándose o que use menos recursos de memoria/procesador?
4. Usabilidad: ¿Es fácil de usar para un nuevo usuario o para mi mismo?
5. Mantenibilidad: ¿Que tan sencillo es realizarle cambios? ¿qué tan prolífico y legible queda el código?
6. Escalabilidad: ¿Como se comportaría si crece en cantidad de datos, de usuarios, de instrumentos?
7. Portabilidad: ¿Me sirve para otro mercado, instrumento, horario, etc?
8. Seguridad: ¿Está protegido de ataques?

Siempre, pero siempre habrá muchas maneras de resolver un problema, y ni hablar en programación, las variantes son infinitas diría, pero también siempre habrá limitaciones y prioridades, de esas 8 premisas nunca jamás se puede ser óptimo en todo, siempre que se elige una prioridad de las 8 se resigna otra (en cierto grado obviamente) por lo tanto es muy pero muy importante siempre pensar antes que nada cual va a ser la prioridad que vamos a tomar

Por ejemplo la **confiabilidad** va a ser la prioridad número 1 en un bot de **backtesting** de un método de trading, pero si quisiera un programa para empezar a **probar una idea**, probablemente no me interese tanto el fino de los datos y los resultados sino que pensaría primero en la **testeabilidad** porque para ser más riguroso con los datos hay tiempo luego en la etapa de producción pero en una primera etapa necesito ser más flexible a probar muchos cambios y soportar errores típicos de una primera etapa de prueba

Si estoy haciendo un bot de **screening** en tiempo real, donde la cantidad de datos es infernal y la velocidad de procesamiento es crucial, ahí la prioridad número 1 va a ser la **performance** ya que al ser un screening puedo dejar para un segundo procesamiento la responsabilidad de la confiabilidad

En fin, son solo ejemplos para mostrar que antes de la primera línea vamos a tener que tener este tipo de cosas presentes

Diferentes entornos para programar

Podemos agrupar a las herramientas para escribir código en grupos, no se vuelvan locos por ahora con esto, lo menciono para ir metiéndose en tema, pero como diré mas adelante vamos a concentrarnos en principio en los Jupyter Notebooks porque me parecen mucho mas amigables para quienes recién arrancan a programar

- Consola de Python: Es la forma más rápida de probar un script, pero terriblemente incómoda, más que nada se usa para ejecutar un programa guardado previamente.
- Jupyter Notebooks: Vienen en Anaconda (que es el único software que instalaremos), son piolas para ir probando código “línea por línea” y son ideales para principiantes por muchas ventajas que ya veremos.
- Google Colab: Son como los Jupyter Notebooks pero online, es decir como una bandeja de emails pero en lugar de emails almacena “cuadernos con código Python” y lo genial es que se puede compartir muy fácilmente
- Intérpretes online: Ej repl.it es interesante para cuando no tenemos nuestra compu y queremos probar pequeñas cosas.
- Editores de código: Serían como un Word pero para codear, algunos ejemplos serían:
 - SublimeText
 - Atom
 - Geany.
- IDEs: Son entornos de desarrollo integrados (integran consola + editor de texto etc), ejemplos:
 - Spyder
 - Visual Studio Code
 - Pycharm

Como les anticipaba, nos vamos a enfocar mas que nada en el uso de los “Jupyter Notebooks” La principal ventaja de los **Jupyter Notebooks** es que combinan anotaciones con texto enriquecido y widgets con código lo que los hace muy piolas para lo didáctico, a su vez otra gran ventaja es que permiten ir ejecutando código línea a línea o celda a celda y reiniciar el kernel solo cuando haga falta.

Por otro lado, en los **intérpretes online** la principal ventaja es que no tenemos que instalar nada, simplemente abrimos una web y listo y si nos registramos podemos dejar guardados nuestros proyectos en carpetas. Obviamente que estas soluciones online no sirven mucho para grandes proyectos o para proyectos con datos privados sensibles, pero para tener algunas cositas siempre a mano vienen muy bien

Por el lado de los **editores de código**, a mi gusto es lo mejor para el programador profesional porque tienen todo tipo de facilitadores, atajos de teclado, funciones de autocompletado etc, lo que acelera la productividad mucho y son super livianos y rápidos, el tema es que no tienen consola ni salidas integradas en el editor, es decir, son simplemente “editores de texto” especiales para programadores pero no tienen salida, es decir, tenés que codear “a ciegas” y luego ir a una consola a ejecutar tu script “a ver que hace”. Obviamente para el programador que ya tiene cierta experiencia es lo ideal, ya que son editores livianos, y sobre todo muy cómodos y customizables y con las mejores herramientas de atajos de teclado y demás. Pero para el principiante puede ser medio tedioso usar un editor de código pelado para aprender, así que si están empezando les recomendaría no arrancar intentando usar este tipo de herramientas al principio.

Por último, tenemos los **IDE** que son **entornos de desarrollo** completos, donde tenemos además del editor de código con todas sus funciones, una consola integrada y la mayoría también cuenta con herramientas de debug más avanzadas para ir viendo el paso a paso de nuestra ejecución si lo quisiéramos, suelen ser programas bastante más pesados que los editores de código, y enlentecen un poco la ejecución. Son super completos y tienen integraciones con muchas herramientas de la programación que veremos mas adelante como github y como les decía tienen una consola integrada que al ejecutar en el mismo editor el programa nos “da una salida” en pantalla del script, cosa que los editores de código no tienen

Bueno, quería hacerles un repaso por todas estas formas porque entiendo que al principio cuando uno arranca hay tanta información y parece todo tan abrumador, que uno no sabe por donde empezar, y muchas veces ven tutoriales de un programador con experiencia que usa una herramienta y a lo mejor no es para nada lo ideal para alguien que recién empieza

Dicho todo esto, asumiendo que la mayoría de lectores arrancan a programar desde cero, **recomiendo enfáticamente la instalación del paquete Anaconda**. Es el único software que hay que instalar

La principal ventaja es que con una sola instalación ya tenemos de una:

- Python 3
- Un prompt en la carpeta de instalación para instalar al toque las librerías externas
- Spyder que es un IDE muy completo y práctico
- Jupyter Notebooks (cuadernos para codear paso a paso con anotaciones enriquecidas)
- Las librerías más usadas como pandas, numpy, matplotlib etc

Si ya tienen instalado algún IDE y se manejan bien con ello, está perfecto, no hace falta seguir al pie de la letra todo, como dije usaré siempre en los ejemplos el Jupyter Notebook tal como viene en Anaconda pero si siguen el código con otro entorno van a poder hacer todos los mismos ejemplos sin problema

Instalaciones

No tengo mucho para agregar respecto a la instalación, al momento de escribir este libro la ultima versión de Python es 3.7 pero si tienen la 3.8, o la 3.9 o cualquier versión superior al momento de instalarlo ustedes mejor.

Sigan el paso a paso tal como sugiere el instalador, si en algún momento les pregunta si desean agregar anaconda como variable de entorno, no hace falta y podría complicar asi que no lo tilden esa opción.

Tips de instalación:

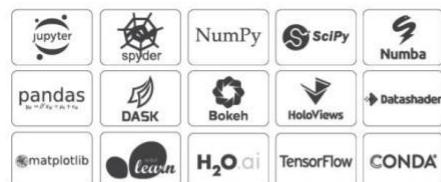
- Descargarlo de www.anaconda.com
- Instalar la versión con Python mas nueva que les ofrezcan (hoy es 3.7 pero si tienen una mayor, mejor)
- No cambien la carpeta default de instalación
- Es recomendable que la ruta de instalación no tenga directorio con acentos, eñe u otro caracter no estándar
- Si lo instalan en una notebook con un firewal con muchas limitaciones pueden tener errores de instalación
- Requerimientos: Con una notebook cualquiera de 2Gb de RAM, medio pelo año 2018 en adelante alcanza perfecto

Puede tardar un lindo rato la instalación, así que paciencia, si tira algún error de instalación, desinstalen y vuelvan a instalar, de todos modos si por alguna razón les explota la compu y no hay caso con instalar esto, no pasa nada, después les comento la alternativa para poder aprender sin tener que instalar nada de nada en su compu.

Capturas de pantalla de la página oficial de anaconda

The open-source [Anaconda Distribution](#) is the easiest way to perform Python/R data science and machine learning on Linux, Windows, and Mac OS X. With over 19 million users worldwide, it is the industry standard for developing, testing, and training on a single machine, enabling *individual data scientists* to:

- Quickly download 7,500+ Python/R data science packages
- Manage libraries, dependencies, and environments with Conda
- Develop and train machine learning and deep learning models with scikit-learn, TensorFlow, and Theano
- Analyze data scalability and performance with Dask, NumPy, pandas, and Numba
- Visualize results with Matplotlib, Bokeh, Datasader, and Holoviews



Windows | macOS | Linux

Anaconda 2019.10 for Windows Installer

Python 3.7 version

[Download](#)

64-Bit Graphical Installer (462 MB)
32-Bit Graphical Installer (410 MB)

Python 2.7 version

[Download](#)

64-Bit Graphical Installer (413 MB)
32-Bit Graphical Installer (356 MB)

Primer Hola Mundo en diferentes entornos

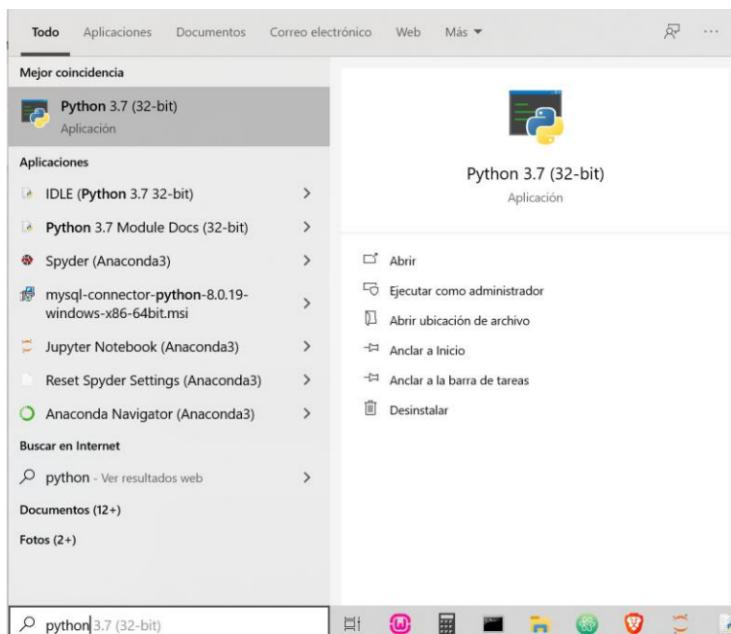
Bueno, el bautismo al entrar en el mundo de la programación es hacer el primero “hola mundo”, esto es realizar un programa hiper sencillo que lo único que hace es imprimir un mensaje en pantalla que por lo general dice “hola mundo”, así que vamos a hacer esto

Consola básica de python

Arranquemos con la consola pelada de python, buscamos python en el buscador de programas de windows y lo ejecutamos

Una vez abierto python ponemos la instrucción y le damos enter para ejecutarla

La verdad que va a ser muy raro que ejecuten código directo desde la consola de python pelada, yo lo he usado solo para hacer alguna prueba cada tanto



Como ven se abre una consola d Python, con fondo negro, me da un cursor para escribir código con los símbolos

>>>

Luego de esos símbolos yo puedo escribir los comandos, en este caso imprimo el comando **print()**

Como todo comando dentro de los paréntesis llevará los argumentos, en este caso el comando print tiene varios argumentos, pero el principal y único obligatorio, que es el primero que le pasamos es el texto a imprimir

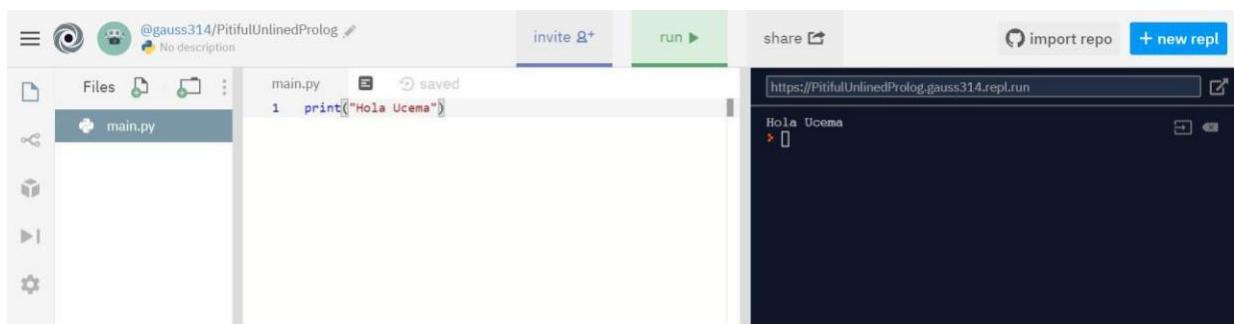
A screenshot of a terminal window titled "Python 3.7 (32-bit)". The window displays the Python 3.7.0 startup message and a command-line session. The user has typed the command ">>> print('Hola UCEMA')". The output shows the string "Hola UCEMA" printed to the screen.

IDE con intérprete online

Ahora vamos a un intérprete online

Usemos www.repl.it Vamos a la web, ponemos nuevo replit, elegimos python y se abre el intérprete online Es un repo gratuito, si se registran pueden guardar sus códigos, es muy útil para usarlo en computadoras donde no tienen anaconda ni python instalado.

También es muy útil para cuando quieran probar librerías desconocidas y pesadas, para no andar instalando nada en su compu prueban en estos servicios online y se sacan la duda si vale la pena para instalarlo o no



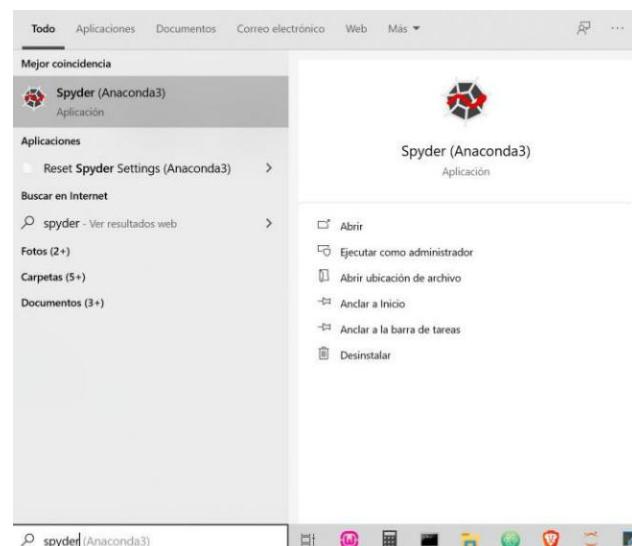
Spyder

Es el IDE que viene instalado en anaconda, es muy práctico y recomendable para principiantes ya que otros IDEs como VisualStudio si bien es mucho mas completo, tienen algunas cositas de configuración un poco mas tediosas, así que para arrancar como primer IDE esta bueno usar Spyder

¿Conviene usar spyder o un IDE para programar?

Como les decía antes, prefiero que arranquen con Jupyter Notebook, mas que nada si arrancan de cero, ya van a ver que es muy práctico y además es 100% integrable con soluciones en línea como Google Colab, que si tienen problemas de software no pasa nada porque no dependen de su compu para que el código les ande siempre.

Para abrirlo buscamos el programa en la barra de Windows y lo ejecutamos



Una vez abierto ejecutamos el comando de nuestro “Hola mundo”

Como verán en el IDE tenemos dos partes, una parte a la izquierda y una a la derecha

Si ejecutan el código en la parte derecha al darle enter se ejecuta, es lo mismo que ejecutarlo en una consola

Pero si lo escriben a la izquierda al darle enter es como si estuviesen escribiendo un texto en word, no se va a ejecutar solo, para que se ejecute le tienen que apretar F5, o “Shift + Enter” (o darle al botón de play de la barra de botones)

The screenshot shows the Spyder IDE interface. On the left, there is a code editor window titled 'temp.py*' containing the following Python code:

```
1 print("Hola Ucema")  
2  
In [1]: print("Hola Ucema")  
Hola Ucema  
In [2]:
```

On the right, there is a terminal window titled 'Terminal 1/A' showing the output of the code execution:

```
Python 3.7.5 (default, Oct 31 2019, 15:18:51) [MSC v.1916 64 bit (AMD64)]  
Type "copyright", "credits" or "license" for more information.  
IPython 7.10.2 -- An enhanced Interactive Python.  
In [1]: print("Hola Ucema")  
Hola Ucema  
In [2]:
```

The status bar at the bottom indicates the session is 'conda: base (Python 3.7.5)'.

El Spyder es un IDE bastante completo, lo que nos permite no solo editar el código completo línea por línea cómodamente como si escribiesen un archivo de word, sino que tiene resaltado de texto en colores, un buen helper en tiempo real de errores de sintaxis, una consola a la derecha para ir viendo el resultado del código escrito, y muchas cosas más que serán útiles más adelante como inspección de variables y esas cosas

The screenshot shows the Spyder IDE interface with syntax highlighting applied to the code in the editor. The code editor window now contains:

```
1 print("Hola UCEMA desde Editor")  
2  
In [1]: print("Hola Ucema")  
Hola Ucema  
In [2]: runfile('C:/Users/floda/.spyder-py3/temp.py', wdir='C:/Users/floda/.spyder-py3')  
Hola UCEMA desde Editor  
In [3]:
```

The terminal window shows the same output as before, but the code in the editor is highlighted in green and blue. The status bar at the bottom indicates the session is 'conda: base (Python 3.7.5)'.

Por ejemplo, si escribimos mal el comando “print” veremos que a la izquierda en donde está el número de línea ya me muestra con una cruz roja que algo mal escribimos antes de ejecutar el código

The screenshot shows the Spyder IDE interface. In the code editor, line 3 contains the incorrect command `prin t("Hola UCEMA con error")`. A red cross mark is placed before the opening parenthesis of this line. To the right, the terminal window shows the execution of the script, where line 3 is highlighted in purple and the output shows the correct command being run.

```

Spyder (Python 3.7)
Archivo Editar Buscar Código fuente Ejecutar Depurar Terminales Proyectos Herramientas Ver Ayuda
C:\Users\floda\spyder-py3\temp.py 00:09:00
temp.py*
1 print("Hola UCEMA desde Editor")
2
3 prin t("Hola UCEMA con error")
4
Terminal 1/A
Python 3.7.5 (default, Oct 31 2019, 15:18:51) [MSC v.1916 64 bit (AMD64)
Type "copyright", "credits" or "license" for more information.
IPython 7.10.2 -- An enhanced Interactive Python.

In [1]: print("Hola Ucema")
Hola Ucema

In [2]: runfile('C:/Users/floda/.spyder-py3/temp.py', wdir='C:/Users/floda/.spyder-py3')
Hola UCEMA desde Editor

In [3]:

```

Por último, veamos los helpers, estos son ayudas que nos da el IDE de cada función, si se posicionan con el mouse encima del comando (en este caso encima de print) nos va a desplegar una ayuda super valiosa de todos los argumentos y sus tipos que admite la función

The screenshot shows the Spyder IDE interface with the cursor hovering over the `print` keyword in the code editor. A tooltip window is displayed, providing detailed documentation for the `print` function, including its definition, parameters, and a detailed explanation of each parameter's behavior.

```

Spyder (Python 3.7)
Archivo Editar Buscar Código fuente Ejecutar Depurar Terminales Proyectos Herramientas Ver Ayuda
C:\Users\floda\spyder-py3\temp.py 00:12:21
temp.py*
1 print("Hola UCEMA desde Editor")
2
3 print("Hola UCEMA con error")
4 print(*values: object, sep: Text=..., end: Text=..., file: Optional[_Writer]=..., flush: bool=...) -> None
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
Prints the values to a stream, or to sys.stdout by default. Optional keyword arguments: file: a file-like object (stream); defaults to the current sys.stdout. sep: string between values, default a space. end: string appended after the last value, default a newline. flush: whether to forcibly flush the stream.

Terminal 1/A
Python 3.7.5 (default, Oct 31 2019, 15:18:51) [MSC v.1916 64 bit (AMD64)
Type "copyright", "credits" or "license" for more information.
IPython 7.10.2 -- An enhanced Interactive Python.

In [1]: print("Hola Ucema")
Hola Ucema

In [2]: runfile('C:/Users/floda/.spyder-py3/temp.py', wdir='C:/Users/floda/.spyder-py3')
Hola UCEMA desde Editor

In [3]:

```

Otra cosa que tiene spyder es que se puede configurar temas oscuros para no dañar la vista cuando estamos muchas horas frente a la pantalla

Para ello van a al menú herramientas

=> Preferencias

=> Apariencia

=> Tema de interfaz: Obviamente como verán en "preferencias" se puede configurar de todo

The screenshot shows the Spyder IDE interface with a dark theme applied. The cursor is hovering over the `print` keyword in the code editor, displaying the same detailed tooltip documentation as the previous screenshot, but with a dark background.

```

Spyder (Python 3.7)
Archivo Editar Buscar Código fuente Ejecutar Depurar Terminales Proyectos Herramientas Ver Ayuda
C:\Users\floda\spyder-py3\temp.py 00:06:12
temp.py*
1 print("Hola UCEMA desde Editor")
2
3 print("Hola UCEMA con error")
4 print(*values: object, sep: Text=..., end: Text=..., file: Optional[_Writer]=..., flush: bool=...) -> None
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
Prints the values to a stream, or to sys.stdout by default. Optional keyword arguments: file: a file-like object (stream); defaults to the current sys.stdout. sep: string between values, default a space. end: string appended after the last value, default a newline. flush: whether to forcibly flush the stream.

Terminal 1/A
Python 3.7.5 (default, Oct 31 2019, 15:18:51) [MSC v.1916 64 bit (AMD64)
Type "copyright", "credits" or "license" for more information.
IPython 7.10.2 -- An enhanced Interactive Python.

In [1]: print("Hola Ucema")
Hola Ucema

In [2]: runfile('C:/Users/floda/.spyder-py3/temp.py', wdir='C:/Users/floda/.spyder-py3')
Hola UCEMA desde Editor

In [3]:

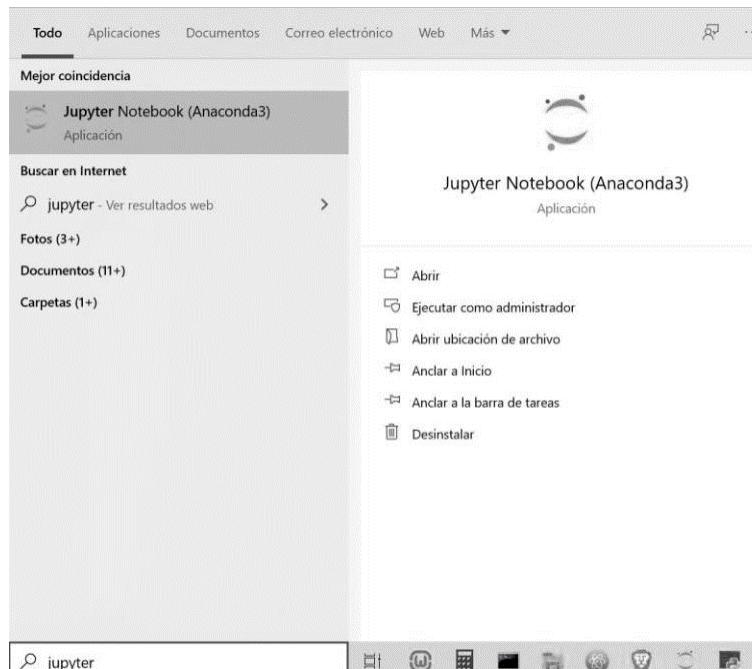
```

Jupyter Notebooks

Los notebooks (cuadernos), son herramientas excelentes para aprender ya que permiten mezclar código con anotaciones y tienen una ventaja muy interesante respecto al resto de opciones que es que nos permiten tener mucho código en el mismo archivo y eso, pero con la particularidad de permitirnos ejecutar de a celdas separadas

Para abrir un Jupyter Notebook, lo buscamos en la barra de windows y vemos que es como un navegador de carpetas y archivos

Ojo, al ejecutarse en un navegador puede que se les abra la primera vez una ventana que les pida confirmación de que navegador prefieren usar (Chrome, Firefox, Brave, etc)



Lo navegamos como si fuera el explorador de archivos de windows, y vamos a la carpeta que queramos o creamos una nueva y de ahí gestionamos nuestros archivos y proyectos, para ello vamos al botón **New** y elegimos python3 para crear un nuevo notebook de python.

The screenshot shows the Jupyter Notebook interface. At the top, there's a header with the logo and navigation buttons for 'Files', 'Running', and 'Clusters'. Below this is a toolbar with buttons for 'Upload', 'New' (which is circled in red), and 'Logout'. The main area is a file browser titled 'Select items to perform actions on them.' It shows a list of directories and files in the current directory ('/'). The columns include 'Name', 'Last Modified', and 'File size'. The list includes entries like '3D Objects', 'Anaconda3', 'Application Data', 'Contacts', 'Desktop', 'Documents', 'Downloads', 'Dropbox', 'Favorites', 'Links', 'Microsoft', 'Music', and 'OneDrive'. Most entries were modified 'a month ago', except for 'Downloads' which was modified '2 hours ago'.

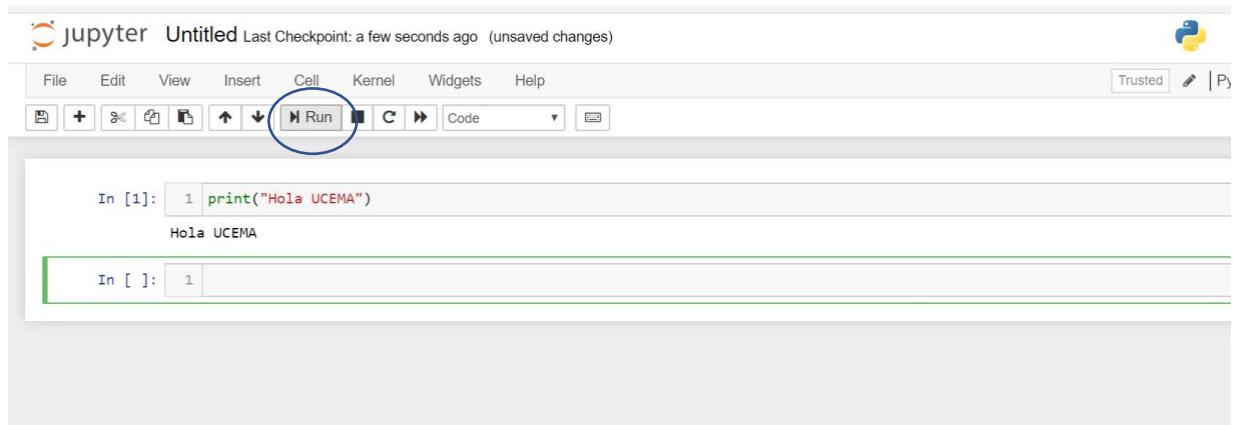
Name	Last Modified	File size
3D Objects	a month ago	
Anaconda3	4 days ago	
Application Data	a year ago	
Contacts	a month ago	
Desktop	an hour ago	
Documents	2 years ago	
Downloads	2 hours ago	
Dropbox	a year ago	
Favorites	a month ago	
Links	a month ago	
Microsoft	4 months ago	
Music	a month ago	
OneDrive	14 days ago	

Nota aparte, si quisieran pueden instalar R en anaconda y crear también cuadernos de R

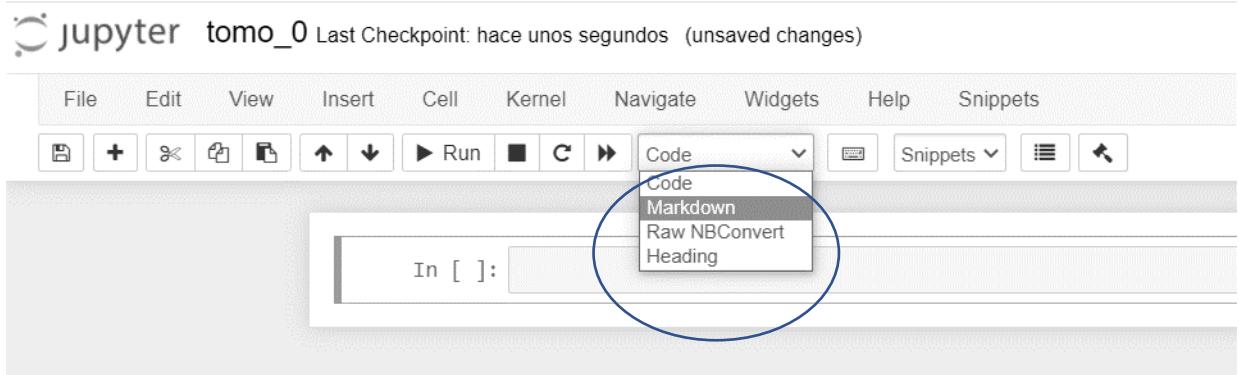
Una vez creado un nuevo Notebook, vemos que tenemos celdas

Las celdas permiten la cantidad de líneas de código que quieran y al ejecutarla se ejecuta todo lo que esté dentro de la celda, para ejecutarlo podemos apretar el botón **Run** o bien las teclas:

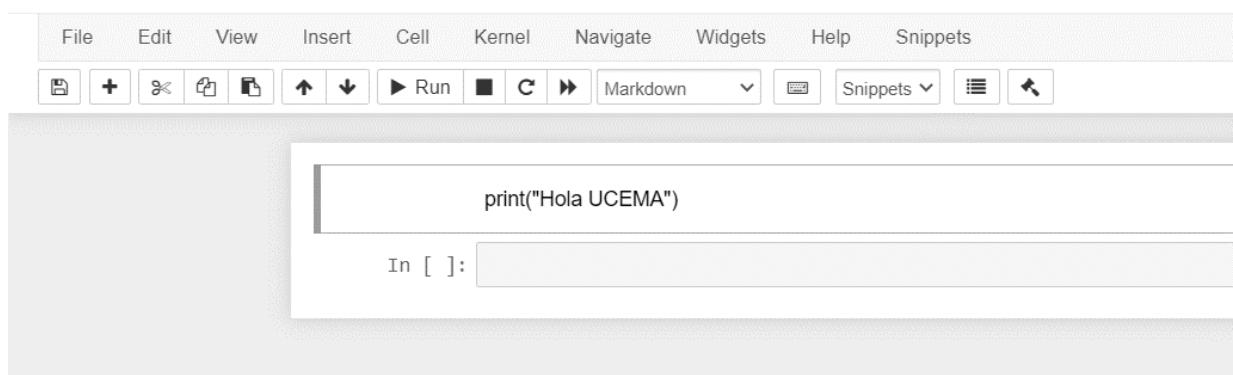
Shift + Enter



Si quieren escribir texto o anotaciones, simplemente seleccionan **Markdown** del desplegable resaltado en lugar de **Code**



Y cuando ejecuten simplemente quedará el texto como “estampado” y no ejecuta la impresión o el comando en sí



Respecto al markdown es un estándar para texto enriquecido muy útil y sencillo de usar, googleen si quieren profundizar, acá les dejo lo que más sirve

Si saben html también acepta HTML los notebooks de jupyter

Esto es como se tipea
en la celda:

```
1 Esto es un texto asi que no se va a ejecutar nada solo me sirve como ayuda
2
3 Puedo usar el estandar markdown para texto enriquecido
4
5 entre guiones bajos es cursiva
6
7 **entre doble asterisco es negrita**
8
9 **Puedo combinar ambas cosas**
10
11 Citas
12 > Esto es una cita
13
14 Listados
15 * Item 1
16 * Item 2
17
18 Links:
19 [Texto del link](http://google.com)
20
21 Encabezados o Títulos
22
23 # H1 - Título principal
24 ## H2 Segundo orden
25 ### H3 Tercer Orden
26 ....
27 ##### H6 Sexto Orden
28
```

Así es como queda impreso
el texto enriquecido con **Shift**
+ Enter

Esto es un texto asi que no se va a ejecutar nada solo me sirve como ayuda

Puedo usar el estandar markdown para texto enriquecido

entre guiones bajos es cursiva

entre doble asterisco es negrita

Puedo combinar ambas cosas

Citas

Esto es una cita

Listados

- Item 1
- Item 2

Links: Texto del link

Encabezados o Títulos

H1 - Título principal

H2 Segundo orden

H3 Tercer Orden

....

H6 Sexto Orden

Google Colab

Este es igual que los Jupyter notebooks pero online

Ventajas respecto a los Jupyter Notebooks

- No hace falta instalar nada
- No consume recursos de nuestra PC (Memoria/disco)
- Con un click podés compartir a quien quieras todo tu notebook
- Se puede elegir que use CPU, GPU o TPU, estos últimos son ideales para trabajos pesados de deep learning ya que están optimizados para calculo matricial, por eso son tan buenas las placas de video (GPUs) para minar cryptomonedas

Desventajas respecto a los Jupyter Notebooks

- Si no te anda internet fuiste
- Estas en las manos de Google, si un día deja de andar, perdes tus archivos porque no estan en tu compu (obvio que se pueden descargar igualmente)
- Para scripts muy simples es medio lento porque cada ejecución es un request http (ya veremos esto mas adelante en otros tomos)
- Para cosas con niveles de seguridad sensibles (claves de brokers y esas cosas) no es recomendable

Dicho esto, el entorno es casi igual al de un Jupyter notebook, solo necesitás tener una cuenta en Google

Primeros pasos en Google Colab

Lo primero es entrar al notebook de bienvenida donde te dan algunos ejemplos tutoriales etc:

<https://colab.research.google.com/notebooks/welcome.ipynb>

The screenshot shows the Google Colab interface. At the top, there's a navigation bar with 'Archivo', 'Editar', 'Ver', 'Insertar', 'Entorno de ejecución', 'Herramientas', and 'Ayuda'. On the right, there are buttons for 'Compartir', 'Configuración', and 'Drive'. Below the bar, there's a sidebar titled 'Índice' with links to 'Primeros pasos', 'Ciencia de datos', 'Aprendizaje automático', 'Más recursos', 'Ejemplos de aprendizaje automático', and 'Sección'. The main content area has a title '¿Qué es Colaboratory?'. It explains what Colab is and its advantages. It also mentions that Colab can facilitate work for students, scientists, and AI researchers. Below this, there's a section titled 'Primeros pasos' with a note about the document being interactive and a code cell showing a calculation of seconds in a day.

```
[ ] seconds_in_a_day = 24 * 60 * 60
seconds_in_a_day
86400
```

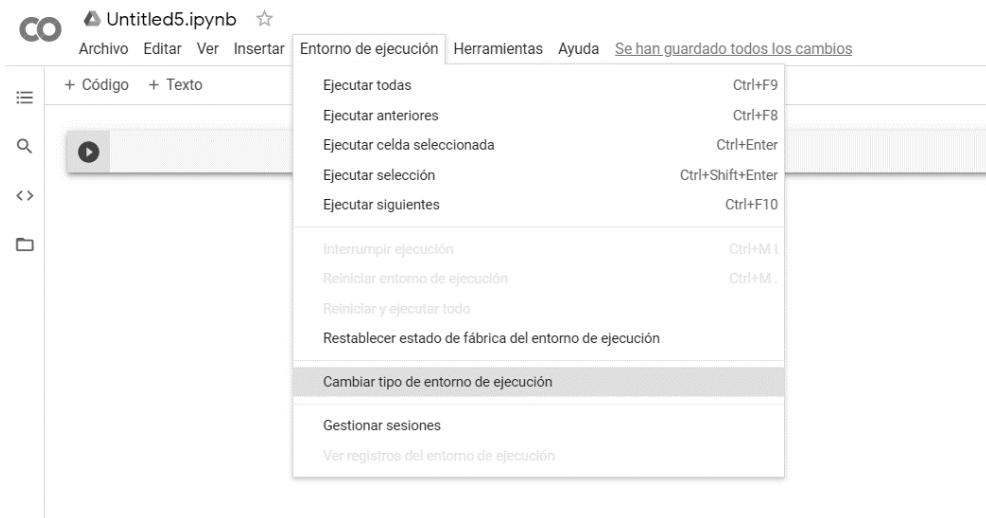
At the bottom, there's a note about executing code and defining variables.

Una vez en esa pantalla vas a archivo >> nuevo cuaderno

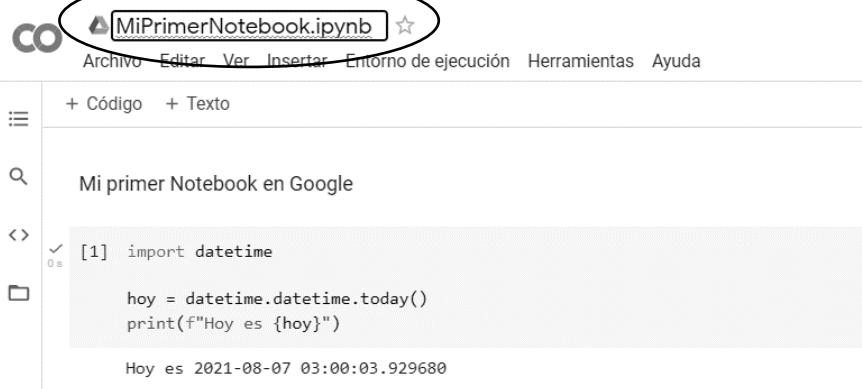
Es muy intuitivo, como cualquier programa, luego podes ir guardando, cambiar nombre, etc..

Se va autoguardando solo en google, pero si querés podés guardarte una copia en tu googleDrive o en Github, ya veremos GitHub en tomos mas adelante.

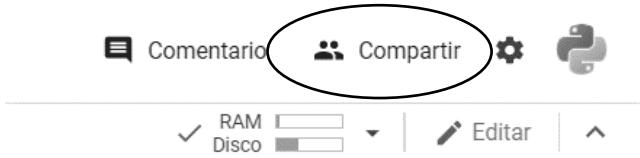
Una de las cosas mas importantes es el entorno de ejecución, se puede elegir GPUs como les decía antes



Como tips, por ejemplo, haciendo clic en el nombre lo editas a un click



Y a la derecha de todo tenemos el botón de compartir:

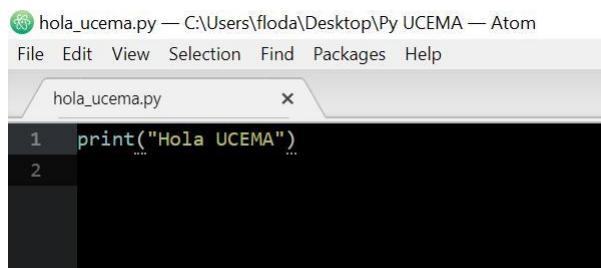


Y como vemos, tenemos un medidor de la RAM y utilización de disco de nuestros scripts entre otras cosas

Editores de código, Atom, SublimeText

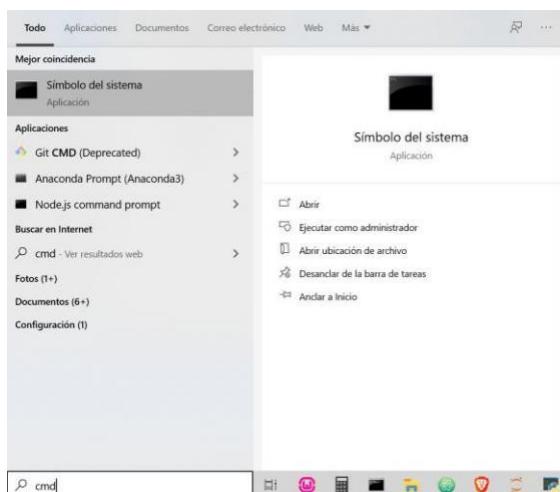
Para probar un script super sencillo este método es más engorroso, pero la realidad es que para proyectos grandes es más cómodo, no hay nada más cómodo que un buen editor de código, pero bueno, por ahora no vamos a usar este método, pero este es el lugar donde tenía que mostrarlo

Primero abro el editor de código Guardo un archivo hola_ucema.py
En el hola_ucema.py escribo el código



```
hola_ucema.py — C:\Users\floida\Desktop\Py UCEMA — Atom
File Edit View Selection Find Packages Help
hola_ucema.py x
1 print("Hola UCEMA")
2
```

Luego abro la consola buscando CMD

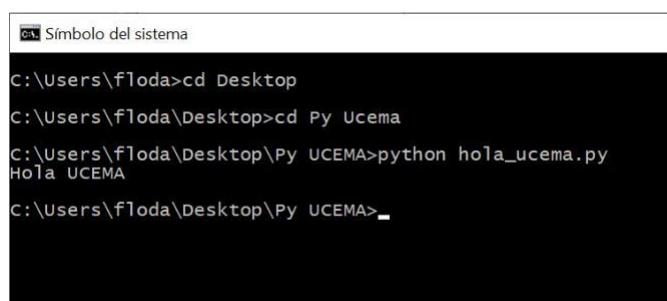


Navego por la consola hasta el directorio del archivo el comando **cd nombre_directorio** pasa al directorio llamado "nombre_directorio" mientras que el comando **cd..** vuelve atrás un nivel desde el directorio donde se encuentre parado

Una vez en el directorio, lo ejecuto

Para ejecutarlo pongo:

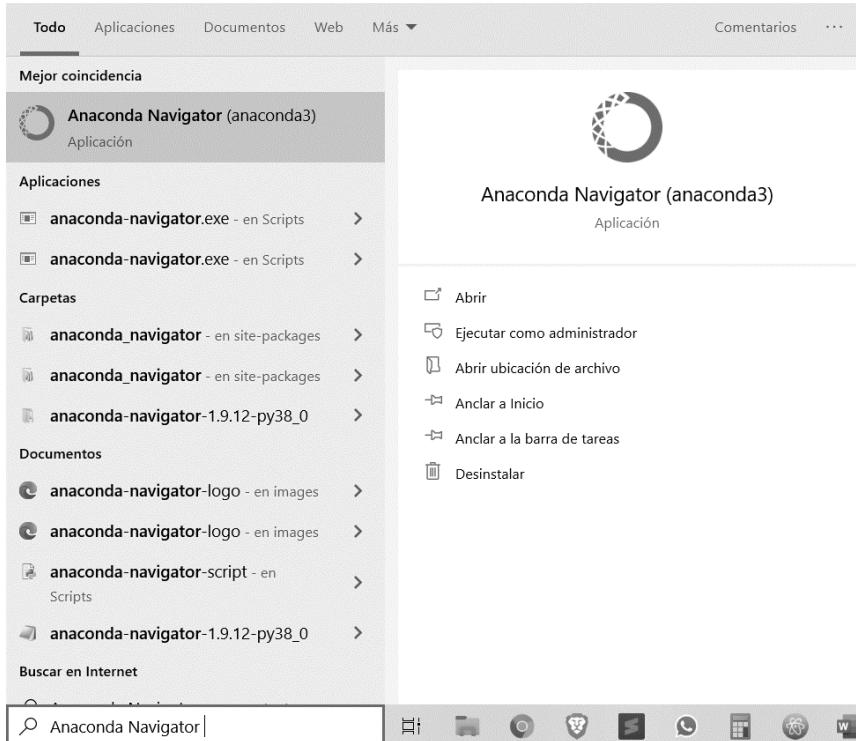
python "nombre_del_archivo.py"



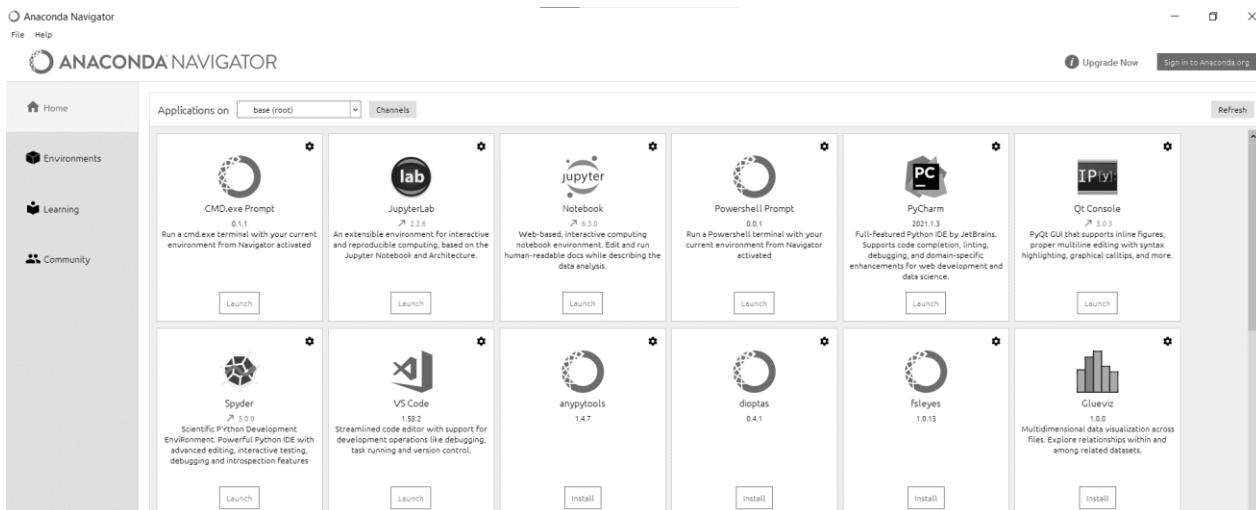
```
C:\ Símbolo del sistema
c:\Users\floida>cd Desktop
c:\Users\floida\Desktop>cd Py UCEMA
c:\Users\floida\Desktop\Py UCEMA>python hola_ucema.py
Hola UCEMA
c:\Users\floida\Desktop\Py UCEMA>
```

Anaconda Navigator

El paquete Anaconda nos provee de un manejador de aplicaciones y entornos super completo (es lento pero muy completo) que es el Anaconda Navigator, lo buscan así:



Se les va a abrir algo así:



Desde ahí pueden lanzar o ejecutar cualquiera de esas aplicaciones, ya vimos Spyder y Jupyter Notebook, pero como ven tiene integrados entornos para ejecutar PyCharm y VisualStudio entre otros, yo no me voy a detener en estos, como les dije antes, ya cuando estén mas sueltos codeando, solos van a ir a probar otros entornos, pero al principio no los quiero complicar más, solo les menciono que desde aquí también pueden lanzar un Jupyter Lab, que es como un Jupyter Notebook mas "tuneado" con mas opciones de configuración,

navegación, ayudas y demás, se los menciono solamente para que los más curiosos lo chusmeen pero de ahora en adelante voy a mostrar ejemplos con un Jupyter Notebook que es más simple para concentrarnos más en Python en sí

Básicos de Jupyter Notebook

Lo principal de los cuadernos de Jupyter notebook es que los archivos que se guardan no son archivos de Python en sí, sino una mezcla de código y comentarios, por lo tanto la extensión de los cuadernos de jupyter no es .py sino .ipynb

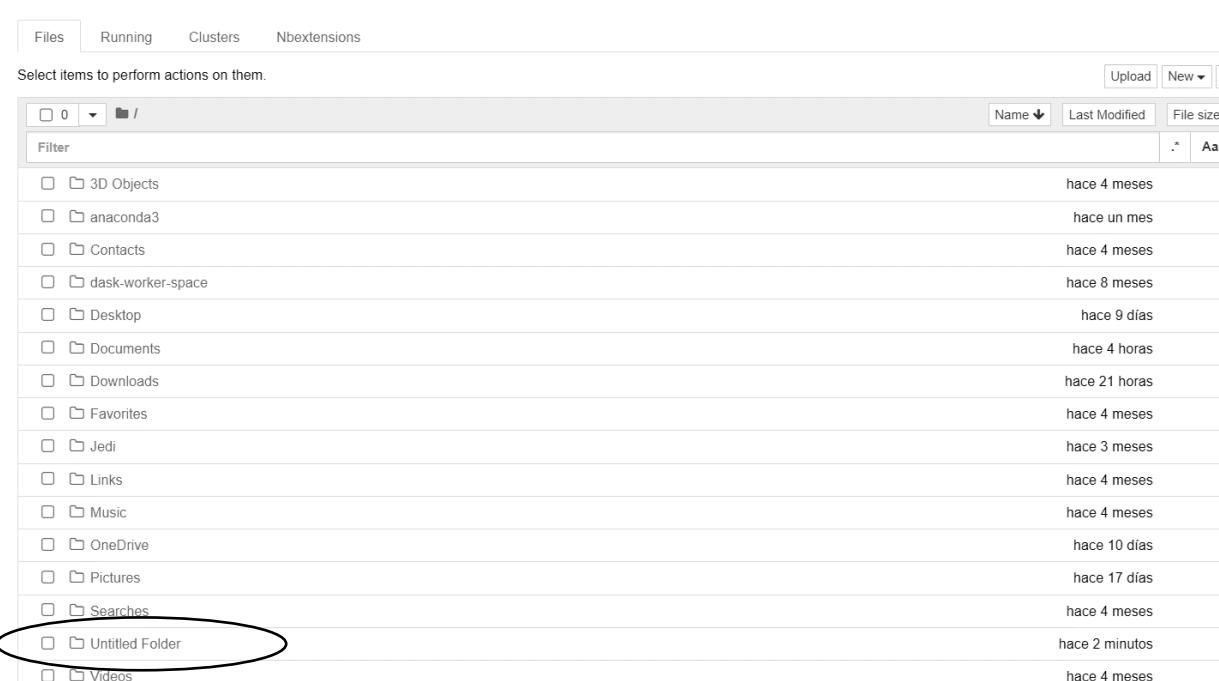
Es normal que al principio se mareen un poco con la interfaz, pero mírenlo como un simple navegador de archivos a la pantalla inicial, de hecho si van a “Desktop” o “Escritorio” o como se llame en su sistema operativo, van a poder acceder a los archivos de su escritorio, o a la carpeta “Mis Documentos” etc..

Dentro de ese navegador principal tienen la opción de crear carpetas:



The screenshot shows the Jupyter Notebook interface with a sidebar on the left containing a file tree and a main workspace on the right. A context menu is open over the workspace, with a circular highlight around the "New" button. The menu options include "Notebook", "Python 3", "Other", "Text File", "Folder", and "Terminal". Below the menu, there is a list of files and their last modified times.

Name	Last Modified
Notebook: Python 3	hace 8 meses
Other:	hace 9 días
Text File	hace 4 horas
Folder	hace 21 horas
Terminal	



This screenshot shows the same Jupyter Notebook interface after a new folder has been created. The newly created "Untitled Folder" is highlighted with a circular oval. The list of files now includes this new folder along with the previously listed items.

Name	Last Modified
Untitled Folder	hace 2 minutos
Notebook: Python 3	hace 8 meses
Other:	hace 9 días
Text File	hace 4 horas
Folder	hace 21 horas
Terminal	

En mi caso creé la carpeta como “Untitled Folder”

Le puedo cambiar el nombre desde acá también, para eso la selecciono, y le doy a "rename"

A screenshot of a file manager interface. At the top, there are tabs for 'Files', 'Running', 'Clusters', and 'Nbextensions'. Below the tabs are buttons for 'Rename', 'Move', and a trash icon. On the right side of the header are buttons for 'Upload', 'New', and a refresh icon. A search bar with dropdown menus for 'Name', 'Last Modified', and 'File size' is followed by filters for '.*' and 'Aa'. The main area shows a list of folders. The first folder, 'Untitled Folder', has a checked checkbox and is circled with a red oval. Other folders listed include '3D Objects', 'anaconda3', 'Contacts', 'dask-worker-space', 'Desktop', 'Documents', 'Downloads', 'Favorites', 'Jedi', 'Links', 'Music', 'OneDrive', 'Pictures', 'Searches', and 'Videos'. To the right of each folder name is a timestamp indicating when it was last modified.

En mi caso le pongo "primeros_scripts"



Y si hago clic en esa carpeta veo que en la barra me muestra esa ruta y la carpeta vacía obviamente

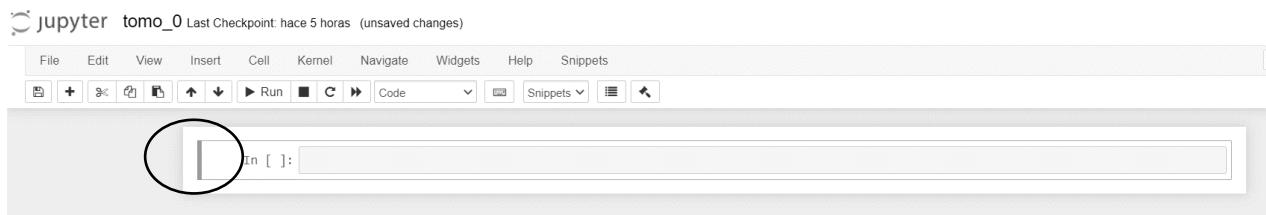
A screenshot of a file manager interface. At the top, there are tabs for 'Files', 'Running', 'Clusters', and 'Nbextensions'. Below the tabs are buttons for 'Upload', 'New', and a refresh icon. A search bar with dropdown menus for 'Name', 'Last Modified', and 'File size' is followed by filters for '.*' and 'Aa'. The main area shows a list of items. The first item, 'primeros_scripts', is highlighted with a red oval and has a checked checkbox. Below it is a folder named '..'. At the bottom, a message says 'The notebook list is empty.' and 'hace unos segundos'.

Y ya dentro de la carpeta voy guardando allí mis notebooks, como archivos .ipynb

Atajos de Teclado de Jupyter

Una de las primeras cosas interesantes que les voy a mencionar son los atajos de teclado que la verdad son super útiles ya que nos hacen ahorrar mucho tiempo, parece una pavada pero les aseguro que a la larga es ridículo no usarlos

Primero hablemos que los notebooks tienen dos modos: Comandos y Edición



Esa zona marcada se pone “azul” cuando estas en modo comandos, y de lo contrario se pone verde cuando estas en modo edición (el modo que es cuando esta el cursor titilando dentro de una celda)

Presionando la tecla “H” en el “modo comandos” (cuando las barras laterales están azules) aparece un cuadro completo con todos los shortcuts, algo así:

The Jupyter Notebook has two different keyboard input modes. **Edit mode** allows you to type code or text into a cell and is indicated by a green cell border. **Command mode** binds the keyboard to notebook level commands and is indicated by a grey cell border with a blue left margin.

Command Mode (press `Esc` to enable)

[Edit Shortcuts](#)

`F`: find and replace

`Ctrl-Shift-F`: open the command palette

`Ctrl-Shift-P`: open the command palette

`Enter`: enter edit mode

`P`: open the command palette

`Shift-Enter`: run cell, select below

`Ctrl-Enter`: run selected cells

`Alt-Enter`: run cell and insert below

`Y`: change cell to code

`M`: change cell to markdown

`R`: change cell to raw

`1`: change cell to heading 1

`2`: change cell to heading 2

`3`: change cell to heading 3

`4`: change cell to heading 4

`5`: change cell to heading 5

`6`: change cell to heading 6

`Ctrl-A`: select all cells

`A`: insert cell above

`B`: insert cell below

`X`: cut selected cells

`C`: copy selected cells

`Shift-V`: paste cells above

`V`: paste cells below

`Z`: undo cell deletion

`D`, `D`: delete selected cells

`Shift-M`: merge selected cells, or current cell with cell below if only one cell is selected

`Ctrl-S`: Save and Checkpoint

`S`: Save and Checkpoint

`L`: toggle line numbers

`O`: toggle output of selected cells

`Shift-O`: toggle output scrolling of

[Close](#)

Pero resumiendo los shortcuts mas usados son los siguientes en modo comando:

- Shift+Enter: Ejecuta la celda activa (corre el código de esa celda)
- Tecla A: Agrega celdas antes (above) de la celda activa
- Tecla B: Agrega celdas después (below) de la celda activa
- Tecla Z: El equivalente al ctrl+Z (paso atrás)
- Doble tecla D: O sea dos veces la tecla D, borra la celda activa
- Tecla Y: Marca la celda activa como celda de “Código”
- Tecla M: Marca la celda activa como celda “Markdown” (de texto)
- Tecla L: Muestra los números de línea de la celda
- Ctrl+S: Guardar los cambios

Si estás en el modo edición, presionando la tecla ESC, pasas a modo comando

Inputs, Outputs y Kernel

Pequeño detalle, los notebooks, tienen celdas donde se puede incluir código (o solo texto) a las que llamamos “inputs” y una salida independiente por cada celda a la que llamamos “output”, es decir que podemos ejecutar cada celda por separado, pero sin embargo el notebook tiene una especie de memoria que engloba a todos los “inputs” y las variables y funciones allí declaradas, a la que llamamos “kernel”

El kernel es como si fuera una memoria donde se guardan todos los inputs y outputs de las celdas que se van ejecutando, y se puede “reiniciar” sin perder lo que está en las celdas



Con ese botón “reiniciamos el kernel”

¿Y qué significa esto?

Vemos, al iniciar un notebook, podemos empezar a asignar valores a variables

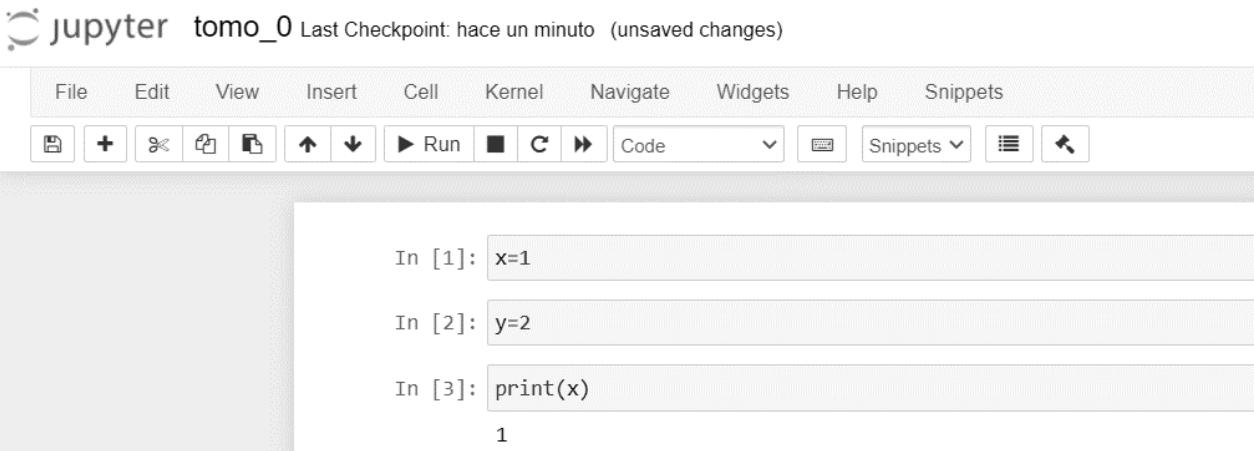
En el siguiente ejemplo, vamos a poner en la celda 1 el siguiente código:

x=1

Esto le asignará el valor de 1 a la variable x, luego en la celda siguiente, asignamos a la variable “y” el valor de 2, y en la tercera celda imprimimos lo que tiene guardado en “x”

Obiamente me muestra 1

Aquí el código:



```
In [1]: x=1
In [2]: y=2
In [3]: print(x)
1
```

Ahora ¿qué pasa si reiniciamos el kernel?

En principio nada, como verán, no se borra nada, pero fíjense que si volvemos luego de reiniciar el kernel a ejecutar la celda 3, nos devuelve un error

```
In [1]: x=1
In [2]: y=2
In [1]: print(x)

-----
NameError                                 Traceback (most recent call last)
<ipython-input-1-fc17d851ef81> in <module>
      1 print(x)

NameError: name 'x' is not defined
```

Y claro, al reiniciar el kernel, los inputs y outputs de las celdas 1 y 2 se perdieron (no las celdas, sino los resultados de haberlas ejecutado), por lo tanto ahora el intérprete de Python no tiene ni idea cuanto vale “x” de hecho no sabe que es “x” porque nunca le definimos eso, si bien quedó en las celdas habrá que volverlas a ejecutar para que quede en la memoria del kernel nuevamente.

Por lo tanto el kernel es como una especie de memoria que al reiniciarse borra todo lo que haya quedado en memoria por haber ejecutado las celdas

Otra cosa a observar es que los inputs (verán a la izquierda de cada celda un In [1], In[2] etc, están numerados, a medida que se van ejecutando, pero al reiniciar un kernel arranca de nuevo de 1 cada input. Otra cosa a saber es que cuando cierro el programa (Jupyter notebook) se “apagan” los kernels, y cuando abra nuevamente los notebooks van a estar “reiniciados”

Outputs versus impresiones de variables

Algo muy característico (y muy cómodo) de los Jupyter Notebooks, es que no necesitamos pedir un print de las variables para saber lo que contienen, ya que simplemente invocándolas, la celda nos devuelve lo que tenía la variable en cuestión

Ejemplo:

```
In [1]: x=1
In [2]: y=2
In [3]: print(x)
1
In [4]: x
Out[4]: 1
```

Como ven, en la celda 4, no necesité escribir print(x), sino que simplemente invocando a la variable, el Jupyter me devuelve solo en un Output lo que contenía esa variable

Es importante aclarar que si usan Spyder, Visual Studio, o cualquier otra interfaz para programar, no tiene una salida automática por el solo hecho de invocar las variable, por lo que en ese tipo de entornos, tendrán que si o si, mandar el print para saber lo que tiene una variable (a menos que usen exploradores de variables u otras herramientas que veremos más adelante seguramente)

Variables

Instrucciones: Una instrucción es pedirle al intérprete de python o el lenguaje que usemos que ejecute una función es decir que me dé una "salida" esa salida puede ser una "acción" o simplemente un valor
La primera instrucción que vamos a usar es "print()" que lo que hace es imprimir en pantalla el argumento que le pasemos entre paréntesis

Variables: Las variables en cambio son como “contenedores” es decir cajas donde guardamos algo, ese “algo” puede ser un valor, una palabra, una oración, un conjunto de valores, una fecha o miles de cosas mas que ya veremos, en principio es un “contenedor de algo”

Ejemplo:

```
In [1]: mensaje = "Hola bro"
print(mensaje)
Hola bro
```

En ese caso estoy guardando en la variable “mensaje” justamente el mensaje “Hola bro” La gracia de guardar algo en una variable, es que luego en otra celda podemos volver a usar lo que estaba en ese contenedor, ejemplo:

```
In [1]: monto = 50
print('El monto es', monto)

El monto es 50

In [2]: print('El monto sigue siendo', monto)

El monto sigue siendo 50

In [3]: monto = monto + 20
print('Ahora sumamos 20, por lo tanto ahora el monto es', monto)

Ahora sumamos 20, por lo tanto ahora el monto es 70
```

Noten que la variable monto, entre la celda 1 y 2, no la cambié ni necesité volver a definirla, ya que sigue valiendo lo que queda en el kernel luego de correr la celda 1 que es 50.

Sin embargo en la celda 3, reescribo la variable “monto” sumándole 20 a lo que tenía esa misma variable.

Otra cosa que vemos en el código es que la variable print, acepta mas de un argumento, es decir mas de una cosa a imprimir, en este caso un texto y una variable, pero veamos por ejemplo el mismo código pero escrito de otra forma:

```
In [3]: monto += 20
print(f'Ahora sumamos 20, por lo tanto ahora el monto es {monto}')

Ahora sumamos 20, por lo tanto ahora el monto es 70
```

Ahí cambiamos dos cositas, en primer lugar en lugar de escribir
monto = monto + 20

Pusimos:

monto += 20

Esto es una simple abreviación de sintaxis que se usa en casi todos los lenguajes, el operador antes del igual indica que vamos a operando sobre la misma variable, si pusiera

x -= 5 (indicaría que a lo que tenga en “x” le tengo restar 5

Lo mismo si uso otros operadores como * para multiplicar, o / para dividir

El otro cambio es que usamos un format string, es decir le pusimos una “f” antes de la comilla del texto a imprimir, y luego dentro de ese texto incluimos la variable {monto} entre llaves, las llaves en un format-string, es decir en un texto antecedido por la letra “f” indican que lo que va dentro de las llaves es una variable y lo que tiene que imprimir no es el nombre de la variable, sino lo que esta contenga

Asignación de valores a variables

Una variable es como una caja donde uno guarda algo, una vez que pongo algo en la caja, la próxima vez que quiera usar eso que puse, solo tengo que ir a la caja y lo que puse allí dentro, allí estará

```
come = 3  
print(come)
```

3

Lo diferente q tienen las variables a las cajas físicas es que puedo poner otra cosa diferente sin tener que sacar lo que tenía previamente, esto es sobrescribir la variable, obviamente que el valor antiguo se pierde

```
ggal = 100  
ggal = 101  
print(ggal)
```

101

Naturalmente queda el último valor asignado, el valor de 100 se pierde porque queda sobrescrito.

Asignación de variables por consola o interfaz de usuario

En algún momento de un programa puede ser necesario tener que pedirle al usuario un dato para guardarlo en una variable, para ello hay muchos método el mas sencillo en python es usar la instrucción input()

```
stopLoss = input("Ingresar el stopLoss ")  
print("Tu stopLoss quedó configurado en",stopLoss)
```

Ingresar el stopLoss 100

Tu stopLoss quedó configurado en 100

Como vemos en este ejemplo ahora a la instrucción print() le pasamos 2 argumentos en vez de 1, un texto "Tu stopLoss quedó configurado en" y el valor que guardamos en la variable

Siempre que las funciones o instrucciones acepten más de un argumento, estos irán separados por comas

Otra manera de hacer exactamente lo mismo es concatenando la salida de print, veámoslo en el ejemplo:

```
stopLoss = input("Ingresar el stopLoss ")
print("Tu stopLoss quedó configurado en " + stopLoss)
```

```
Ingresar el stopLoss 100
Tu stopLoss quedó configurado en 100
```

Y otra manera en python de concatenar un texto al valor de una variable es como vimos antes, usando "fstrings" son cadenas formateadas y para indicarle al intérprete que queremos referirnos a este tipo de cadenas le poneos la letra **f** antes de iniciar la cadena.

```
stopLoss = input("Ingresar el stopLoss ")
print(F"Tu stopLoss quedó configurado en {stopLoss}")
```

```
Ingresar el stopLoss 100
Tu stopLoss quedó configurado en 100
```

Python es Case Sensitive

¿Qué significa eso?

Que es sensible a minúsculas o mayúsculas, es decir que diferencia minúsculas de mayúsculas, o dicho de otra forma que la variable monto es diferente a la variable Monto, y a la variable MONTO

```
In [1]: monto = 1
In [2]: Monto = 5
In [3]: print(monto)
        1
In [4]: print(Monto)
        5
In [5]: print(MONTO)
-----
NameError                                 Traceback (most recent call last)
<ipython-input-5-7ec2a2f7e449> in <module>
----> 1 print(MONTO)

NameError: name 'MONTO' is not defined
```

Obviamente en el ejemplo anterior, guarda en cajas distintas los valores de monto y Monto ya que las considera cosas distintas, y si le preguntamos por lo que tiene en MONTO, no la reconoce

Tipos de datos

Hay muchos tipos de datos en Python, vamos a ir viéndolos de a poco, empezemos por los mas sencillos:

Para saber de que tipo es un dato, podemos usar el comando type()

Números Enteros int()

```
a = 5  
type(a)
```

int

Números decimales float()

```
a = 3.14  
type(a)
```

float

Booleanos (True or False)

```
a = False  
type(a)
```

bool

Ojo con la primera letra de True o False, ya que si no es mayúscula Python no lo va a entender, recordemos que Python es Case-Sensitive, por lo tanto no es lo mismo True que true veamos

```
a = True  
type(a)
```

bool

Strings

Los strings se denotan entre comillas, es lo mismo comillas simples que dobles

```
a = "True"  
type(a)
```

str

Notar que si ponemos "True" entre comillas python no va a entenderlo como un valor booleano sino como un texto cualquiera

Nulos

```
a = None  
type(a)
```

NoneType

Y eso que fue?

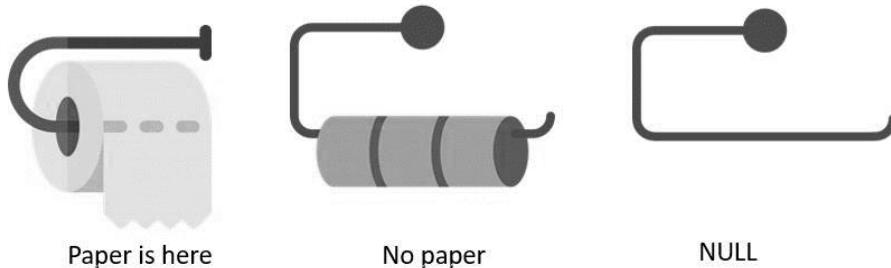
Eso es un dato Nulo, en otros lenguajes se los llama **Null** que sería como una especie de "no seteado" que no es lo mismo que vacío ya que vacío es más un sinónimo de "cero", en cambio el "Nulo" es algo que ni siquiera sabemos si estamos hablando de una variable que representa una fecha o un número o un string por decir algo.. Mas adelante se va a entender mejor el concepto

Por ejemplo, si en la variable mensaje voy a guardar un texto pero aun no se que texto va a llevar se suele definir como

```
In [1]: mensaje = ""
```

Pero en ese caso si está seteado el tipo como un string, un string vacío pero una cadena de texto al fin porque se puso entre comillas, en cambio cuando se declara como None, ni siquiera se sabe de que tipo será

Una especie de meme para entender las variables nulas en programación es el siguiente:



En el ejemplo del papel higiénico llevado a código sería lo siguiente:

```
In [1]: rollo_1 = "Rollo de papel higiénico"  
rollo_2 = ""  
rollo_3 = None
```

Otros tipos de Variables

Obvio que hay muchos tipos mas de variables pero los vamos a ir viendo de a poco a medida que vayamos avanzando, les voy adelantando las mas usadas que veremos en breve:

- Fechas
- Listas
- Tuplas
- Diccionarios
- Objetos

Ojota con esto, operar variables de diferente tipo

Multiplicamos dos números, tudo bom tudo legal:

```
a = 2  
b = a * 3  
print(b)
```

6

Y si multiplicamos algo parecido?

```
a = "2"  
b = a * 3  
print(b)
```

222

Veamos que pasó acá, en realidad, el “2” entre comillas no es el número 2, sino que es la cadena de texto “2”, por lo tanto lo trata como string, y multiplicar por 3 un string, es repetir esa cadena de texto o concatenarla a si misma x veces, en este caso 3 veces , por eso “2” * 3 = “222”

Errores por operar tipos de datos diferentes

```
a = "2"  
b = a + 3  
  
print(b)
```

```
-----  
TypeError Traceback (most recent call last) <ipython-input-  
16-78260d30c223> in <module> 1 a = "2"  
      2 b = a + 3  
----> 3 TypeError: can only concatenate str (not "int") to str
```

Como vemos, le asigné a una variable un número y a otra un texto, ahora que pasa si las quiero concatenar en una sola oración

Lo que me dice el intérprete de python es que **no puede concatenar strings con enteros** Si ambos hubieran sido números el intérprete los hubiera sumado

Si ambos hubiesen sido texto, los hubiera "concatenado" un texto atrás del otro
Pero como le mandamos un número y un texto, no sabe qué hacer y tira error

Si hay algo copado que tiene python es que su intérprete es muy claro al mostrar donde se producen los fallos, aprovechen eso, lean bien el error que tira para entender dónde está fallando nuestra lógica

Operaciones permitidas con diferentes tipos

Esto es algo loco que permite Python que muy pocos lenguajes de programación permiten que es operar con diferentes tipos en algunas ocasiones, pero ojo que a veces no es tan esperable ese comportamiento, ejemplo:

```
In [1]: a = True  
        b = 5  
        print(a + b)
```

6

Como ven, Pytohn evaluá el True (booleano) como 1, y al False como 0, es decir que me dejará operar con ellos como si fueran los números 1 y 0 respectivamente

Python es un lenguaje interpretado, dinámicamente tipado

¿Lo Qué?

Sin entrar en tecnicismos, hay lenguajes de programación compilados y otros interpretados, esto quiere decir que el código que nosotros escribimos para ser ejecutado por la computadora debe primero ser interpretado o compilado

- **Los compiladores** primero "leen" todo el código, luego lo compilan y recién ahí puede ejecutarse
- **Los intérpretes** Interpretan el código para que pueda ser ejecutado línea por línea

Los lenguajes interpretados como Python, tienen un intérprete que va "traduciendo" a lenguaje de maquina nuestro código línea por línea mientras se va ejecutando.

Y lo de "dinámicamente tipado"?

Esto quiere decir que en una línea del código yo puedo definir una variable como un "string" o cadena de texto, y un par de líneas más adelante usar esa variable y asignarle un valor numérico "integer" por ejemplo

```
a = "GGAL"  
print(type(a))  
a = 100  
print(type(a))  
<class 'str'>  
<class 'int'>
```

Si python hubiese sido un lenguaje de tipado estático, yo no hubiera podido asignar a la misma variable primero un tipo de datos y después otro

Por el contrario, un lenguaje compilado, en lugar de tener un "intérprete" que pase a lenguaje de maquina nuestro código línea por línea mientras se va ejecutando, tiene un "compilador" que antes de ejecutar nada hace toda la traducción y luego, una vez traducido (compilado) todo, recién ahí se puede ejecutar

El tipado estático contrariamente al dinámico tiene una ventaja principal que es que al compilarse el código me estaría diciendo el compilador que hay una "incoherencia" en una misma variable a la que le quiero asignar datos que nada que ver y me permitiría corregir esto antes de correr el programa.

En cambio, en un lenguaje interpretado como Python, el intérprete arranca "a ciegas" ejecutando sin tener idea de qué tipo de valor le asignaremos luego una variable y esto es cómodo al principio, pero si nos mandamos cualquiera el código se va a romper en medio de la ejecución

Operaciones Básicas

Bueno, obviamente las operaciones aritméticas básicas serían:

```
a = 5  
b = 3  
suma = a+b  
producto = a*b  
division = a/b  
división_entera = a // b  
mod = a % b  
potencia = a**b
```

Y si veo lo que guardó la celda anterior en cada variable:

```
producto
```

15

El mod es el resto de la división:

```
mod
```

2

La división entera es justamente eso, es decir solo la parte entera de la operación:

```
division_entera
```

125

```
potencia
```

125

Métodos básicos para trabajar con strings

Los métodos más usuales son

- `string.capitalize()` => Pasa a mayúscula la primera letra de la variable string
- `string.upper()` => Pasa todas las letras a mayúsculas de la variable string
- `string.lower()` => Pasa todas las letras a minúsculas de la variable string
- `string.title()` => Pasa las primeras letras de cada palabra a mayuscula de la variable string
- `string.rfind("string_buscado")` => Devuelve la última posición donde aparece por primera vez el string_buscado en la variable string
- `string.find("string_buscado")` => Devuelve la primera posición donde aparece por primera vez el string_buscado en la variable string
- `string.replace("str_buscado","str_reemplazo")` => Reemplaza "str_buscado" por "str_reemplazo" en la variable string
- `string.zfill(largo)` => Rellena con ceros a la izquierda hasta completar el largo pasado como argumento
- `string.count("string_buscado")` => Devuelve la cantidad de veces que aparece el string_buscado en la variable string
- `len(string)` => Cuenta la cantidad de caracteres de la variable string
- `string.isalnum()` => True si todos los caracteres son alfanumericos
- `string.isalpha()` => True si todos los caracteres son letras.
- `string.isdigit()` => True si todos los caracteres son dígitos

Ejemplos

```
string = "AAPL"  
string.upper()
```

```
'AAPL'
```

```
string = "El dolar va a valer $211,11"
string.title()
'El Dolar Va A Valer $211,11'
```

```
string = "GFGC100.AB"
string.find(".")
7
```

```
string = "AAPL,GGOGL,AMZN,FB,TSLA"
string.replace("TSLA", "BTC")
'AAPL,GGOGL,AMZN,FB,BTC'
```

```
string="1"
string.zfill(3)
'001'
```

```
string = "AAPL,GGOGL,AMZN,FB,TSLA"
string.count("A")
```

4

```
string = "BTC"
len(string)
```

3

```
string = "123"
string.isdigit()
```

True

```
string = "123"
string.isalnum()
```

True

```
string = "123"
string.isalpha()
```

False

Ejercicios con Strings 1

- 1- Dada la siguiente cadena de texto:

"La ponderación de AAPL en nuestro fondo pasó desde el 4.35% del portafolio al 4.23%"

- a) Encontrar la posición donde se encuentra el ticker AAPL
- b) Contar la cantidad de veces que aparece en el texto el símbolo "%"

- 2- Dado el siguiente ticker de opción: "GFGC155.AG"

- 1- Extraer los primeros 3 caracteres correspondientes al subyacente
- 2- Extraer todos los caracteres posteriores al punto

Respuestas, ejercicios con strings 1

```
#=====
#      Ejercicio 1a      #
#=====

texto = "La ponderación de AAPL en nuestro fondo pasó desde el 4.35% del portafolio al 4.23%"
texto.find("AAPL")
```

18

```
#=====
#      Ejercicio 1b      #
#=====

texto = "La ponderación de AAPL en nuestro fondo pasó desde el 4.35% del portafolio al 4.23%"
texto.count("%")
```

2

```
#=====
#      Ejercicio 2a      #
#=====

ticker = "GFGC155.AG"

subyacente = ticker[:3]
subyacente

'GFG'
```

```
#=====
#      Ejercicio 2b      #
#=====

ticker = "GFGC155.AG"

posicion_punto = ticker.find(".")
ultimos = ticker[posicion_punto +1 :]
ultimos

'AG'
```

Cambiando los tipos de datos

Generalmente este tipo de cosas de cambiar el tipo de datos que tiene una variable no es una gran idea en programación ya que seguramente si nos viene una variable en un tipo de dato (entero, flotante, texto etc) es porque en algún momento el programa estará esperando ese tipo y si en el medio se lo cambiamos probablemente más adelante aparezcan problemas, esto claro está cuando trabajamos en proyectos grandes, si son pequeños programas caseros donde todo el código lo escribimos nosotros mismos, no hay problema, y suele ser un recurso muy usado...

Dicho esto, ya puedo dormir tranquilo que les dije que no era buena idea así que veamos esta joyita que permite python

```
precio = 100.23433
print(precio,type(precio))
```

```
100.23433 <class 'float'>
```

```
precio = 100.23433
precio = int(precio)
print(precio,type(precio))
```

```
100 <class 'int'>
```

```
precio = 100.23433
precio = str(precio)
print(precio,type(precio))
```

```
100.23433 <class 'str'>
```

O sea que puedo convertir cualquier número flotante en entero o en string
Pero ojo, siempre recuerden a Peter Parker: "Todo gran poder conlleva una gran responsabilidad"

Esto de andar jugando demasiado con los tipos de datos puede generar cosas inesperadas y de consecuencias impredecibles

```
precio = str(100.23433)
precio * 2
'100.23433100.23433'
```

Palabras Reservadas

Ojo al piojo con las palabras reservadas, son pocas pero son palabras que python se reserva para "uso especial"

Esto quiere decir que no podemos usar esas palabras para llamar a variables o funciones etc
Se puede llamar a estas palabras con el comando help("keywords")

```
help("keywords")
```

```
Here is a list of the Python keywords. Enter any keyword to get more help.

False      class      from      or
None       continue   global    pass
True       def        if        raise
and        del        import   return
as         elif       in       try
assert    else       is        while
async     except    lambda   with
await     finally   nonlocal yield
break
```

Veamos qué pasa si le quiero poner de nombre a una variable una palabra reservada

```
global = 12
print(global)
```

SyntaxError: invalid syntax

Y es lógico que me devuelva error de sintaxis, o error de asignación, en todo caso eviten usar nombres de variables que se usen para otras cosas porque a veces incluso sin ser palabras reservadas, son nombres de funciones importantes que no me devolverá error pero me generará problemas posteriores

En el ejemplo muestro como piso el método print() con un valor string, y luego no me deja usar la función print, claro porque está pisada, ahora print() ya no es mas un método sino un string con el valor "Juan"

```
print = "Juan"

print("Hola gente")

-----
TypeError                                Traceback (most recent call last)
<ipython-input-19-1c3c0d0d0561> in <module>
----> 1 print("Hola gente")

TypeError: 'str' object is not callable
```

Números al Azar

Muchas veces en nuestros programas vamos a necesitar tomar determinados valores arbitrarios, para ello contamos con las siguientes funciones

Valor flotante al azar entre 0 y 1

Es importante entender que este comando devuelve un valor pseudoaleatorio según una distribución uniforme, es decir que cualquier valor dentro del intervalo 0-1 tiene la misma probabilidad de ser elegido

```
import random  
random.random()
```

0.5291112005019688

Valor flotante entre min y max

De la misma manera puedo buscar un valor siguiendo una distribución uniforme pero acotado entre dos valores reales cualesquiera, ejemplo:

```
random.uniform(2,3)
```

2.432532146642264

Valor entre min y max definiendo intervalo o "step"

También puedo elegir un valor “entero”

En este ejemplo busco valores pares entre 0 y 100:

```
min = 0  
max = 100  
paso = 2  
random.randrange(min,max,paso)
```

Número aleatorio en una Distribución

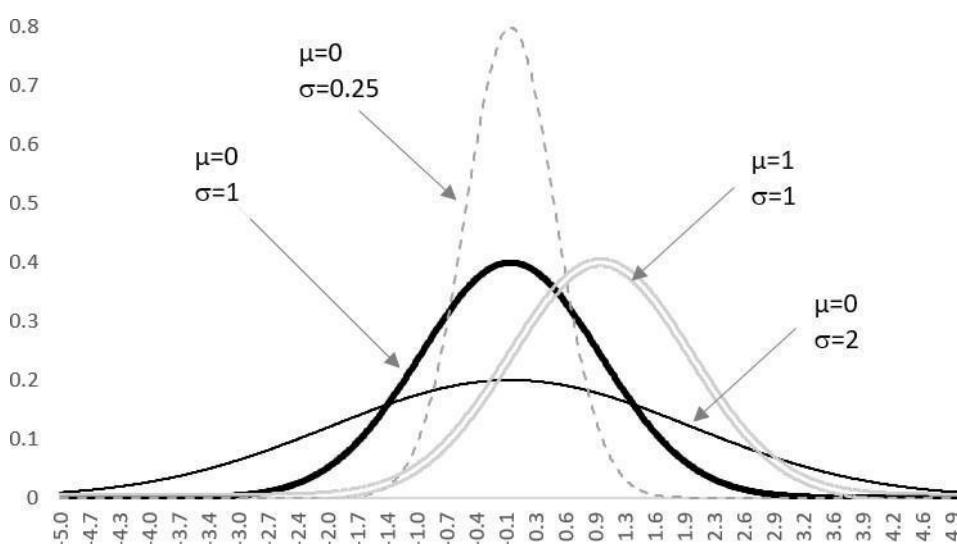
A parte de darnos números aleatorios al azar, la librería random nos provee de funciones para obtener un valor aleatorio de una función de distribución de probabilidades como por ejemplo de la famosa distribución normal, esto nos va a ser sumamente útil para realizar más adelante simulaciones de montecarlo

Repasemos primer que es una distribución normal

Lo que muestra esta función en el eje "Y" es la probabilidad de ocurrencia de un valor del eje "X", a mas alto el punto en la gráfica más probable es.

Podremos ver los valores del eje "X" como +/- la cantidad de desvíos estándar (σ) de la media (μ)

Por ejemplo, si asumimos que los retornos diarios de una acción siguen una distribución normal, μ sería el retorno medio y σ la volatilidad de los retornos



Utilizando la función `random.normalvariate(mu,sigma)` vamos a obtener valores aleatorios del eje X de la normal (μ , σ)

```
mu = 0
sigma = 1
random.normalvariate(mu,sigma)
```

-0.06611273222951637

La normal 0,1 o de media = 0 y desvío estándar = 1, es la conocida como “normal estándar” y es muy usada en el campo de la estadística, sin embargo como pueden imaginar podría usar números al azar con cualquier distribución normal pasándole argumentos personalizados con el mu y sigma que quisiera

Si bien son muy útiles las distribuciones normales porque muchísimas cosas tienen una distribución normal en la vida real, lo cierto es que muchas veces nos encontramos con distribuciones con otras formas y necesitaremos generar valores aleatorios de distribuciones no-normales, la librería estándar de random a la hora de escribir estas líneas no es muy completa y solo nos provee una limitada cantidad de distribuciones entre ellas:

- `random.betavariate(alfa,beta)`
- `random.gamavariate(alfa,beta)`
- `random.expovariate(lambda)`
- `random.lognormalvariate(mu,sigma)`
- `random.paretovariate(alfa)`
- `random.weibullvariate(alfa,beta)`

Seguramente en algún momento incluirán alguna distribución más, pero les voy adelantando que la librería de Python por excelencia para trabajar con mas distribuciones y funciones de estadística es `scipy` emas específicamente el módulo `stats` pero lo veremos en el t6 mas adelante.

También es importante aclarar que la librería `numpy` que veremos mas adelante como la librería `pandas`, ambas levantan el módulo de la librería `random` y lo complementan con funcionalidades vectoriales mas que interesantes, con lo cual posiblemente en aplicaciones mas avanzadas usemos esas librerías

Semilla

Muchas veces al usar números aleatorios van a escuchar hablar de “semillas” de hecho si se han metido a investigar un poco el mundo de las cryptomonedas les sonará eso de la “frase semilla” o “palabras semilla” que son una serie de 12 o mas palabras que funcionan como una especie de “clave” para recuperar las wallets

En realidad no son una clave, son justamente una semilla, la función de una semilla es algo así como “fijar un punto de partida”, ¿para qué serviría esto?

Bueno, pensemos, al crear una wallet, en realidad tenemos que crear un conjunto de claves pública y privada “aleatorias” pero ese “aleatorias” es mas complejo de lo que parece

Es decir, si yo te digo, “elegí un ‘numero al azar entre 0 y 100”

Vos sos perfectamente capaz de hacerlo, totalmente al azar, pero ¿podría hacer eso una computadora?

La pregunta es ¿tiene libre albedrío una computadora?

Claramente las computadoras no tienen libre albedrío, por lo tanto son totalmente incapaces de generar números o caracteres al azar, sencillamente no pueden hacerlo, lo único que pueden hacer las computadoras es seguir una serie de instrucciones bien precisas y concretas para obtener un resultado, es decir seguir un algoritmo, pero no pueden elegir números al azar “caprichosamente”, tienen que seguir instrucciones bien claras para lograrlo

Entonces dirán “si sigue instrucciones para obtener un número al azar entonces no es al azar”

Bingo!

Por eso los números que generan los algoritmos los llamamos “pseudoaleatorios”

Volviendo al concepto de las frases semilla de las wallets, el algoritmo que crea una wallet, arma una dirección con caracteres que parecen totalmente aleatorios como por ejemplo:

- 0x723c595f843ad4a78c7fe73413faf7c35bb01d51

Y para esa llave pública generará con otro algoritmo una llave privada para firmar transacciones, el punto es que el algoritmo que usa para generar ese par de llaves no sabe generar números o caracteres al azar, sino que simplemente emplea un algoritmo bastante caótico que al menor cambio en la semilla genera pares totalmente diferentes, tal que parecen “al azar” o parecen “aleatorios” pero no lo son, son pseudoaleatorios, es decir, siguiendo un algoritmo generan un resultado partiendo de un input llegan a un output, es decir una función perfectamente definida, el tema es que ese input es el que define todos los outputs que genera el algoritmo, por eso son tan importantes las palabras semilla de una wallet, porque aplicando el algoritmo a esa semilla podemos generar todas las direcciones y pares público privado.

Bien, volviendo a Python y lo que nos incumbe ahora, si pretendemos que Python o una computadora genere números “al azar” en realidad vamos a tener que darle una semilla y que aplique un algoritmo tan caótico que parezca que esa salida es un número totalmente al azar

Veamos el ejemplo:

```
import random  
  
random.seed(0)  
random.random()  
  
0.8444218515250481  
  
random.seed(0)  
random.random()  
  
0.8444218515250481
```

Aquí usamos como semilla el “0” y ejecutamos con esa misma semilla dos veces el random.random() y como vemos nos genera dos veces exactamente el mismo número: 0.8444218515250481

¿Y a mi que me importa esto de las semillas si yo solo quiero los números aleatorios?

Buena, pregunta, en la primera edición de este tomo no había ni nombrado el tema pero empecé a recibir muchas consultas de alumnos con la misma duda:

“¿y si quiero que genere aleatorios pero que cuando la ejecute en una compu o en otra me de lo mismo?”

Entonces, les decía “fijá la misma semilla” y claro, me daba cuenta que todos me respondían lo mismo:

“lo queeee?”

Así que bueno, ya saben para que sirve entender esto, para que ejecute donde lo ejecute, o cuando lo ejecute siempre me va a generar el mismo “aleatorio”

Es importante entender que con la misma semilla puedo generar varios “aleatorios” como en el siguiente ejemplo donde genero 3 aleatorios diferentes y los guardo en las variables: r1, r2, y r3

```
import random
random.seed(0)
r1, r2, r3 = random.random(), random.random(), random.random()
print(r1,r2,r3)
0.8444218515250481 0.7579544029403025 0.420571580830845
```

Fíjense que el primer “aleatorio” ese 0.84442185.. es el mismo de siempre que use la random.seed(0) pero los otros son diferentes, aunque siempre que use esa semilla se generarán los mismos en el mismo orden, es lo mismo que pasa con las wallets y sus palabras semilla, con esas palabras, siempre generaré los mismos pares de claves públicas-privada en la compu que sea en el momento que sea, por eso es tan importante guardar bien esa frase, porque con ella, todas las wallets (todas) que haya creado el software (sea metamask, trustwallet, mew, o una hard wallet o lo que sea) se podrán regenerar (tanto las direcciones como sus llaves privada para firmar transacciones)

Por si te lo estás preguntando, ¿qué hace el programa si no fijo ninguna semilla?

El programa calcula una semilla en función de la hora del sistema, es decir, insisto, no tiene libre albedrío es una computadora, no tiene ese grado de libertad.

Operaciones Matemáticas. Librería Math

Esta librería no se usa mucho ya que es super chica, pero como recién arrancamos viendo librerías me parece buen momento para mostrarla

```
import math
```

Constantes

Tenemos las dos constantes mas usadas en matemática que son pi y e.

```
math.pi
```

```
3.141592653589793
```

```
math.e
```

```
2.718281828459045
```

Truncado, piso y techo de un flotante

Otra funcionalidad que nos brinda math es el truncado, piso y techo de un flotante

Entero inmediato inferior

```
math.floor(5.44)
```

```
5
```

Entero inmediato superior

```
math.ceil(5.44)
```

```
6
```

Entero

```
math.trunc(-5.34)
```

```
-5
```

Redondear

Esta función admite como parámetro la cantidad de decimales

Esta ya no es una función de la librería math sino que es una función nativa de Python, por eso no la invocamos antecediendo la librería math, sino que directamente llamamos a la función:

```
round(5.76543, 3)
```

```
5.765
```

Logaritmos

Logaritmo base 10 (Si se pasa un segundo argumento a la función es la base, por default es e)

```
math.log(100,10)
```

2.0

Como saben el logaritmo en base 10 de 100 es 2 porque 10 al cuadrado es 100

```
e = math.exp(1)  
math.log(e)
```

1.0

El logaritmo natural de e es 1 porque e elevado a la 1 es e

Máximo común divisor

El famoso MCD que vemos en el colegio primario

```
math.gcd(9,24)
```

3

Factorial

```
math.factorial(6)
```

720

Como saben el factorial de 6 es 720, se escribe como: $6! = 720$

Porque $6 * 5 * 4 * 3 * 2 * 1 = 720$

isClose

Decidir si un numero está cerca de otro o no en función de una tolerancia porcentual

```
a = 989  
b = 1000  
tol= 0.05  
  
math.isclose(a,b,rel_tol=0.01)
```

False

Ejercicios con random

1 - Obtener un precio aleatorio no importa la distribución, para la acción de GGAL que oscile entre 100 y 110 y con precisión de 0.01, asignarlo a la variable ggal

Como habrán notado el principal problema es que la función randrange permite sensibilidades enteras no de 1/100 como pide el ejercicio, así que tenemos que usar el ingenio para adaptar la herramienta a nuestro problema

2- Resolver el ejercicio 1 de nuevo, pero de otra forma totalmente diferente

No, no es joda, ese es el ejercicio.. No me puteen, esperen, posta que es muy importante ejercitarse esto, muchas veces en programación resolvemos algo de una forma, y por "x" motivo, después se complica todo y debemos volver a resolverlo de otra, si bien estos ejemplos son triviales, mas adelante vendrán desafíos más difíciles y está bueno ir entrenando la creatividad un poco a la hora de plantear o "cranear" como vamos a resolver el problema.

3 - Asumiendo que un activo tiene una distribución de sus rendimientos diarios muy similar a la distribución normal, y que su media diaria es de 0.1% con una volatilidad diaria del 2.5%. Escribir la instrucción que me devuelva valores aleatorios de su rendimiento diario

4 - Vamos a simular un recorrido de precios para GGAL durante toda una semana partiendo que el último viernes cerró a \$135.51 y que se mueve diariamente con una media del 0.012% y un desvío estándar del 3.5%. La idea es obtener una serie de precios para el fin del lunes, martes, miércoles, jueves y viernes.

- a) Obtener la serie fijando la semilla 123 (a todos nos tiene que dar lo mismo obvio)
- b) Obtener una serie pseudoaleatoria sin especificar nada

Respuestas

Un primer camino (el que uso yo siempre) es jugar con las unidades, imaginar el precio en lugar de en centavos, y luego si volver a pasarlo una vez resuelto el problema y listo, quedaría así la solución

```
#=====
#          Ejercicio 1          #
#=====

import random

min = 10000
max = 11000
step = 1
ggal = random.randrange(min,max,step)/100
ggal
```

103.4

Otra alternativa sería separando el problema en dos, buscar un aleatorio entero entre 100 y 109 y otro aleatorio decimal entre 0 y 1 y sumarlos luego. Esto conlleva mayor esfuerzo computacional por lo que en este caso no es la mejor alternativa ya que la solución mostrada anteriormente es más simple y requiere menos esfuerzo computacional.

Esto del esfuerzo computacional en un ejemplito simple, así como este no tiene sentido plantearlo, pero imaginen que "ese precio aleatorio" es algo que voy a usar en una simulación que repite el proceso miles de veces para cada fecha y en varias fechas para cada activo y lo hace diariamente con miles de activos

```
#=====
#          Ejercicio 1 - Otra opcion      #
#=====

min = 100
max = 109
step = 1
entero = random.randrange(min,max,1)
decimales = random.uniform(0,1)
ggal = round(entero + decimales,2)
ggal
```

104.97

```
#=====#
#      Ejercicio 3          #
#=====#
```

```
import random
random.seed(0)
random.normalvariate(0,2.5)
```

```
-0.45967055273314567
```

La variante del ejercicio 4b es exactamente igual pero sin fijar la semilla en la línea 6

```
1 #=====
2 #      Ejercicio 4          #
3 #=====#
4
5 import random
6 random.seed(123)
7 precio_inicial = 135.51
8 mu, sigma = 0.00012, 0.035
9
10 variacion = random.normalvariate(mu, sigma)
11 lunes = precio_inicial * (1+variacion)
12
13 variacion = random.normalvariate(mu, sigma)
14 martes = lunes * (1+variacion)
15
16 variacion = random.normalvariate(mu, sigma)
17 miercoles = martes * (1+variacion)
18
19 variacion = random.normalvariate(mu, sigma)
20 jueves = miercoles * (1+variacion)
21
22 variacion = random.normalvariate(mu, sigma)
23 viernes = jueves * (1+variacion)
24
25 print(f"Los precios simulados son: ")
26 print(f"\{lunes:.2f}, \{martes:.2f}, \{miercoles:.2f}, \{jueves:.2f}, \{viernes:.2f}\")
```

```
Los precios simulados son:
134.68, 135.13, 138.55, 136.53, 135.98
```

Trabajando con Fechas

La librería datetime

Primero debo importar la librería datetime para trabajar fechas y horas, es una librería de las más estándar de python

```
import datetime as dt
```

Y que significa el "as dt" bueno esto es un "alias" es decir que cuando nos queramos referir a la librería datetime la llamaremos **dt**

Como vemos hasta ahora el importar una librería no me devuelve nada, solo carga en el kernel las funciones de la librería

Antes de avanzar con los métodos de esta librería vamos a ver un concepto importante de las librerías que son las sus librerías, o paquetes individuales de la librería maestra En el caso de la librería **datetime** tenemos los siguientes paquetes o su librería:

- date
- time
- datetime
- timedelta
- timezone
- tzinfo

Importando sub-paquetes

Importemos una sublibreria de la librería principal **datetime**, por ejemplo, de la siguiente manera:

```
from datetime import datetime as dt_dt
from datetime import date as dt_date
```

Esos alias nos permiten referirnos en modo más abreviado a los sub-paquetes:

- En lugar de **datetime.datetime** solo diremos **dt_dt**
- En lugar de **datetime.date** solo diremos **dt_date**

Obviamente no hace falta ponerle alias a los paquetes ni subpaquetes, yo acá lo hago para mostrar el tema “alias” y para acortar las llamadas a los subpaquetes, ya que estas dos formas son equivalentes

Bueno, vamos a ver cuál es la fecha actual con la hora actual

```
hoy = dt_dt.now()  
print(hoy, type(hoy))
```

```
2020-03-13 16:34:55.435794 <class 'datetime.datetime'>
```

Como vemos, el dato de la fecha almacenado en la variable “hoy” es un dato de tipo objeto, ya veremos más adelante en detalle el significado de un “objeto” en programación, pero por ahora vamos a decir que a diferencia de una variable “común” (un simple número) un objeto además de un valor único tiene atributos y métodos

Atributos del objeto datetime

Bueno, como se imaginarán los atributos de una fecha van a ser:

- El año, El mes, El día, La Hora, Los minutos, Los segundos, Los microsegundos

¿Y cómo accedemos a esos atributos?

```
from datetime import datetime as dt_dt  
hoy = dt_dt.today()  
  
año = hoy.year  
mes = hoy.month  
dia = hoy.day  
hora = hoy.hour  
minutos = hoy.minute  
segundos = hoy.second  
microsegundos = hoy.microsecond  
  
print(f"El año es: {año}")  
print(f"El mes es: {mes}")  
print(f"El dia es: {dia}")  
print(f"La hora es: {hora}")  
print(f"Los minutos son: {minutos}")  
print(f"Los segundos son: {segundos}")  
print(f"Los microsegundos son: {microsegundos}")
```

```
El año es: 2020  
El mes es: 3  
El dia es: 13  
La hora es: 16  
Los minutos son: 35  
Los segundos son: 20  
Los microsegundos son: 94416
```

Si en lugar de now() usamos el método today() de la misma librería obtenemos el mismo resultado

```
hoy = dt_dt.today()  
print(hoy, type(hoy))
```

```
2020-03-13 16:46:59.125873 <class 'datetime.datetime'>
```

¿Y si quiero solo la fecha y no me importa la hora?

Para ello usaremos una sublibreria de datetime que es date y su método today() que se llama igual que en la sublibreria datetime

```
hoy = dt_date.today()  
print(hoy, type(hoy))
```

```
2020-03-13 <class 'datetime.date'>
```

¿Pero qué tipo de dato es en realidad ese? Acá estamos hablando de objetos. Una de las ventajas de tener las fechas como objetos es que podemos aplicarles todos los métodos que ya vienen en la librería escritos y acceder a sus atributos por separado

¿Y que métodos tiene la librería datetime?

Por ejemplo uno de ellos puede ser un método como isoformat() que me permite convertir un objeto fecha en un string del formato YYYY-MM-DD que es el estándar ISO

Como venimos viendo con el método dt_dt.today() que me devuelve la fecha de hoy, se llaman poniendo un punto al paquete que lo contiene que puede ser una librería o un objeto, y luego el nombre del método y una apertura y cierre de paréntesis dentro de los cuales irían los atributos (si los necesitara)

Fechas en formato string

El método ctime() es un método muy sencillo que me devuelve una fecha legible muy cómoda en formato string partiendo de un objeto fecha datetime

```
hoy = dt.datetime.today()  
hoy_str = hoy.ctime()  
print(hoy_str, type(hoy_str))
```

```
Fri Mar 13 16:40:57 2020 <class 'str'>
```

Convertir un string en un Objeto de DateTime

¿y para que querría hacer tal cosa? si alguien se hizo esta pregunta, excelente

Siempre hay que hacerse esa pregunta antes de volverse loco buscando el método que lo permita

En este caso recontra vale la pena la ecuación costo-beneficio de hacer esta conversión

El tema es que las fechas en modo "string" no me sirven más que para mostrar en pantalla, si quiero saber la diferencia de días, minutos o lo que fuera entre dos fechas, las necesito como objetos de fecha

```
hoy = dt_dt.today()  
expiracion_str = "2020-04-17"  
expiracion = dt_dt.strptime(expiracion_str, '%Y-%m-%d')  
expiracion  
  
datetime.datetime(2020, 4, 17, 0, 0)
```

*Nótese que como segundo parámetro de la función strftime() pusimos un string "%Y-%m-%d"

Como primer parámetro habíamos puesto la fecha en formato string que queríamos pasar a objeto

Mas adelante retomaremos con este tema pero ese segundo parámetro es en donde le decimos al intérprete como viene la fecha string que tenemos, ya que el intérprete no sabe adivinar si le pasamos primero el año, después el mes y el dia o si le pasamos primero el dia y después el mes, etc..

Ahora que ya tengo la fecha de hoy y la de expiración en el mismo tipo de objeto, puedo calcular la diferencia entre ambas como una simple resta

```
expiracion - hoy  
datetime.timedelta(days=34, seconds=26251, microseconds=602673)
```

Como vemos el resultado de esa resta es un objeto de la librería **timedelta** que como dijimos es un sub-paquetes de **datetime**

Ahora si queremos obtener los días o los segundos de ese objeto en una variables simplemente acudimos al atributo **days** o al atributo **seconds**

```
dias = (expiracion - hoy).days  
segundos = (expiracion - hoy).seconds  
print("Días restantes:", dias, "\nSegundos restantes:", segundos)
```

Días restantes: 36

Segundos restantes: 63156

Y si queremos acceder al string de la expiración que ya convertimos en objeto, podemos usar el método ctime()

```
dt_dt.ctime(expiracion)
```

```
'Fri Apr 17 00:00:00 2020'
```

Como les prometí dejamos colgado el tema de ese string raro "%Y-%m-%d" que le pasamos como argumento a la función strftime()

Veamos, como se imaginarán los % indican algún tipo de carácter especial que significa algo, por ejemplo %Y significa el año con 4 dígitos, el %d el número de día con dos dígitos y así..

Directivas de día, mes y año

Directiva	Significado	Ejemplo
%a	Día de la semana como nombre abreviado según la configuración regional.	<i>Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)</i>
%A	Día de la semana como nombre completo de la localidad.	<i>Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)</i>
%w	Día de la semana como un número decimal, donde 0 es domingo y 6 es sábado.	0, 1, ..., 6
%d	Día del mes como un número decimal relleno con ceros.	01, 02, ..., 31
%b	Mes como nombre abreviado según la configuración regional.	<i>Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)</i>
%B	Mes como nombre completo según la configuración regional.	<i>January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)</i>
%m	Mes como un número decimal rellenado con ceros.	01, 02, ..., 12
%y	Año sin siglo como un número decimal relleno con ceros.	00, 01, ..., 99
%Y	Año con siglo como número decimal.	0001, 0002, ..., 2013, 2014, ..., 9998, 9999

Directivas de zonas horarias

%z	Desplazamiento (<i>offset</i>) UTC en la forma <code>±HHMM[SS[.fffffff]]</code> (cadena de caracteres vacía si el objeto es naíf (<i>naive</i>)).	(vacío), +0000, -0400, +1030, +063415, -030712.345216
%Z	Nombre de zona horaria (cadena de caracteres vacía si el objeto es naíf (<i>naive</i>)).	(empty), UTC, GMT

Otras Directivas

Directiva	Significado	Ejemplo
%j	Día del año como un número decimal relleno con ceros.	001, 002, ..., 366
%U	Número de semana del año (domingo como primer día de la semana) como un número decimal llenado con ceros. Todos los días en un nuevo año anterior al primer domingo se consideran en la semana 0.	00, 01, ..., 53
%W	Número de semana del año (lunes como primer día de la semana) como número decimal. Todos los días en un nuevo año anterior al primer lunes se consideran en la semana 0.	00, 01, ..., 53
%c	Representación apropiada de fecha y hora de la configuración regional.	<i>Tue Aug 16 21:30:00 1988 (en_US); Di 16 Aug 21:30:00 1988 (de_DE)</i>
%x	Representación de fecha apropiada de la configuración regional.	08/16/88 (<i>None</i>); 08/16/1988 (<i>en_US</i>); 16.08.1988 (<i>de_DE</i>)
%X	Representación de la hora apropiada de la configuración regional.	21:30:00 (<i>en_US</i>); 21:30:00 (<i>de_DE</i>)

Ahora, ni ahí me acuerdo todos estos, lo importante siempre es saber como buscar este tipo de cosas, lo que se llama “aprender a aprender” es decir que ahora que ya saben que buscar y que es lo que van a encontrar, el ejercicio sería que prueben googleando o yendo a la documentación oficial para encontrar esto, veamos

Primera opción Google

- Python datetime doc

Eso te va a llevar a esta documentación:

<https://docs.python.org/es/3/library/datetime.html>

Yo se que al principio la documentación oficial puede parecer bastante intimidante, pero a no desesperar que es nuestro mejor aliado, en realidad no va a hacer mucha falta para los ejemplos de este libro pero si para que ustedes aprendan a aprender, y ahora que estamos con temas super fáciles me parece el mejor momento para ver esto, a continuación les pego una imagen de como se ve la doc oficial de Python, la pantalla inicial, luego veremos como llegar a la de datetime y luego ahí como encontrar las directivas de fechas y strings

Por suerte esta super bien documentado todo y en español así que no hay excusas!!

Como les decía la documentación oficial se ve así:

Les marco las tres opciones que vamos a usar,

- 1- Lenguaje, no comments
- 2- Versión: Es importante esto porque a veces leemos la documentación de una versión vieja y resulta que no nos andan las cosas luego, lo mejor es leer la mas actual pero asegurarse que también tengan la mas actual instalada en su sistema, pero si instalaron Python hace poco seguro tendrán la ultima o casi
- 3- Bueno, después está la navegación como cualquier página, pero en nuestro caso queríamos ver algo de la librería datetime, y esta es una librería nativa de Python, es decir una de las librerías que ya vienen con el programa, así que ese link que les marco de "Library Reference" es donde están todas las librerías nativas de Python, por si quieren ir chusmeando que mas hay para ir aprendiendo por su cuenta que les llame la atención.

Entonces, haciendo clic en las librerías llegamos a un listado laaaargo de librerías estándar que están divididas por temática entre las que hay una temática de fechas, otra de números y así.. les pego un recorte a continuación

Como ven en la de números está la librería random que ya hemos visto y la librería math que también vimos. Y en la de fechas vemos que tenemos por ejemplo la librería datetime

- Data Types
 - [datetime](#) — Basic date and time types
 - [zoneinfo](#) — IANA time zone support
 - [calendar](#) — General calendar-related functions
 - [collections](#) — Container datatypes
 - [collections.abc](#) — Abstract Base Classes for Containers
 - [heapq](#) — Heap queue algorithm
 - [bisect](#) — Array bisection algorithm
 - [array](#) — Efficient arrays of numeric values
 - [weakref](#) — Weak references
 - [types](#) — Dynamic type creation and names for built-in types
 - [copy](#) — Shallow and deep copy operations
 - [pprint](#) — Data pretty printer
 - [reprlib](#) — Alternate `repr()` implementation
 - [enum](#) — Support for enumerations
 - [graphlib](#) — Functionality to operate with graph-like structures
- Numeric and Mathematical Modules
 - [numbers](#) — Numeric abstract base classes
 - [math](#) — Mathematical functions
 - [cmath](#) — Mathematical functions for complex numbers
 - [decimal](#) — Decimal fixed point and floating point arithmetic
 - [fractions](#) — Rational numbers
 - [random](#) — Generate pseudo-random numbers
 - [statistics](#) — Mathematical statistics functions

Luego haciendo click en la librería datetime llegamos acá (que es el link que les pégue cuando les puse el ejemplo de lo que me sale googleando “Python datetime doc”

En fin, llegamos aca:

Python » Spanish ▾ 3.9.6 ▾ Documentation » La Biblioteca Estándar de Python » Tipos de datos »

Tabla de contenido

- datetime — Tipos básicos de fecha y hora
 - Objetos conscientes (*aware*) y naïfs (*naive*)
 - Constantes
 - Tipos disponibles
 - Propiedades comunes
 - Determinando si un objeto es Consciente (*Aware*) o Naïf (*Naive*)
 - Objetos `timedelta`
 - Ejemplos de uso: `timedelta`
 - Objeto `date`
 - Ejemplos de uso: `date`
 - Objetos `datetime`
 - Ejemplos de uso: `datetime`
 - Objetos `time`
 - Ejemplos de uso: `time`
 - Objetos `tzinfo`
 - Objetos `timezone`
 - Comportamiento
 - `strptime()` y `strftime()`
 - Códigos de formato `strftime()` y `strptime()`
 - Detalle técnico

datetime — Tipos básicos de fecha y hora

Source code: Lib/datetime.py

El módulo `datetime` proporciona clases para manipular fechas y horas.

Si bien la implementación permite operaciones aritméticas con fechas y horas, su principal objetivo es poder extraer campos de forma eficiente para su posterior manipulación o formateo.

Ver también:

Módulo calendar

Funciones generales relacionadas a *calendar*.

Módulo time

Acceso a tiempo y conversiones.

Paquete dateutil

Biblioteca de terceros con zona horaria ampliada y soporte de análisis.

Objetos conscientes (*aware*) y naïfs (*naive*)

Date and time objects may be categorized as «*aware*» or «*naive*» depending on whether or not they include timezone information.

Y lo que estábamos buscando está en el índice marcado como “Códigos de formato” así que último clic para llegar al cuadro ese que les pégue al principio, como ven son solo un par de clics y el mismo o análogo proceso va a repetirse cada vez que busquen algo de una librería nativa de Python, así que si bien no es necesario para seguir los ejemplos del libro de acá en mas es una práctica que sugiero que hagan por su cuenta lo de chusmear y acostumbrarse a buscar cosas en la documentación oficial.

Generar una fecha como objeto de datetime

Otra forma de generar una fecha como objeto de datetime en lugar de partir de un string puede ser partiendo de los valores de año, mes y día

```
año = 2020
mes = 12
dia = 24
hora = 23
minutos = 59
segundos = 59
fecha = dt.datetime(año,mes,dia,hora,minutos, segundos)
fecha, fecha.ctime()
```

```
(datetime.datetime(2020, 12, 24, 23, 59, 59), 'Thu Dec 24 23:59:59 2020')
```

Como ven genero tanto el objeto fecha, como el string con el método ctime() a partir del objeto

Convertir un objeto Datetime a string

Hay varias formas de convertir un objeto fecha a string

```
import datetime as dt
fecha = dt.datetime(2020,3,10)

str(fecha)
'2020-03-10 00:00:00'

fecha.isoformat()
'2020-03-10T00:00:00'

fecha.ctime()
'Tue Mar 10 00:00:00 2020'

fecha.strftime(format='%Y-%m-%d')
'2020-03-10'
```

Desde un simple: str(objeto_fecha_aca_adentro)

Hasta un método que me permite personalizar las directivas de formato de salida del string como el último ejemplo con objeto_fecha_a_transformar.strftime(formato_directivas_salida)

Distintos formatos para mostrar fechas

Claramente los formatos de fecha usados en bases de datos no siempre son cómodos de visualizar, de hecho, generalmente son muy incómodos, por lo que es común usar en interfaces de usuarios formatos más amigables a la lectura humana, para ello vamos a usar la función **strftime** para pasar el objeto fecha a un string cómodo de leer

```
from datetime import datetime as dt_dt
hoy = dt_dt.today()
hoy_txt = dt_dt.strftime(hoy, "%d %B, %Y")
hoy_txt
```

'13 March, 2020'

Seteo de parametrización local

La Librería **locale** nos permite setear a los usos locales los atributos, entre otras de

- Fechas y horas: locale.LC_TIME
- Signos y nomenclaturas monetarias: locale.LC_MONETARY
- Numeración y signos de puntuación: locale.LC_NUMERIC

```
import locale
from datetime import datetime as dt_dt

# Configurado en inglés
locale.setlocale(locale.LC_TIME, "en")
hoy_en = dt_dt(2020,3,10)
hoy_en = dt_dt.strftime(hoy_en, "%A %d %B, %Y")

# Configurado en francés
locale.setlocale(locale.LC_TIME, "fr")
hoy_fr = dt_dt(2020,3,10)
hoy_fr = dt_dt.strftime(hoy_fr, "%A %d %B, %Y")

# Configurado en español
locale.setlocale(locale.LC_TIME, "esp")
hoy_es = dt_dt(2020,3,10)
hoy_es = dt_dt.strftime(hoy_es, "%A %d de %B, %Y")

print(f"La fecha en inglés queda así: {hoy_en}")
print(f"La fecha en francés queda así: {hoy_fr}")
print(f"La fecha en español queda así: {hoy_es}")
```

```
La fecha en inglés queda así: Tuesday 10 March, 2020
La fecha en francés queda así: mardi 10 mars, 2020
La fecha en español queda así: martes 10 de marzo, 2020
```

¿Qué es un timestamp?

El timestamp más usado es el número de segundos que han transcurrido desde las 0 horas del 1 de enero de 1970 GMT.

O sea que es una cantidad de segundos que nos sirve para identificar una fecha y hora exactas con un simple número entero y su uso es muy común en informática ya que ocupa poco espacio en una base de datos y es universalmente conocido.

También se lo conoce como:

- Fecha "Epoch"
- Fecha UNIX o tiempo UNIX

Como curiosidad si quieren saber la fecha actual en este formato visiten

- <https://www.epochconverter.com/> (<https://www.epochconverter.com/>)
- <https://currentmillis.com/> (<https://currentmillis.com/>)

Obtener el timestamp de un objeto fecha

```
from datetime import datetime as dt_dt
hoy = dt_dt.today()
timestamp = hoy.timestamp()
print(timestamp)
```

1584131430.513358

Obtener el objeto fecha de un timestamp

```
timestamp = 1584131430.513358
fecha = dt_dt.fromtimestamp(timestamp)
fecha
```

datetime.datetime(2020, 3, 13, 17, 30, 513358)

Pasando a Otro huso horario

```
import pytz
from datetime import datetime
as dt_dt hoy = dt_dt.now()
hoy_berlin = hoy.astimezone(pytz.timezone('Europe/Berlin'))

print("La hora local es",hoy.ctime())
print("La hora de Berlin es",hoy_berlin.ctime())
```

La hora local es Sat Mar 14 18:37:52 2020

La hora de Berlin es Sat Mar 14 22:37:52 2020

Dejo comando para listar todas las zonas mas usuales, imprimo solo las primeras 10, pero si quieren ver todas pongan directamente

- pytz.common_timezones

```
pytz.common_timezones[0:10]
```

```
['África/Abidjan', 'África/Accra', 'África/Addis_Ababa', 'África/Algiers',
'África/Asmara', 'África/Bamako', 'África/Bangui', 'África/Banjul',
'África/Bissau', 'África/Blantyre']
```

¿Y qué otras funciones tienen esta librería datetime?

Muchas más, obvio no vamos a mostrar todas, pero acostúmbrense a visitar la documentación de las librerías que usan ya que muchas veces les va a simplificar la vida

En este caso se trata de una librería estándar de python así que vamos a encontrar su documentación en [docs.python.org](https://docs.python.org/3/library/datetime.html) pero si no fuera el caso nuestro mejor aliado siempre es Google

<https://docs.python.org/3/library/datetime.html> (<https://docs.python.org/3/library/datetime.html>)

Librería Calendar

Impresión en pantalla de un calendario

La función prcal() de la librería calendar nos permite imprimir un calendario para poder visualizar de modo amigable para un humano la distribución de las fechas en cada mes

```
import calendar

# Parametros
año = 2020
separacion_horiz = 1
separacion_vert = 1
calle = 3
meses_por_fila = 3

calendar.prCal(año, separacion_horiz, separacion_vert, calle, meses_por_fila)
```

2020

enero							febrero							marzo							
lu	ma	mi	ju	vi	sá	do	lu	ma	mi	ju	vi	sá	do	lu	ma	mi	ju	vi	sá	do	
1	2	3	4	5			1	2	3	4	5	6	7	1	2	3	4	5	6	7	8
6	7	8	9	10	11	12	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
13	14	15	16	17	18	19	10	11	12	13	14	15	16	9	10	11	12	13	14	15	
20	21	22	23	24	25	26	17	18	19	20	21	22	23	16	17	18	19	20	21	22	
27	28	29	30	31			24	25	26	27	28	29		23	24	25	26	27	28	29	

abril							mayo							junio						
lu	ma	mi	ju	vi	sá	do	lu	ma	mi	ju	vi	sá	do	lu	ma	mi	ju	vi	sá	do
1	2	3	4	5			1	2	3	4	5	6	7	1	2	3	4	5	6	7
6	7	8	9	10	11	12	4	5	6	7	8	9	10	8	9	10	11	12	13	14
13	14	15	16	17	18	19	11	12	13	14	15	16	17	15	16	17	18	19	20	21
20	21	22	23	24	25	26	18	19	20	21	22	23	24	22	23	24	25	26	27	28
27	28	29	30				25	26	27	28	29	30	31	29	30					

julio							agosto							septiembre						
lu	ma	mi	ju	vi	sá	do	lu	ma	mi	ju	vi	sá	do	lu	ma	mi	ju	vi	sá	do
1	2	3	4	5			1	2	3	4	5	6		1	2	3	4	5	6	
6	7	8	9	10	11	12	3	4	5	6	7	8	9	7	8	9	10	11	12	13
13	14	15	16	17	18	19	10	11	12	13	14	15	16	14	15	16	17	18	19	20
20	21	22	23	24	25	26	17	18	19	20	21	22	23	21	22	23	24	25	26	27
27	28	29	30	31			24	25	26	27	28	29	30	28	29	30				

octubre							noviembre							diciembre						
lu	ma	mi	ju	vi	sá	do	lu	ma	mi	ju	vi	sá	do	lu	ma	mi	ju	vi	sá	do
1	2	3	4				1		1	2	3	4	5	6	1	2	3	4	5	6
5	6	7	8	9	10	11	2	3	4	5	6	7	8	7	8	9	10	11	12	13
12	13	14	15	16	17	18	9	10	11	12	13	14	15	14	15	16	17	18	19	20
19	20	21	22	23	24	25	16	17	18	19	20	21	22	21	22	23	24	25	26	27
26	27	28	29	30	31		23	24	25	26	27	28	29	28	29	30	31			

O podemos imprimir solo un mes cualquiera

```
año = 2020  
mes = 3  
calendar.pmonth(año, mes)
```

```
March 2020  
Mo Tu We Th Fr Sa Su  
      1  
2 3 4 5 6 7 8  
9 10 11 12 13 14 15  
16 17 18 19 20 21 22  
23 24 25 26 27 28 29  
30 31
```

Años bisiestos

Por ejemplo, vamos a preguntar si un año es bisiesto o no

Dentro de calendar ya está predefinido un método que se llama isleap()

Y ese método admite un argumento (en este caso obligatorio que es el año en cuestión)

```
calendar.isleap(2020)
```

True

También podemos saber la cantidad de días/años bisiestos entre dos años

```
calendar.leapdays(2020, 2035)
```

4

Funciones básicas de calendar

Si queremos saber que día de la semana caerá una fecha (Por default está seteado 0=lunes, 1=martes. y así hasta 6=Domingo)

```
dia = calendar.weekday(2020, 3, 27)  
print(dia)
```

4

Sigue funcionando el seteo de la librería **locale** para las funciones de calendar

```
import locale

dia = calendar.weekday(2020, 3, 27)
locale.setlocale(locale.LC_TIME, "en")

calendar.day_name[dia]

'Friday'
```

```
locale.setlocale(locale.LC_TIME, "esp")
calendar.day_name[dia]
```

```
'viernes'
```

Función monthrange

Esta función me devuelve el número de día (lunes=0... domingo=6) del primer día del mes, y la cantidad de días del mes

```
locale.setlocale(locale.LC_TIME, "esp")
calendar.monthrange(2020, 3)
```

```
(6, 31)
```

Ante cualquier duda de los métodos de esta librería, ver la documentación oficial:

<https://docs.python.org/3/library/calendar.html> (<https://docs.python.org/3/library/calendar.html>)

Ejercicios

1 Hacer un script que devuelva en la variable año, el año actual

2 Escribir el código al que el usuario le asigne:

- día
- mes
- año

Y devuelva el nombre en español del día de la semana cae esa fecha

Usar para ello la librería calendar

2b- Hacer el mismo ejercicio pero en lugar de usar calendar, usar solo la librería datetime

3- Hacer un script que le informe al usuario cuantos días faltan para terminar el año

4- Hacer un script que le informe al usuario que fracción de año queda para la expiración de una opción que vence el 18 de diciembre del corriente año desde la fecha en que se ejecuta el script

- Que funcione sea cual sea el año que se ejecute el código
- Usar el sub-paquete date de la librería datetime

5- Hacer un script que el usuario ingrese en las variables mes y año los valores del mes y año para el cual se quiere saber el nombre del primer dia de ese mes

Respuestas

```
#=====
#       Ejercicio 1          #
#=====

from datetime import datetime

año = datetime.today().year
año

2020
```

```
#=====
#       Ejercicio 2          #
#=====

import calendar
import locale

locale.setlocale(locale.LC_TIME, "es")
año = 2018
mes = 12
dia = 9

diaSemana = calendar.weekday(año,mes,dia)
dia_nombre = calendar.day_name[diaSemana]
print(f"El dia para el {dia}/{mes}/{año} es un: ",dia_nombre.upper())

El dia para el 9/12/2018 es un: DOMINGO
```

```
#=====
#       Ejercicio 2b         #
#=====

from datetime import datetime
import locale

locale.setlocale(locale.LC_TIME, "es")
año = 2018
mes = 12
dia = 9

dia_ = datetime(año,mes,dia).strftime('%A')
print(f"El dia para el {dia}/{mes}/{año} es un: ",dia_.upper())

El dia para el 9/12/2018 es un: DOMINGO
```

```
#=====
#       Ejercicio 3          #
#=====

from datetime import datetime as dt_dt

hoy = dt_dt.now()
año = hoy.year
año_nuevo = dt_dt(año+1,1,1)

dias_faltantes = (año_nuevo - hoy).days
dias_faltantes
```

147

```
#=====
#       Ejercicio 4          #
#=====

from datetime import date

hoy = date.today()
año = hoy.year

expiracion = date(año,12,18)
dias_faltantes = (expiracion - hoy).days
fraccion_año = dias_faltantes/365
print(f"Hoy {hoy} faltan: {fraccion_año} años para el vencimiento del {expiracion}")

Hoy 2020-03-10 faltan: 0.7753424657534247 años para el vencimiento del 2020-12-18
```

```
#=====
#       Ejercicio 5          #
#=====

from datetime import date

mes = 5
año = 2020

dia_ = datetime(año,mes,1).strftime('%A')
print(f"El dia para el 1/{mes}/{año} es un: ",dia_.upper())

El dia para el 1/5/2020 es un:  VIERNES
```

Colecciones de datos en python

Lo que vamos a ver ahora son colecciones, es decir son tipos de datos que están pensados para guardar “conjuntos” de datos, series, sucesiones, como lo quieran llamar.. Como veremos hay tres tipos de colecciones en Python bien diferenciadas

- Colecciones tipo vectores/arrays:
 - o Listas
 - o Tuplas
 - o Sets
- Colecciones de clave/valor:
 - o Diccionarios
- Generadores: Ya los veremos tranquilos
 - o range() por ejemplo, pero hay muchos mas

Vamos a empezar con el primero tipo, que son simples estructuras que tienen un valor diferente en cada espacio interno, sin nominar, pero ordenados, veamos cada uno de estos por separado

Listas

Las listas son simplemente un conjunto de datos y se lo define entre corchetes con sus elementos separados con coma

```
lista = [1,2,3,4,"Juan","Pablo",True]
print(lista)

[1, 2, 3, 4, 'Juan', 'Pablo', True]
```

Como ven, en cada “compartimiento” puedo poner un tipo de datos diferente, y la cantidad de compartimientos la defino al definir la lista, puede ser la cantidad que quiera, a continuación, una representación gráfica/visual de los compartimientos de una lista, como verán el primer espacio es el número CERO, es decir no empieza con 1, sino con 0, y cada compartimiento es un número natural consecutivo al anterior:



Una lista vacía se la escribe así

```
lista_vacia = list()  
lista_vacia
```

[]

O así

```
lista_vacia = []  
lista_vacia
```

[]

Una lista con los números del 1 al 4 se la puede definir así:

```
lista = [1,2,3,4]  
print(lista)
```

[1, 2, 3, 4]

O así

```
lista = list([1,2,3,4])  
print(lista)
```

[1, 2, 3, 4]

Si quisiera poner dentro de la lista elementos de distinto tipo, como dijimos, no hay problema

```
listado = ["CERO",1,2,"tres",4.0,True]  
listado
```

['CERO', 1, 2, 'tres', 4.0, True]

¿Y qué pasa si le preguntamos al intérprete de que tipo es la variable listado?

```
listado = ["CERO",1,2,"tres",4.0,True]  
type(listado)
```

list

Obviamente es del tipo “lista”

Para acceder a un elemento cualquiera de una lista tengo que referirme al elemento entre corchetes Recordar siempre que **el primer elemento de una lista es el número CERO**

Ejemplo, accedamos al primer elemento de nuestro “listado”

```
listado = ["CERO",1,2,"tres",4.0,True]
print(listado[0])
```

CERO

Así como el primer elemento es el 0, si sigo para atrás al -1 es como empezar de derecha a izquierda desde el final, como si fuera circular la lista

```
listado = ["CERO",1,2,"tres",4.0,True]
print(listado[-1])
```

True

Por lo tanto si quisiera el anteúltimo, es como el segundo contando desde atrás hacia adelante, por eso llamo al elemento "-2"

```
listado = ["CERO",1,2,"tres",4.0,True]
print(listado[-2])
```

4.0

Slicing de una lista

Para recortar un "pedazo" de la lista accedo con el carácter ":" A la izquierda de los ":" va el "DESDE"

A la derecha de los ":" va el "HASTA"

Cabe aclarar que

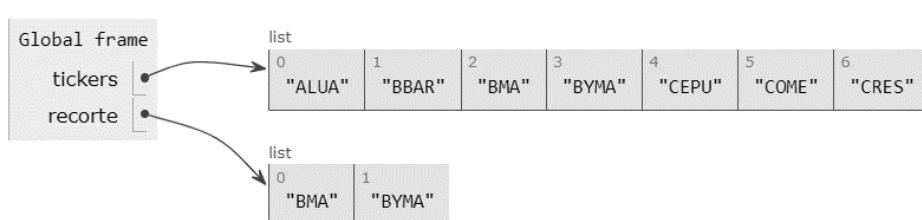
DESDE (inclusive) :HASTA (exclusive)

Recordemos siempre que el primer elemento es el 0, veamos un ejemplo:

```
tickers = ["ALUA", "BBAR", "BMA", "BYMA", "CEPU","COME","CRES"]
recorte = tickers[2:4]
print(recorte)

['BMA', 'BYMA']
```

Y así es como se visualiza cada "compartimiento" tanto de la variable "tickers" como de la variable "recorte"
Como el hasta es "exclusive" no se selecciona en recorte a "CEPU"



¿Qué pasa con [x:]?

Va desde el elemento **x** inclusive hasta el final

```
tickers = ["ALUA", "BBAR", "BMA", "BYMA", "CEPU", "COME", "CRES"]
recorte = tickers[2:]
print(recorte)
```

```
['BMA', 'BYMA', 'CEPU', 'COME', 'CRES']
```

Y se visualizarían así los “compartimientos”

Obviamente en la nueva variable se “resetean” las posiciones



¿Qué pasa con [:x]?

Va desde el 1º elemento de la lista hasta el número **x** no inclusive

```
tickers = ["ALUA", "BBAR", "BMA", "BYMA", "CEPU", "COME", "CRES"]
recorte = tickers[:4]
print(recorte)
```

```
['ALUA', 'BBAR', 'BMA', 'BYMA']
```

Y se visualizarían así los “compartimientos”



¿Qué pasa con [:]?

Son todos los elementos, desde el primero al último

```
tickers = ["ALUA", "BBAR", "BMA", "BYMA", "CEPU", "COME", "CRES"]
recorte = tickers[:]
print(recorte)
```

```
['ALUA', 'BBAR', 'BMA', 'BYMA', 'CEPU', 'COME', 'CRES']
```

¿Qué pasa con los números negativos [-x:-y]?

En lugar de contar de izquierda a derecha cuentan de derecha a izquierda, veamos ejemplos

Por ejemplo, si quisiera los dos últimos elementos de una lista podría hacer:

```
tickers = ["ALUA", "BBAR", "BMA", "BYMA", "CEPU", "COME", "CRES"]
recorte = tickers[-2:]
print(recorte)

['COME', 'CRES']
```

O ¿los últimos 4 pero sin el último?

```
tickers = ["ALUA", "BBAR", "BMA", "BYMA", "CEPU", "COME", "CRES"]
recorte = tickers[-4:-1]
print(recorte)

['BYMA', 'CEPU', 'COME']
```

Un Tercer argumento para el slicing: El paso [desde:hasta:paso]

El paso define cada cuanto avanza el cursor para tomar el siguiente elemento, por default sino se indica es claramente 1.

```
listado = [0,1,2,3,4,5,6,7,8,9,10,11,12]
listado[3::2]

[3, 5, 7, 9, 11]
```

En ese ejemplo le pido que vaya desde el 3º elemento hasta el final, y por último que vaya de a 2 en 2, es decir que, del 3, salta al 5 y así.

Veamos otros ejemplos

Si usamos por ejemplo `[:3]` al no poner ni desde ni hasta le decimos que agarre todos, y el 3 del final le estamos diciendo que vaya saltando de a 3 en 3

```
listado = [0,1,2,3,4,5,6,7,8,9,10,11,12]
listado[:3]

[0, 3, 6, 9, 12]
```

Por último, veamos un caso muy usado que es el `[::-1]`, como se imaginarán lo que hace es recorrer todo, pero al revés es decir devuelve la lista "dada vuelta"

```
listado = [0,1,2,3,4,5,6,7,8,9,10,11,12]
listado[::-1]
```

```
[12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Esto suele ser un truco bastante piola para “dar vuelta” una serie rápidamente.

Longitud de una lista

Para saber la cantidad total de elementos de una lista usamos la función nativa de Python `len()`

```
listado = ["CERO",1,2,"tres",4.0,True]
len(listado)
```

6

El típico mensaje de "Index out of range"

Cuando recorremos los elementos de una lista, lo primero que tenemos que pensar es que existan los elementos que busquemos porque si le pedimos a python que nos devuelva un numero de elemento que la lista no tiene, nos va a tirar un error y se va a cortar el programa en ese momento (ya que python no sabe que hacer)

```
listado = ["CERO",1,2,"tres",4.0,True]
print(listado[12])
```

```
IndexError Traceback (most recent call last)
<ipython-input-20-7876f73ae61a> in <module>
  1 listado = ["CERO",1,2,"tres",4.0,True]
----> 2 print(type(listado[12]))
```

```
IndexError: list index out of range
```

Cambiando algún valor de la lista

En el siguiente ejemplo le vamos a cambiar el segundo elemento (es decir el de índice 1, recuerden que arranca de 0)

```
listado      =      [ "GGAL" , "PAMP" , "YPFD" , "CEPU" , "EDN" , "LOMA" , "CRES" ]
listado[1]=[ "COME" ]
print(listado)
```

```
[ 'GGAL' , 'COME' , 'YPFD' , 'CEPU' , 'EDN' , 'LOMA' , 'CRES' ]
```

Cambiando varios valores de la lista por un solo valor

```
listado      =      [ "GGAL" , "PAMP" , "YPFD" , "CEPU" , "EDN" , "LOMA" , "CRES" ]
listado[1:3]=[ "COME" ]
print(listado)
```

```
[ 'GGAL' , 'COME' , 'CEPU' , 'EDN' , 'LOMA' , 'CRES' ]
```

Fíjense que listado[1:3] era una sublista de dos elementos (PAMP e YPFD) pero al remplazar esa sublista por “COME” me remplaza dos elementos por uno solo

Es decir, hace exactamente lo que le pedimos

Pero... ¿Y si quería cambiar los dos elementos por COME?

Pues, bueno, habría que haberle dicho exactamente eso, que a los dos les ponga “COME”, veamos:

```
listado      =      [ "GGAL" , "PAMP" , "YPFD" , "CEPU" , "EDN" , "LOMA" , "CRES" ]
listado[1:3]=[ "COME" , "COME" ]
print(listado)
```

```
[ 'GGAL' , 'COME' , 'COME' , 'CEPU' , 'EDN' , 'LOMA' , 'CRES' ]
```

Eligiendo valores al azar de una lista

Volvemos a usar la librería Random que usamos la clase pasada

```
import random
```

Elegimos 1 solo valor al azar

```
listado      =      [ "GGAL" , "PAMP" , "YPFD" , "CEPU" , "EDN" , "LOMA" , "CRES" ]
random.choice(listado)
```

```
'PAMP'
```

Elegimos "n" valores al azar, sin repetir, es decir "n" tiene que ser menor o igual a la cantidad de elementos de la lista, para ello usamos la función sample()

```
random.sample(listado, 3)
```

```
['YPFD', 'GGAL', 'CRES']
```

Reordenamos aleatoriamente la lista

```
random.shuffle(listado)  
print(listado)
```

```
['LOMA', 'GGAL', 'YPFD', 'CRES', 'CEPU', 'PAMP', 'EDN']
```

Ya vamos a ver más funciones de la librería random, pero por ahora sigamos con las tuplas

Fíense que curioso esto, cuando hicimos random.shuffle(listado) se alteró la variable listado, y ya quedó así alterada, es decir no nos permitió guardar la lista “sin alterar”, con lo cual ojo cuando usen estas funciones de ordenamiento y reordenamiento ya que por cuestiones de eficiencia todas funcionan igual, es decir que perdemos el original, con lo cual, si van a usar esta función hagan primero una copia de la lista original (digo si les interesa guardarla sin la alteración)

Tuplas

Las tuplas son una estructura similar a las listas, pero son inmutables, es decir que una vez que las definimos, ya no les podemos cambiar los valores

A diferencia de las listas las tuplas se escriben entre paréntesis

```
listado = ("GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES")  
type(listado)
```

```
tuple
```

```
listado = ("GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES")  
print(listado[1])
```

```
PAMP
```

Acostúmbrense a leer los errores, vamos a provocar el error de querer asignar un nuevo valor a un elemento de una tupla

```
listado = ("GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES")
listado[1]=[ "COME"]
```

```
TypeError Traceback (most recent call last)
<ipython-input-23-a13bdbe480fb> in <module>
      1     listado = ("GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES")
----> 2 listado[1]=[ "COME"]
```

TypeError: 'tuple' object does not support item assignment

Fíjense que el error que nos da es bastante entendible, nos dice básicamente que los objetos TUPLAS no soportan la asignación de valores, esto pasa porque como les decía las tuplas son inmutables, es decir una vez que tienen un valor cada “compartimiento” no se pueden modificar

¿Y para que voy a querer que una variable sea inmutable?

Bueno, a veces es super necesario asegurarse que los valores adentro de esa variable no me los modifique ninguna función y demás, por ahora el código que venimos viendo es muy sencillo pero a medida que avancemos vamos a ver que se va a ir complicando y muchas veces vamos a perder bastante el hilo de por donde va pasando cada variable y a veces vamos a querer asegurarnos que no se modifique, así que para eso sirven las tuplas. Sacando ese uso por lo general se usan más listas que tuplas.

¿Pero y si por “x” razón tengo una tupla y le quiero cambiar un valor? Python tiene un método para pasar de tupla a lista **list()** y viceversa **tuple()**

```
listado = ("GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES")
print(type(listado))
listado = list(listado)
print(type(listado))
listado = tuple(listado)
print(type(listado))

<class 'tuple'>
<class 'list'>
<class 'tuple'>
```

Métodos de Tuplas y Listas

Min y Max

```
cantidades = (4,2,5,6,8,10,3,5,7,4,2,6)
print(min(cantidades))
```

2

```
cantidades = (4,2,5,6,8,10,3,5,7,4,2,6)
print(max(cantidades))
```

10

Index

Obtenemos el índice dentro de la lista del elemento pasado como argumento (la primera aparición si esta repetido)

```
cantidades = (4,2,5,6,8,10,3,5,7,4,2,6)
cantidades.index(4)
```

0

Count

Cuenta la cantidad de apariciones de un determinado elemento en la lista o tupla

```
cantidades = (4,2,5,6,8,10,3,5,7,4,2,6)
cantidades.count(4)
```

2

Sum

Suma todos los elementos de la lista

```
cantidades = (4,2,5,6,8,10,3,5,7,4,2,6)
sum(cantidades)
```

62

Métodos de las listas

Sort

En el caso de las listas, no así para las tuplas, existe el método sort para reordenarlas

Al ser inmutables no se pueden modificar por eso las tuplas no lo admiten y las listas si

```
cantidadesLista = list(cantidades)
cantidadesLista.sort()
print(cantidadesLista)
```

[2, 2, 3, 4, 4, 5, 5, 6, 6, 7, 8, 10]

```
cantidadesLista.sort(reverse=True)
print(cantidadesLista)
```

[10, 8, 7, 6, 6, 5, 5, 4, 4, 3, 2, 2]

Append

Atenti acá que es uno de los métodos mas usados para listas

Agregando un valor al final de una lista

```
listado = ["GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES"]
print(listado)

listado.append("ALUA")
print(listado)
```

['GGAL', 'PAMP', 'YPFD', 'CEPU', 'EDN', 'LOMA', 'CRES']
['GGAL', 'PAMP', 'YPFD', 'CEPU', 'EDN', 'LOMA', 'CRES', 'ALUA']

Nótese que el método append() afecta a la lista, es decir, no hace falta decir algo así como:

- lista_con_agregado = lista.append("elemento_a_agregar")

Ya que con solo decir:

- lista.append("elemento_a_agregar")

La variable lista ya quedará automáticamente modificada con el elemento agregado

Extend

Agregado de una lista entera a otra

```
listado = ["GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES"]  
  
listadoPanelGral = ["BHIP", "CELU", "CTIO"]  
listado.extend(listadoPanelGral)  
print(listado)
```

```
['GGAL', 'PAMP', 'YPFD', 'CEPU', 'EDN', 'LOMA', 'CRES', 'BHIP', 'CELU', 'CTIO']
```

Insert

Agregado de un elemento en un lugar definido

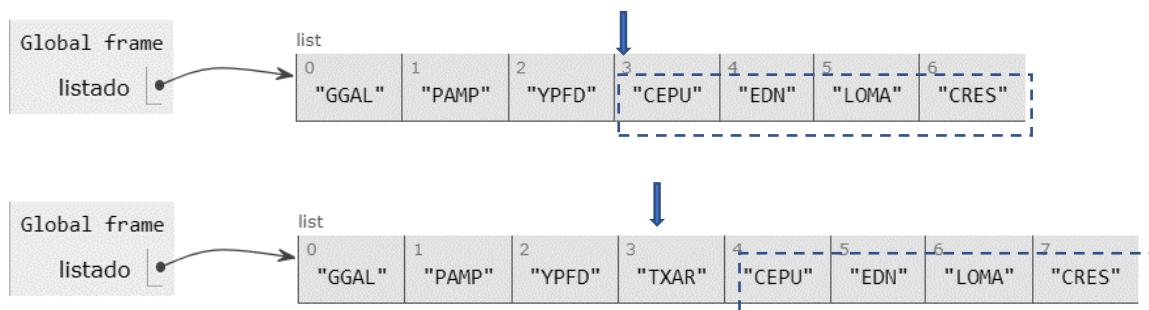
A veces queremos agregar un elemento a una lista pero no queremos que se agregue simplemente al final, sino que queremos que se agregue en una determinada posición, para ello usamos insert()

Veamos un ejemplo, insertemos “TXAR” en la posición 3

```
listado = ["GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES"]  
print(listado)  
  
listado.insert(3,"TXAR")  
print(listado)
```

```
['GGAL', 'PAMP', 'YPFD', 'CEPU', 'EDN', 'LOMA', 'CRES']  
['GGAL', 'PAMP', 'YPFD', 'TXAR', 'CEPU', 'EDN', 'LOMA', 'CRES']
```

Esquema visual:



Vemos como los últimos 4 tickers “se desplazan” a la derecha por la inserción en el espacio “3” de “TXAR”

Pop (muy interesante)

Atenti acá porque es un método muy pero muy útil porque me da dos funciones en una:

- Primero que nada quita el último elemento de la lista
- Pero no solo lo quita sino que me lo “devuelve”

Ahora ¿que es eso de que “devuelve” al último elemento?

Aun no vimos funciones, pero les adelanto que hay funciones que tienen “return” es decir devoluciones, y funciones que no las tienen, las funciones como print, sort, append, etc, no tienen devolución, es decir hacen algo pero no me devuelven nada, en cambio la función pop(), no solo va a quitar el último elemento de la lista sino que me lo va a devolver y eso me permite guardarlo en alguna otra variable, veamos todo esto en acción en un ejemplo mejor:

```
listado = ["GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES"]

# Aquí guardo en la variable "eliminado" el elemento extraido
eliminado = listado.pop()

print("El elemento eliminado es "+eliminado)
print("\nLa lista final es la siguiente")
print(listado)

El elemento eliminado es CRES

La lista final es la siguiente
['GGAL', 'PAMP', 'YPFD', 'CEPU', 'EDN', 'LOMA']
```

Como ven, guardo en la variable “eliminado” al ticker que extrae de la lista “listado” y todo en un solo paso, es decir altero la lista y guardo esa “devolución” en la variable “eliminado”

¿Y si en lugar de extraer el último quiero extraer el primero o cualquier otro?

La función pop() me permite pasar como argumento el número de elemento que quiero extraer, por defecto es el último pero puedo pasar el valor que quiera, veamos el ejemplo extrayendo el elemento “2”

```
listado = ["GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES"]

# Aquí guardo en la variable "eliminado" el elemento extraido
eliminado = listado.pop(2)

print("El elemento eliminado es "+eliminado)
print("\nLa lista final es la siguiente")
print(listado)

El elemento eliminado es YPFD

La lista final es la siguiente
['GGAL', 'PAMP', 'CEPU', 'EDN', 'LOMA', 'CRES']
```

Reverse

Da vuelta la lista, esto es interesante cuando nos viene una serie temporal en un orden y la queremos exactamente al revés

```
listado = ["GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES"]
listado.reverse()
print(listado)
```

```
['CRES', 'LOMA', 'EDN', 'CEPU', 'YPFD', 'PAMP', 'GGAL']
```

Clear

Borra todos los elementos de la lista

```
listado = ["GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES"]
listado.clear()
print(listado)
```

```
[]
```

Diccionarios

Un diccionario es una estructura Clave => Valor

Por ejemplo, en lugar de tener un listado de las notas de mis alumnos, podría tener un diccionario con la nota asignada al nombre de cada uno,a diferencia de las listas donde cada posición era una “ubicación indexada”, en los los diccionarios, cada posición es una ubicación “nominada” es decir con nombre

Los diccionarios, se escriben con la siguiente sintaxis y reglas:

- Se abre y cierra el diccionario con llaves {}
- Los elementos se separan entre si con comas ,
- Cada elemento del diccionario si o si debe tener una clave y un valor
- La clave puede ser un número pero por lo general es un string (nombre del elemento)
- El valor puede ser un número, un string, una lista o cualquier otro objeto, ya veremos
- Entre la clave y el valor se separan con un dos puntos :
- Las claves no se pueden repetir, si las repetimos en realidad sobreescrivimos la clave anterior

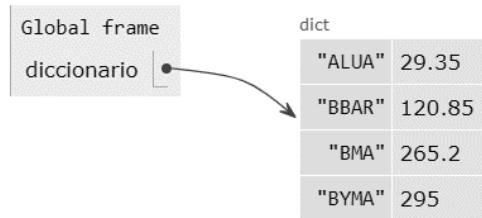
Tranquilos, ya se van a acostumbrar, veámoslo con un ejemplo directamente

Ejemplo, definamos un diccionario con los precios de algunos tickers:

```
diccionario = {"ALUA":29.35,"BBAR":120.85,"BMA":265.2,"BYMA":295}  
type(diccionario)
```

```
dict
```

Obviamente es un tipo nuevo (dict)
y veamos la representación visual:



Como ven, los elementos no están ordenados por un índice numérico, sino que cada elemento tiene su nombre (clave)

Veamos algo que quizá pasó medio desapercibido, que es que las claves no se pueden repetir porque son únicas, la pregunta es ¿Qué pasa si las repito entonces? Bueno, como les anticipé se sobreescrive ese nombre de elemento, veamos un ejemplo, definimos el mismo diccionario que antes, pero agregamos un último elemento donde repetimos la clave "ALUA" y le asignamos a este el valor 211.11

Vean lo que sucede:

```
diccionario = {"ALUA":29.35,"BBAR":120.85,"BMA":265.2,"BYMA":295, "ALUA":211.11}  
print(diccionario)
```

```
{'ALUA': 211.11, 'BBAR': 120.85, 'BMA': 265.2, 'BYMA': 295}
```

Como anticipábamos, el valor de la primera clave ALUA es sobreescrito con el 211.11

Acceso a valores puntuales de un diccionario

Dos Formas de Acceder a los valores de una clave de un diccionario

La mas usual es poniendo corchetes y llamando a una determinada clave:

```
print(diccionario['BBAR'])
```

```
120.85
```

La otra es usando el método get() que permite obtener el valor de una determinada clave:

```
print(diccionario.get('BBAR'))
```

```
120.85
```

Ahora ¿Cuál conviene?

Supongamos que sabemos que podría no estar el ticker buscado en el diccionario

Para ello conviene usar el método get, ya que tiene un valor por default en caso que el índice buscado no se encuentre para evitar que tire error el programa, ese valor por default es **None** es decir un dato nulo pero como decía no me tira error

```
print(diccionario.get('FRAN'))
```

None

Asimismo, puedo elegir el valor por default que me devuelva en caso de no encontrar el índice en el diccionario, es el segundo argumento que no es obligatorio

```
print(diccionario.get('FRAN',0))
```

0

Accedemos por separado a las Claves y Valores

Tenemos un método para acceder a todas las claves en una lista de claves: keys()

```
print(diccionario.keys())
```

```
dict_keys(['ALUA', 'BBAR', 'BMA', 'BYMA'])
```

Y por supuesto tenemos otro método para acceder a toda la lista de valores únicamente: values()

```
values = diccionario.values()
```

```
print(values)
```

```
dict_values([29.35, 120.85, 265.2, 295])
```

Si queremos tener los valores como una lista y no como un objeto de valores de diccionario:

```
values = list(values)
```

```
print(values)
```

```
[29.35, 120.85, 265.2, 295]
```

Lo mismo con el dict_keys, podemos hacer un list() de eso para tenerlo en una lista pura

Accedemos a los pares Clave => Valor

También podríamos querer tener los pares clave/valor en tuplas, para ello usaríamos el método `ítems()`

Veamos un ejemplo:

```
items = diccionario.items()  
print(items)  
  
dict_items([('ALUA', 29.35), ('BBAR', 120.85), ('BMA', 265.2), ('BYMA', 295)])
```

Y de todos los items como lista, podríamos acceder a cualquier item por ejemplo al primero:

```
items = diccionario.items()  
print(list(items)[0])  
  
('ALUA', 29.35)
```

Pop

Es lo mismo que el `pop()` para listas pero con elementos de un diccionario

Tomar un Elemento y borrarlo pasando la clave, funciona igual que en las listas, aquí devuelve solo el valor, no la clave

```
diccionario = {"ALUA":29.35, "BBAR":120.85, "BMA":265.2, "BYMA":295}  
itemTomado = diccionario.pop("BBAR")  
print(itemTomado)  
print(diccionario)
```

120.85

{'ALUA': 29.35, 'BMA': 265.2, 'BYMA': 295}

Update

Actualiza los valores de un diccionario según las coincidencias de las claves del segundo

```
diccionario1 = {"ALUA":29.35, "BBAR":120.85, "BMA":265.2, "BYMA":295}  
diccionario2 = {"BYMA":290, "CEPU":29, "COME":3, "CRES":40.7}  
  
diccionario1.update(diccionario2)  
print(diccionario1)  
  
{'ALUA': 29.35, 'BBAR': 120.85, 'BMA': 265.2, 'BYMA': 290, 'CEPU': 29, 'COME': 3, 'CRES': 40.7}
```

Insertar nuevo elemento a un diccionario

Insertar un nuevo elemento en realidad es tan sencillo como escribir el nombre del diccionario, abrir corchetes y definir la nueva clave, cerrar corchetes, y asignarle un valor

```
dic1 = {"ALUA":29.35, "BBAR":120.85, "BMA":265.2, "BYMA":295}  
dic1["CEPU"] = 29  
print(dic1)
```

```
{'ALUA': 29.35, 'BBAR': 120.85, 'BMA': 265.2, 'BYMA': 295, 'CEPU': 29}
```

Inicialización de Listas y Diccionarios

Si quisieramos agregar elementos a listas o diccionarios que no existen vamos a tener errores

```
dicNuevo["GGAL"] = 90  
dicNuevo
```

```
NameError Traceback (most recent call last)  
----> 1 dicNuevo["GGAL"] = 90  
2     dicNuevo  
NameError: name 'dicNuevo' is not defined
```

Esto pasa porque al momento de asignarle el primer valor python no sabe a qué nos referimos porque nunca lo nombramos antes

Para evitar este error debemos definirlos al inicio como una lista o diccionario vacío según corresponda

```
dicNuevo = {}  
dicNuevo["GGAL"] = 90  
dicNuevo
```

```
{'GGAL': 90}
```

```
listaNueva = []  
listaNueva.append("GGAL")  
listaNueva
```

```
['GGAL']
```

Asignación ligada

La asignación múltiple está permitida en python y es básicamente asignarles a muchas variables el mismo valor, pero ojo que no todo es lo que parece, veamos

Miren bien estas sencillas líneas de código:

```
1 lista1 = lista2 = lista3 = [1,2,3,4,5]
2 lista1.append('Hola')
3 lista3 = []
4
5 print(lista1, lista2, lista3)
```

```
[1, 2, 3, 4, 5, 'Hola'] [1, 2, 3, 4, 5, 'Hola'] []
```

¿Que tiene de raro?

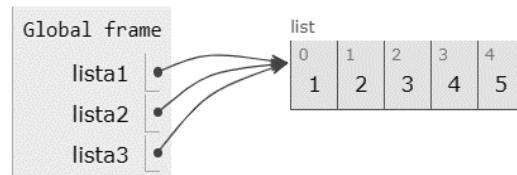
¿Por que la lista2 tambien tiene el "Hola" como ultimo elemento si solo se lo apendié a lista1?

Veamos línea a línea lo que hace el intérprete

Línea 1:

```
lista1 = lista2 = lista3 = [1,2,3,4,5]
```

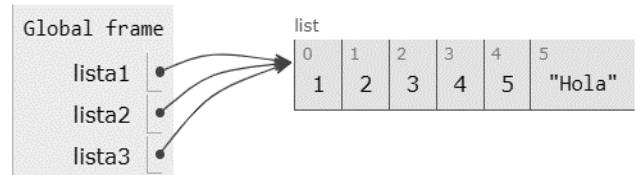
Vean bien, en realidad no genera “3 cajas” distintas, sino que las 3 variables apuntan al mismo objeto, es decir quedan “ligadas”



Línea 2:

```
lista1.append('Hola')
```

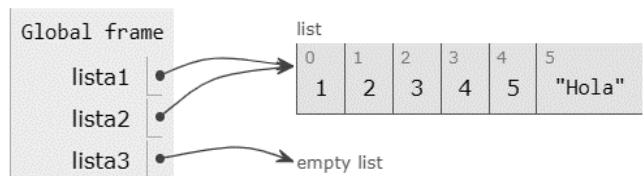
Acá al modificar “lista1” en realidad modifíco las 3 variables al mismo tiempo porque, como dijimos, había un solo objeto y las tres variables apuntaban al mismo, por lo que al modificar ese objeto, afectó a las 3 variables



Línea 3:

```
lista3 = []
```

Acá al redefinir lista3 como otro objeto nuevo, se “desliga” de los objetos de lista1 y lista2, ya que ahora lista3 apunta a otro espacio de memoria



Mas ejemplos de asignaciones ligadas

Miren ahora esto:

```
lista1 = lista2 = lista3 = [1,2,3,4,5]
lista4 = lista1
lista4.append('Hola')

print(lista1, lista2, lista3, lista4)

[1, 2, 3, 4, 5, 'Hola'] [1, 2, 3, 4, 5, 'Hola'] [1, 2, 3, 4, 5, 'Hola'] [1, 2, 3, 4, 5, 'Hola']
```

¿qué pasa?

Que al hacer: lista4 = lista1, ahora la lista4 tambien queda ligada a las otras 3 listas previas ya que todas apuntan siempre al mismo lugar de memoria porque estaban ligadas al mismo objeto

El método copy()

A veces se usa de mas este método y no es necesario, pero por ejemplo si en el ejemplo anterior que yo tengo 3 variables ligadas, si quisiera una cuarta variable igual a una de ellas pero no ligada, como en el ejemplo anterior era "lista4", la manera de lograrlo es con copy(), veamos:

```
lista1 = lista2 = lista3 = [1,2,3,4,5]
lista4 = lista1.copy()
lista4.append('Hola')

print(lista1, lista2, lista3, lista4)

[1, 2, 3, 4, 5] [1, 2, 3, 4, 5] [1, 2, 3, 4, 5] [1, 2, 3, 4, 5, 'Hola']
```

Como ven, al usar lista4 = lista1.copy(), lo que logro es independizar lista4 del resto, con lo cual luego el append solo afecta a lista 4 sin afectar a las otras, la diferencia visualmente es esta:

Sin usar copy() // Usando copy()



Esto es importante porque si no se usa copy() al decir que una lista es igual a otra, lo que haces no es solo una asignación sino que las estamos ligando (aun sin saberlo) esto porque el intérprete de Python optimiza los espacios en memoria, pero si queremos decir que una lista es igual a otra y no la queremos ligar ya saben, deben usar el .copy()

Generadores

Los generadores son justamente eso, “generadores” de colecciones, es decir son como instrucciones que me generarían una colección en forma implícita pero sin generarla, por ejemplo, si quisiera armar una lista con todos los números del 1 al 12 podría decir:

```
lista = [1,2,3,4,5,6,7,8,9,10,11,12]
```

No está mal, pero ¿si quisiera hacer lo mismo para números del 1 al 1250?

... Ni lo piensen jajaj

Para eso hay generadores, en Python para números simplemente usamos el método nativo range() que acepta 3 parámetros, similares a los de slices de listas

- Desde (Inclusive)
- Hasta (Exclusive)
- Intervalo/paso

Por ejemplo:

```
range(1,13,1)
```

Me generaría los números: 1,2,3,4,5,6,7,8,9,10,11,12

Pero momento, fíjense que dije “generaría” es decir que no genera nada en realidad, solo si los necesitara estarían generados, es decir que si quiero que ese rango se transforme en una lista con los números ya generados, debo decirle esto al intérprete, vean ustedes la diferencia:

```
rango = range(1,12,1)
print(rango)
```

```
range(1, 12)
```

```
rango = range(1,12,1)
lista = list(rango)
print(lista)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

En el primer caso, el rango es solo eso, un rango, no están los números que lo componen, sino, digamos, la definición del mismo. En cambio en el segundo caso, tenemos la lista desplegada, que como el hasta es exclusive, no incluye al 12.

Otro ejemplo, como ven no hacen falta variables intermedias, puedo convertir un rango en lista directo:

```
print(list(range(100,200,5)))
```

```
[100, 105, 110, 115, 120, 125, 130, 135, 140, 145, 150, 155, 160, 165, 170, 175, 180, 185, 190, 195]
```

El uso más común del generador range() es en realidad solo con el hasta, algo así

```
list(range(10))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Como no tiene desde, arranca en 0, y como no tiene paso por default es 1.

Esto de los generadores me lleva la siguiente nivel para definir colecciones..

Colecciones por comprensión

Este es un concepto similar a la definición de conjuntos, las colecciones como las listas las puedo definir enumerando cada elemento (por extensión) o definiendo en si al conjunto, veamos ejemplos mejor:}

Sucesión de primeros 10 número impares:

```
lista = [2*i+1 for i in range(10)]  
lista  
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Como ven, instancio una lista, y le digo que va a contener todos los $2*i+1$ para todo “i” dentro de los números que genera el generador range(10) que son, como vimos: 0,1,2,3,4,5,6,7,8,9

Esto es sumamente útil, porque me permite en una línea definir conjuntos con lógicas muy diversas, veamos más ejemplos:

Ejercicios de colecciones I

Dada la siguiente lista

```
listado = [1,2,3,4,5,6,7,8,9,10,11,12]
```

1 – filtrar solo los números pares

2 – Filtrar solo los valores del 4 hasta el final de la lista

3 – Filtrar solo los valores del 5 hasta el 10

4 – Filtrar solo los valores menores a 10 que sean pares

5 – Filtrar solo los múltiplos de 3 ordenados al revés

6 - Calcular el promedio de todos los números (usando las funciones sum y len)

Dadas las siguientes listas:

```
líderes = ["GGAL","PAMP","YPFD","TECO2","EDN","LOMA","BBAR"]
galpones = ["AGRO","FIPL","MIRG","GARO","LONG"]
```

7 - Elegir un galpón al azar y agregarlo a la lista líderes al final

8 - Elegir un galpón al azar y remplazar a EDN por ese galpón

9 - Elegir 3 galpones al azar y remplazar una líder cualquiera por esos dos galpones

Dado el siguiente diccionario:

```
panel = {'ALUA': 29.35, 'BBAR': 120.85, 'BMA': 265.2, 'BYMA': 290, 'CEPU': 29, 'COME': 3, 'CRES': 40.7}
```

10 - Remplazar el precio de BBAR por un precio al azar que elija python que varíe +/- \$1 del precio que tiene (con precisión de 0,01)

11 - Remplazar el precio de BBAR por un precio al azar que elija python que varíe +/- 3% del precio que tiene (con precisión de 0,01)

12 - Elegir un ticker al azar y mostrarlo

13 - Re-ordenar el diccionario al azar

14 - Re-ordenar el diccionario alfabéticamente en forma inversa (de la Z a la A)

Respuestas ejercicios de colecciones I

```
#-----#
# Rta Ejercicio 1 #
# ----- #

listado = [1,2,3,4,5,6,7,8,9,10,11,12]
listado[1::2]
```

[2, 4, 6, 8, 10, 12]

```
#-----#
# Rta Ejercicio 2 #
# ----- #

listado = [1,2,3,4,5,6,7,8,9,10,11,12]
listado[3:]
```

[4, 5, 6, 7, 8, 9, 10, 11, 12]

```
#-----#
# Rta Ejercicio 3 #
#----- #

listado = [1,2,3,4,5,6,7,8,9,10,11,12]
listado[4:10]
```

[5, 6, 7, 8, 9, 10]

```
#-----#
# Rta Ejercicio 4 #
# ----- #

listado = [1,2,3,4,5,6,7,8,9,10,11,12]
listado[1:-3:2]
```

[2, 4, 6, 8]

```
#----- #
# Rta Ejercicio 5 #
#----- #

listado = [1,2,3,4,5,6,7,8,9,10,11,12]
listado[-1::-3]
```

```
[12, 9, 6, 3]
```

```
#----- #
# Rta Ejercicio 6 #
#----- #

listado = [1,2,3,4,5,6,7,8,9,10,11,12]
sum(listado)/len(listado)
```

```
6.5
```

```
#-----#
# Rta Ejercicio 7 #
#----- #

import random

lideres = ["GGAL", "PAMP", "YPFD", "TECO2", "EDN", "LOMA", "BBAR"]
galpones = ["AGRO", "FIPL", "MIRG", "GARO", "LONG"]
galpon = random.choice(galpones)
lideres.append(galpon)

lideres

['GGAL', 'PAMP', 'YPFD', 'TECO2', 'EDN', 'LOMA', 'BBAR', 'LONG']
```

```
#-----#
# Rta Ejercicio 8 #
# ----- #

lideres = ["GGAL", "PAMP", "YPFD", "TECO2", "EDN", "LOMA", "BBAR"]
galpones = ["AGRO", "FIPL", "MIRG", "GARO", "LONG"]
galpon = random.choice(galpones)
lideres[4] = galpon
lideres

['GGAL', 'PAMP', 'YPFD', 'TECO2', 'LONG', 'LOMA', 'BBAR']
```

```

#-----#
# Rta Ejercicio 9 #
# ----- #

lideres = ["GGAL", "PAMP", "YPFD", "TECO2", "EDN", "LOMA", "BBAR"]
galpones = ["AGRO", "FIPL", "MIRG", "GARO", "LONG"]
galpon_x3 = random.sample(galpones,3)
indice = random.randrange(0,5)
lideres[indice] = galpon_x3
lideres

```

```
['GGAL', 'PAMP', 'YPFD', ['LONG', 'GARO', 'AGRO'], 'EDN', 'LOMA', 'BBAR']
```

```

#-----#
# Rta Ejercicio 10 #
# ----- #

import random
panel = {'ALUA': 29.35, 'BBAR': 120.85, 'BMA': 265.2, 'BYMA': 290, 'CEPU': 29}
valorAzar = random.uniform(-1,1)
panel['BBAR'] = round(panel['BBAR'] + valorAzar,2)
panel

```

```
{'ALUA': 29.35, 'BBAR': 121.55, 'BMA': 265.2, 'BYMA': 290, 'CEPU': 29}
```

```

#-----#
# Rta Ejercicio 11 #
# ----- #

import random
panel = {'ALUA': 29.35, 'BBAR': 120.85, 'BMA': 265.2, 'BYMA': 290, 'CEPU': 29}

banda = 0.03 * panel['BBAR']
valorAzar = random.uniform(-banda,banda)
panel['BBAR'] = round(panel['BBAR'] + valorAzar,2)
panel

```

```
{'ALUA': 29.35, 'BBAR': 117.37, 'BMA': 265.2, 'BYMA': 290, 'CEPU': 29}
```

```

#-----#
# Rta Ejercicio 12 #
#----- #

import random

panel = {'ALUA': 29.35, 'BBAR': 120.85, 'BMA': 265.2, 'BYMA': 290, 'CEPU': 29}

tickers = panel.keys()
random.choice(list(tickers))

```

'BYMA'

```

#-----#
# Rta Ejercicio 13 #
#----- #

panel = {'ALUA': 29.35, 'BBAR': 120.85, 'BMA': 265.2, 'BYMA': 290, 'CEPU': 29}

keys = list(panel.keys())
random.shuffle(keys)

listaReordenada = [(key, panel[key]) for key in keys]
dict(listaReordenada)

{'BYMA': 290, 'BBAR': 120.85, 'ALUA': 29.35, 'CEPU': 29, 'BMA': 265.2}

```

```

#-----#
# Rta Ejercicio 14 #
#----- #

panel = {'ALUA': 29.35, 'BBAR': 120.85, 'BMA': 265.2, 'BYMA': 290, 'CEPU': 29}

keys = list(panel.keys())
keys.sort()
keys.reverse()

listaReordenada = [(key, panel[key]) for key in keys]
dict(listaReordenada)

{'CEPU': 29, 'BYMA': 290, 'BMA': 265.2, 'BBAR': 120.85, 'ALUA': 29.35}

```

Decisiones

Las expresiones de comparación siempre devuelven True o False, no hay más posibilidades, y esa devolución dependerá de lo que comparemos

Operadores Relacionales:

- IGUAL ==
- DISTINTO !=
- Mayor >
- Mayor o igual >=
- Menor <
- Menor o igual <=

```
COME = 3
```

```
GGAL = 100
```

Una vez definidas las variables "a" y "b" vamos a ver que devuelven las expresiones de comparación

```
COME > GGAL
```

```
False
```

```
COME <= GGAL
```

```
True
```

Ojo con cómo se pregunta por una igualdad, es muy común que nos pase de querer comparar si "a" es igual a "b" y escribirlo con un solo signo igual.

Pero ¿qué pasa si preguntamos eso poniendo un solo signo igual?

Lo que pasaría en ese caso es que a la variable **a** le estaríamos asignando el valor de la variable **b**

```
COME == GGAL
```

```
False
```

```
COME != GGAL
```

True

¿Y qué pasa si comparamos un número con un texto? apuesto que siempre va a tirar error...

```
COME > "COME"
```

TypeError

```
Traceback (most recent call last) <ipython-input-11-9ae4ff098aa0> in <module>
----> 1 COME > "COME"
```

TypeError: '>' not supported between instances of 'int' and 'str'

¿Pero siempre da error comparar un número con un string?

```
COME != "COME"
```

True

mira vos? y sí. le estoy preguntando si son distintos y efectivamente son distintos.

```
COME == "COME"
```

False

¿Y si comparamos un número con un booleano?

```
COME > True
```

True

aaapa, bueno, sorpresa? el tema es que True es como sinónimo de 1 y False como sinónimo de 0 Es decir que en esa comparación está diciendo que 3>1 por eso me devuelve verdadero

```
COME == True
```

False

```
1 == True
```

True

```
1 <= True
```

True

```
0 == False
```

True

ojo con esto, este tipo de cosas suele generar ese tipo de bugs o fallas muy difíciles de encontrar

¿Y qué pasa con los valores nulos en las comparaciones?

```
COME == None
```

False

```
COME > None
```

TypeError

```
Traceback (most recent call last) <ipython-input-42-52baf6bf6235> in <module>
      ----> 1 COME > None
```

```
TypeError: '>' not supported between instances of 'int' and 'NoneType'
```

Ojo con esto que puede traer a confusión, si bien el valor nulo "None" es evaluado como falso (ya lo veremos en unos minutos), si evaluó comparar una variable nula con el 0 me devuelve falso, porque Python no tiene comparación de igualdad estricta, por lo tanto, la igualdad es estricta de por si

```
0 == None
```

False

Sentencia IF

Obviamente la sentencia IF lo que hace es preguntar si se cumple o no una condición

En caso que la condición preguntada sea **True** ejecuta la línea que sigue después de los DOS PUNTOS, caso negativo termina ahí si no hay else, pero si hubiera un else, ejecuta la línea o líneas que siguen después de los DOS PUNTOS del else

```
1 Estructura del IF
2
3 if (condicion):
4
5     1º linea a ejecutar si condición es verdadera
6     2º linea a ejecutar si condición es verdadera
7     .....
8     nº linea a ejecutar si condición es verdadera
9
10 else:
11
12     1º linea a ejecutar si condición es falsa
13     2º linea a ejecutar si condición es falsa
14     .....
15     nº linea a ejecutar si condición es falsa
16
```

Concepto de la identación

Como habrán observado, las líneas no están ordenadas todas hacia la izquierda, sino que dentro de cada “**BLOQUE**” de código hay como una especie de sangría.

Esa sangría no es sólo estética, aunque también ayuda a la lectura humana del bloque, sino que cumple una función que es indicarnos que instrucción o conjunto de instrucciones están dentro de cada bloque y que instrucción no.

Llamamos indentación a esa especie de sangría o espacio a la izquierda del texto que en programación nos sirve para dar jerarquía al orden en que el programa debe leer cada línea

En python es super fundamental el concepto de indentación porque de no respetarlo el intérprete nos va a dar un error de sintaxis, por ejemplo, fíjense en el ejemplo de estructura del IF, si ven todas las líneas que se ejecutan cuando la condición es verdadera, notarán que están todas identadas al mismo nivel, es como una manera de agrupar todas estas líneas, en otros lenguajes como javascript o php no importa la indentación, pero se delimitan los bloques con llaves por ejemplo

Lo importante es siempre recordar al salir del else, que si no respetamos la indentación y ponemos una línea que debe ejecutarse dentro del else sin la indentación esa línea se ejecuta siempre ya que es la primera línea luego de terminar la sentencia if, ya sea que haya entrado por el if o por el else

Veamos algún ejemplo básico

```
if True:  
    print("-Entra por verdadero")  
    print("-Segunda linea a ejecutarse por haber entrado por verdadero")  
else:  
    print("-Entra por falso")  
  
print("-Esta línea la imprime siempre, esta identada en el mismo nivel que el IF")
```

- Entra por verdadero
- Segunda línea a ejecutarse por haber entrado por verdadero
- Esta línea la imprime siempre esta identada en el mismo nivel que el IF

Evaluación de True o False en distintos tipos de Variables

Veamos esto con algunos ejemplos

Cabe preguntarse ¿qué pasa si preguntamos por una variable por ejemplo Nula?

En el caso de Python las variables nulas las evalúa como Falsas

```
variableNula = None  
if variableNula:  
    print("Evalúa Verdadero")  
else:  
    print("Evalúa Falso")
```

Evalúa Falso

¿Y si preguntamos por una variable = 0?

Las variables iguales a 0 python también las evalúa como falsas

```
variableCero = 0

if variableCero:
    print("Evalúa Verdadero")
else:
    print("Evalúa Falso")
```

Evalúa Falso

Otro tipo de comparaciones

Un tipo de comparación muy común en entre una variable y una constante, un operador de mayor o menor o mayor o igual o menor o igual

```
COME = 3

if COME < 4:
    print("Evalua Verdadero")
else:
    print("Evalua Falso")
```

Evalúa Verdadero

También podemos comparar entre dos variables predefinidas anteriormente:

```
GGAL = 12
stopLoss = 11
if (GGAL <
    stopLoss) :
print("Vender ya")
else:
    print("Sigo Adentro")
```

Sigo Adentro

Es muy común simplificar preguntando solo si la variable es verdadera y definir la comparación antes:

```
GGAL = 12
stopLoss = GGAL < 11
if stopLoss :
    print("Vender ya")
else:
    print("Sigo Adentro")
```

Sigo Adentro

Ustedes dirán ¿cuál es la ventaja de esta segunda opción?

Imaginen que en lugar de un stopLoss fijo, tenemos un trailing StopLoss, que como saben se va modificando a cada rato, en ese caso tendríamos que meter adentro de la pregunta del IF toda la lógica del stopLoss que mientras más compleja sea más complicaría la lectura, sin embargo es más sencillo definir aparte la variable stopLoss a la cual incluso, como más adelante veremos se le puede asignar la salida de una función y en el IF solo preguntar si se activó o no

Tiene una segunda ventaja no menor, y es la performance, en general vamos a ver muchos ifs dentro de bucles o ciclos, ya lo veremos en breve, que se repiten varias veces, y si cada vez que preguntamos tenemos que "evaluar" si es verdadero o falso, estamos perdiendo milisegundos que pueden ser importantes en el agregado, así que siempre es buena práctica definir la evaluación antes en una variable booleana y preguntar simplemente por esa variable

O sea que SIEMPRE busquen la forma de definir las comparaciones antes y llegar al if con un true o false

Otros operadores útiles

Un operador muy útil para clarificar el código es el operador **is not** que significa obviamente "no es"

```
accion = "GGAL"
if accion is not "GGAL":
    print("No es GGAL")
else:
    print("Es GGAL")
```

Es GGAL

Otro operador extremadamente útil es el operador **in** que me va a servir para saber si un valor está o no dentro de una lista

```
accion = "YPF"
cartera = ["AAPL", "AMZN", "FB"]
if accion in cartera:
    print(F"{accion} está en la cartera")
else:
    print(F"{accion} NO está en la cartera")
```

YPF NO está en la cartera

También se puede usar el operador con el **not** adelante como cualquier otro operador

```
accion = "GGAL"
cartera = ["AAPL", "AMZN", "FB"]
if accion not in cartera:
    print(F"Ojo que ${accion} NO está en la cartera")
else:
    pass
```

Ojo que \$GGAL NO está en la cartera

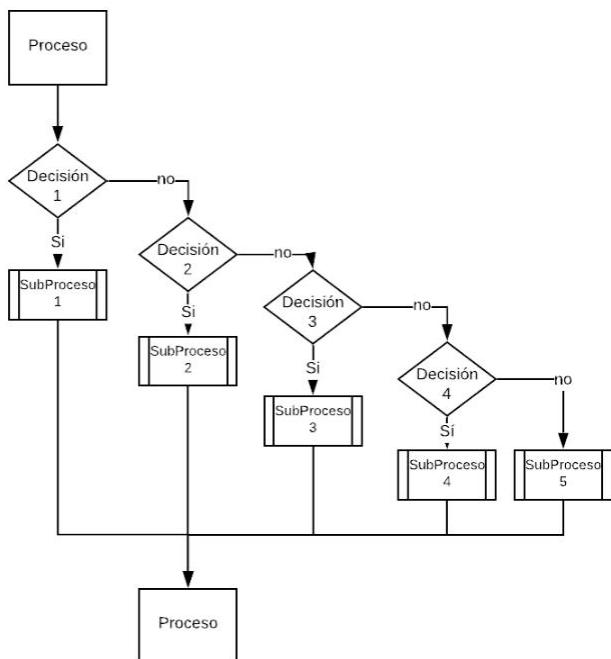
Como verán la gran ventaja de usar este tipo de operadores es la claridad del código que se entiende por si solo

Obviamente el “else: pass” no hace falta, ya que no es obligatoria la parte del ELSE, es normal poner el bloque igual con un “pass” para dejarlo armado con algún comentario adentro como para no olvidarse luego de completar esa parte de la lógica de lo que estemos programando.

Como aclaración vale decir que no es lo mejor usar negadores a menos que semánticamente realmente tenga sentido, por ejemplo si voy a tomar la decisión de comprar algo solo si “NO está” previamente en la cartera, ese si es un buen ejemplo trivial de uso, pero el problema de los negadores es que a medida que el código evoluciona es común encontrarse con dobles negaciones (no en el código per se) por ejemplo preguntar si no se cumple algo, y lo que hay que hacer en tal caso esta en el ELSE, y en la evaluación verdadera del IF un simple paso o impresión etc..

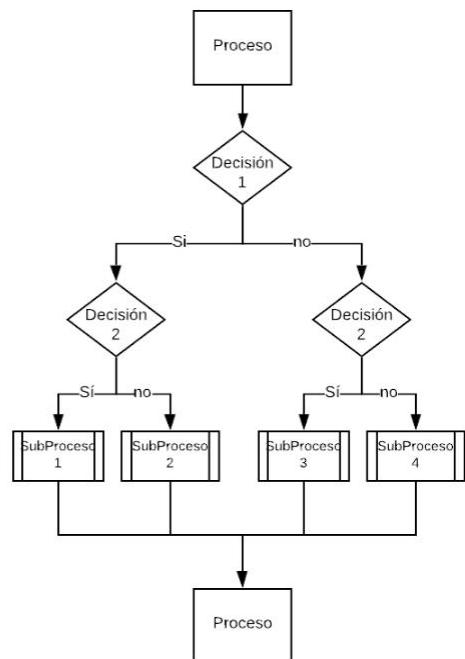
Decisiones concatenadas y anidadas

Decisiones Consecutivas o Concatenadas



Subprocesos = Decisiones + 1

Decisiones Anidadas



Subprocesos = $2^{\text{Decisiones}}$

Decisiones Consecutivas

Las decisiones consecutivas son muy útiles cuando tenemos que ir preguntando algo que en lugar de ser blanco/negro o si/no se va dividiendo como en grupos o escalas, por ejemplo, para rangos de malo, regular, bueno, muy bueno, excelente, vamos a evaluar una variable consecutivamente con los límites de esos grupos, veamos primero la estructura de las decisiones consecutivas:

```

if condicion_1:
    acciones si condicion_1 es verdadera

elif condicion_2:
    acciones si condición_2 es verdadera

#.....

elif condicion_n:
    acciones si condicion_n es verdadera

else:
    acciones si no fue verdadera ninguna de las condiciones anteriores

sigue el programa
  
```

Ejemplo típico de decisiones consecutivas

```
merval = 1800
if (merval>1500):
    print("All in Argy")
elif (merval>1000):
    print("Es por acá")
else:
    print("Piña historica")
```

All in Argy

Decisiones Anidadas

Las decisiones anidadas son ni más ni menos que una decisión adentro de otra

Veamos un ejemplo:

```
merval = 1200
usd = 60

if (merval>800):
    if (usd < 40):
        print("All in Argy")
    else:
        print("All in Brasil")
else:
    if (usd < 70):
        print("Piña historica")
    else:
        print("Yo te avisé")
```

All in Brasil

Operadores Lógicos

Tablas de VERDAD

Las tablas de verdad son solo esquemas de varios tipos de condiciones y operadores, por ejemplo, con el operador **and** el resultado de unir dos condiciones A y B será verdaderos solo si ambas (AyB) son verdaderas

Por otro lado, si uno dos condicionas A o B con el operador **or**, su resultado será verdadero cuando ambas son verdaderas, pero también cuando una de ellas es verdadera y la otra no.

Tabla de Verdad and

A	and B	Resultado
F	F	F
F	V	F
V	F	F
V	V	V

Tabla de Verdad or

A	Or	B	Resultado
F	F	F	F
F	V	V	V
V	F	V	V
V	V	V	V

En realidad las tablas de verdad no se usan nunca así como elemento teórico, como recurso pedagógico uno las muestra para que se entienda la lógica implícita en los operadores and, or y las comparaciones pero lo cierto es que uno se tiene que acostumbrar a “pensar como una maquina lógica” y asimilar estos conceptos de comparación binaria lo mas humanamente posible para que nos sea natural

Veamos todo esto en unos ejemplos

Supongamos un caso muy típico, que tenemos un par de indicadores técnicos (partamos de la base de algo bien sencillo, dos medias móviles) si una media móvil rápida es mayor a una media móvil lenta digamos que es un indicador de compra y asociamos esa comparación a una variable "buySignal"

Luego vamos a suponer que para comprar necesitamos preguntar si hay liquidez suficiente en la cuenta, por lo tanto para gatillar una orden de compra vamos a tener que preguntar si el saldo de la cuenta es mayor a un determinado límite mínimo y si es afirmativo definir como True la variable liquidez

```
smaFast = 20.4
smaSlow = 20
price = 19
saldo = 15000

buySignal = smaFast > smaSlow
liquidez = saldo > 1000

buy = buySignal and liquidez
print(buy)
```

True

Ahora a nuestra lógica le vamos a agregar el estado "hold" que se va a mantener como True siempre que haya dado señal de compra y siempre y cuando el precio actual no sea menor a mi stopLoss

También se puede usar el operador **not** que lo que hace es preguntar por lo opuesto

```
smaFast = 20.4
smaSlow = 20
price = 19
saldo = 15000
stopLossPrice = 18

buySignal = smaFast > smaSlow
liquidez = saldo > 1000
stopLoss = price <= stopLossPrice

hold = buySignal and liquidez and not stopLoss
print(hold)
```

True

Manejo básico de errores, Try/Except

La sentencia Try/Except se usa para evitar que el script se corte inesperadamente por una cuenta o una instrucción que da error porque el intérprete no puede resolverla

Recordemos que el flujo de información en un script es continuo y cualquier error que el intérprete no sepa resolver va a cortar ese flujo y va a impedir que el script termine su rutina, por lo tanto, este método try/except nos va a permitir "intentar" una instrucción que sospechamos puede dar error, y si ocurre el error realizar la instrucción que le pasamos en el "except"

Es como advertirle al intérprete, "ojo que te mando esta instrucción dudosa, fíjate primero si no es para kilombo" y ahí el intérprete en lugar de seguir de una el flujo normal del script, lo que hace es ver si falla y si falla se va por el otro lado (el del except)

Veámoslo con el típico error de la división por cero, hay veces que no sabemos si nos va a venir un dato vacío o no, y ahí es útil usar este método:

```
volumen = 1200
dias = 0

try:
    volMedio = volumen / dias
except:
    volMedio = None

print(volMedio)
```

None

```
volumen = 1200
dias = 0
volMedio = None

try:
    volMedio = volumen / dias
except:
    pass

print(volMedio)
```

None

También existe la instrucción **pass** que lo que hace es seguir de largo

Y por último también existe la instrucción **finally** que lo que hace es ejecutarse independientemente de si salió por el **try** o por el **except**

Como verán en los ejemplos pasa siempre por el bloque finally, independientemente si haya habido o no un error.

Probablemente se estarán preguntando para que sirve el bloque finally si podría haber puesto esas líneas (El print "por acá pasa siempre") al mismo nivel que el except al terminar ese bloque

Les cuento que yo me pregunté lo mismo cuando vi esto y tardé bastante en entender el por qué..

Es una cuestión más bien de prolijidad, es exactamente lo mismo poner un bloque de instrucciones en el finally o ponerlo después del bloque except sin más,, Pero queda más prolijo y entendible cuando todo el bloque Try/Except/Finally tiene una lógica asociada, por ahora les parecerá una pavada, es normal que piensen eso, como les digo me pasó lo mismo, pero estos detalles a la larga hacen el código más limpio y mantenable

```
volumen = 1200
dias = 2

try:
    volMedio = volumen / dias
except:
    volMedio = None
finally:
    print("por aca pasa siempre")

print(volMedio)
por acá pasa siempre
600.0
```

Ahora al mismo Código vamos a ponerle como variable la cantidad de días = 0 para que no pueda hacer la division y veremos que igualmente ejecuta el bloque del finally por mas que haya pasado por el except

```

volumen = 1200
dias = 0

try:
    volMedio = volumen / dias
except:
    volMedio = None
finally:
    print("por acá pasa siempre")

print(volMedio)

```

por acá pasa siempre

None

Catching, capturando el tipo de error

Además de salir por el except evitando el corte del script por un error, también podemos "capturar" el error que salta en el try

Para ello debemos importar la librería sys que es una librería básica de python

Y como verán en el bloque except capturamos el error usando el método exc_info() de la librería sys que importamos antes.

```

import sys

volumen = 1200
dias = 0

try:
    volMedio = volumen / dias
except:
    error = sys.exc_info()[1]

print("Error detectado:",error)

```

Error detectado: division by zero

Otra forma de capturar el error es usando la clase Exception en el bloque, el código sería el siguiente, en este caso genero un error diferente intentando, en lugar de dividir por ceo, dividir por un string, que obviamente me va a devolver un error de tipos.

```
volumen = 1200
dias = "hola"

try:
    volMedio = volumen / dias
except Exception as e:
    error = e

print("Error detectado:",error)
```

```
Error detectado: unsupported operand type(s) for /: 'int' and 'str'
```

Ejercicios de condiciones

Partiendo de las siguientes variables:

- **price** que tiene el precio actual del activo en el mercado
- **buyPrice** que tiene el valor al que compramos el activo al ingresarla a la cartera
- **smaFast y smaSlow** que son dos medias móviles que usaremos para saber si está en tendencia

Ir modificando la variable price para verificar que el script anda

Configurar las variables con los siguientes valores

- smaFast = 100
- smaSlow = 98
- buyPrice = 106

1- Armar un script que me defina en una sola condición la variable **hold** para un activo es verdadera o falsa, en función de las siguientes premisas:

- Que esté en tendencia bull, asumiendo para ello que la media rápida sea un 2% mayor a una media lenta
- Que no rompa un stopLoss del 5% abajo del precio de compra
- Que la ganancia acumulada no sea superior al 20% ya que en ese caso se tomaría ganancias

Y verificar que nos devuelve lo siguiente para los siguientes precios:

- price = 97 => False
- price = 100 => False
- price = 101 => True
- price = 121 => True
- price = 128 => False

2 - Con las mismas premisas del ejercicio anterior, modificarlo para que me devuelva en una variable **state** un string que me indique si sigo invertido en ese activo, pero además en caso de ser falso me indique si se salió por stopLoss o por toma de ganancias

Verificar que nos devuelve lo siguiente para los siguientes precios:

- price = 97 => Salida por stopLoss
- price = 100 => Salida por stopLoss
- price = 101 => Invertidos
- price = 121 => Invertidos
- price = 128 => Salida por takeProfit

3 Reforzar la lógica del ejercicio anterior agregando a la condición del stopLoss que el precio tiene que cortar a la media móvil lento para abajo para disparar el stopLoss. Verificar que nos devuelve lo siguiente para los siguientes precios:

- price = 97 => Salida por stopLoss
- price = 100 => Invertidos
- price = 101 => Invertidos
- price = 121 => Invertidos
- price = 128 => Salida por takeProfit

4 - Reforzar la lógica del ejercicio anterior, haciendo que para gatillar el take profit puede o bien darse el 20% de ganancia o bien un 10% de ganancia y el precio superar en más del 20% a la media móvil rápido Verificar que nos devuelve lo siguiente para los siguientes precios:

- price = 97 => Salida por stopLoss
- price = 100 => Invertidos
- price = 101 => Invertidos
- price = 121 => Salida por takeProfit
- price = 128 => Salida por takeProfit

5 - Hacer un script que arranque definiendo el diccionario:

```
cartera = {"AAPL":30,"AMZN":25,"NFLX":20,"FB":10,"KO":5,"USD":10}
```

Que representa los activos de la cartera y sus porcentajes de peso y que me pida el ticker por un input y me devuelva el % de cada uno y si ingreso en el input un activo inexistente que me devuelva un mensaje aclarando que ese activo no está en la cartera

Tener en cuenta que el script debe funcionar si el usuario ingresa el ticker con alguna o todas sus letras en minúscula

Dada la siguiente lista de activos

```
activos1 = ["ALUA","BMA","BYMA","CEPU","COME","CRES","CVH","EDN","GGAL","MIRG"]  
activos2 = ["PAM","SUPV","TECO2","TGNNO4","TGSU2","TRAN","TXAR","VALO","YPF"]
```

6- Armar un script que genere una lista llamada **activos** con todos los elementos de ambas listas y devolver la cantidad total

7- Pedir un ticker al usuario vía input () y devolver si está o no en la variable activos y que funcione independientemente si se ingresa el ticker en minúscula o mayúscula

8- Dado el siguiente diccionario

```
precios = {"GGAL":80,"YPF":500}
```

Hacer un script que le pida por un input () un ticker al usuario y

- Si el ticker está en el listado de precios le devuelva el precio
- Si no está el precio, pero sí se encuentra en el listado de activos, que le devuelva el mensaje que no tiene el precio de ese ticker (y nombrarlo en el mensaje)
- Si ni siquiera encuentra el ticker en el listado de activos que le avise que el ticker no está ni en el listado de tickers

9- Hacer un script que tome como dato la variable **maxPerdida** que sea un valor porcentual que representa el stopLoss máximo tolerado, y que devuelva en otra variable del tipo string llamada **tipo** las siguientes opciones:

- Stop Largo: Cuando $8\% < \text{maxPerdida} \leq 15\%$
- Stop SemiLargo: Cuando $6\% < \text{maxPerdida} \leq 8\%$
- Stop Standar: Cuando $3\% < \text{maxPerdida} \leq 6\%$
- Stop Corto: Cuando $1\% < \text{maxPerdida} \leq 3\%$
- Fuera de rango: Resto de los casos

Respuestas ejercicios de condiciones

```
#-----#
# Rta Ejercicio 1 #
#-----#

smaFast = 100
smaSlow = 98
buyPrice = 106
price = 128

trend = smaFast > smaSlow * 1.02
stopLoss = price < buyPrice * 0.95
takeProfit = price > 1.2 * buyPrice

hold = trend and not stopLoss and not takeProfit
hold
```

False

```
#-----#
# Rta Ejercicio 2 #
#-----#

smaFast = 100
smaSlow = 98
buyPrice = 106
price = 128

trend = smaFast > smaSlow * 1.02
stopLoss = price < buyPrice * 0.95
takeProfit = price > 1.2 * buyPrice

hold = trend and not stopLoss and not takeProfit

if hold:
    state = "Estamos invertidos aun"
else:
    if stopLoss:
        state = "Salimos por Stop Loss"
    else:
        state = "Salimos por toma de ganancias"

print(state)
```

Salimos por toma de ganancias

```

#-----#
# Rta Ejercicio 3 #
#-----#

smaFast = 100
smaSlow = 98
buyPrice = 106
price = 128

trend = smaFast > smaSlow * 1.02
stopLoss = price < buyPrice * 0.95 and price < smaSlow
takeProfit = price > 1.2 * buyPrice

hold = trend and not stopLoss and not takeProfit

if hold:
    state = "Estamos invertidos aun"
else:
    if stopLoss:
        state = "Salimos por Stop Loss"
    else:
        state = "Salimos por toma de ganancias"

print(state)

```

Salimos por toma de ganancias

```

#-----#
# Rta Ejercicio 4 #
#-----#

smaFast = 100
smaSlow = 98
buyPrice = 106
price = 121

trend = smaFast > smaSlow * 1.02
stopLoss = price < buyPrice * 0.95 and price < smaSlow

takeProfit = price > 1.2 * buyPrice or (price > 1.1 * buyPrice and price > 1.2 * smaFast)

hold = trend and not stopLoss and not takeProfit

if hold:
    state = "Estamos invertidos aun"
else:
    if stopLoss:
        state = "Salimos por Stop Loss"
    else:
        state = "Salimos por toma de ganancias"

print(state)

```

Salimos por toma de ganancias

```

#-----#
# Rta Ejercicio 5 #
#-----#

cartera={"AAPL":30,"AMZN":25,"NFLX":20,"FB":10,"KO":5,"USD":10}
ticker=input("Ingrese el activo a saber su % de ponderacion: ").upper()

try:
    print(f"La ponderación de {ticker} en su cartera es {cartera[ticker]} %")
except:
    print("Ingresaste cualquier verduda man")

```

Ingrese el activo que quiere saber su % de ponderacion: aapl
La ponderación de AAPL en su cartera es 30 %

```

#-----#
# Rta Ejercicio 6 #
#-----#

activos1=["ALUA","BMA","BYMA","CEPU","COME","CRES","CVH","EDN","GGAL","MIRG"]
activos2=["PAM","SUPV","TECO2","TGNN04","TGSU2","TRAN","TXAR","VALO","YPF"]

activos = activos1
activos.extend(activos2)
len(activos)

```

19

```

#-----#
# Rta Ejercicio 7 #
#-----#

activos1=["ALUA","BMA","BYMA","CEPU","COME","CRES","CVH","EDN","GGAL","MIRG"]
activos2=["PAM","SUPV","TECO2","TGNN04","TGSU2","TRAN","TXAR","VALO","YPF"]
activos = activos1

ticker=input("Ingrese un ticker: ").upper()

if ticker in activos:
    print("Si, el ticker está en el listado")
else:
    print("NO, el ticker no se encuentra")

```

Ingrese un ticker: alua
Si, el ticker está en el listado

```

#-----#
# Rta Ejercicio 8 #
#-----#


precios = {"GGAL":80,"YPF":500}
activos1=["ALUA","BMA","BYMA","CEPU","COME","CRES","CVH","EDN","GGAL","MIRG"]
activos2=["PAM","SUPV","TECO2","TGNN04","TGSU2","TRAN","TXAR","VALO","YPF"]
activos = activos1

tickerPrecios = precios.keys()
ticker=input("Ingrese un ticker: ").upper()

if ticker in tickerPrecios:
    print(F"El precio de {ticker} es {precios[ticker]}")
else:
    if ticker in activos:
        print(F"No tengo el precio de {ticker}")
    else:
        print(F"El ticker {ticker} ni siquiera está en mi listado de activos")

```

Ingrese un ticker: alua
No tengo el precio de ALUA

```

#-----#
# Rta Ejercicio 9 #
#-----#


# Alternativa 1

maxPerdida = 16

if maxPerdida > 1 and maxPerdida <=15:

    if maxPerdida > 8 :
        tipo = "Stop Largo"
    elif maxPerdida >6 :
        tipo = "Stop Semilargo"
    elif maxPerdida >3 :
        tipo = "Stop Standar"
    elif maxPerdida >1 :
        tipo = "Stop Corto"

else:
    tipo = "Fuera de rango"

tipo

```

'Fuera de rango'

```
#-----#
# Rta Ejercicio 9 #
#-----#

# Alternativa 2

maxPerdida = 0

if maxPerdida > 8 and maxPerdida <=15:
    tipo = "Stop Largo"

elif maxPerdida >6 and maxPerdida <=15:
    tipo = "Stop SemiLargo"

elif maxPerdida >3 and maxPerdida <=15:
    tipo = "Stop Standar"

elif maxPerdida >1 and maxPerdida <=15:
    tipo = "Stop Corto"

else:
    tipo = "Fuera de rango"

tipo
```

'Fuera de rango'

Archivos necesarios

En esta unidad utilizaremos algunos archivos para trabajar ellos son:

- SPY.csv
- balances.csv
- estado_resultados.csv

Todos ellos pueden descargarse de: www.bit.ly/39Am64T

Instalación Necesaria

Vayan instalando el openpyxl para poder manejar excels se instala simplemente con el siguiente comando desde el “anaconda prompt” o incluso en el mismo jupyter notebook

```
pip install openpyxl
```

Ciclos Finitos e Infinitos

Los distintos tipos de esquemas de ciclos se pueden agrupar en:

- Ciclos Definidos: Sabemos de antemano exactamente cuántas veces se va a ejecutar (por ejemplo 5 veces)
- Ciclos Indefinidos: No sabemos cuántas veces se va a ejecutar, pero sabemos que tiene una cantidad máxima dada por el elemento a iterar
- Ciclos Infinitos
 - Ciclos Infinitos Interactivos: Se ejecuta cada vez que aparece un nuevo feed al ciclo, corta cuando se corta el feed
 - Ciclos Infinitos con centinela: De antemano se ejecuta infinitas veces hasta que aparece señal de corte

Ciclos Definidos

Estos son los más sencillos, básicamente se utiliza el comando FOR para esto. En estos ciclos sabemos de antemano la cantidad de iteraciones que hará

La estructura del comando FOR es muy sencilla, ya que se recorre un elemento iterable (generalmente una lista de elementos) y por cada elemento recorrido se ejecuta el cuerpo del FOR

```
listado = ["GGAL", "PAMP", "YPFD", "CEPU", "EDN", "LOMA", "CRES"]
for ticker in listado:
    print(ticker)
```

GGAL
PAMP
YPFD
CEPU
EDN
LOMA
CRES

Ojo que por ejemplo un string es decir un texto cualquiera es también un objeto iterable, ya que cada letra de la palabra se considera un elemento del string

Esto nos puede ser útil en algún momento para evaluar por separado cada elemento de un ticker de opciones

```
ticker="GFGC140.AB"
for letra in ticker:
    print(letra)
```

G
F
G
C
1
4
0
.A
B

Iterando una lista

Hay varias formas de iterar una lista, la mas sencilla y común es la siguiente:

```
listado = ["AAPL", "AMZN", "NFLX", "FB"]

for elemento in listado:
    print(elemento)
```

AAPL
AMZN
NFLX
FB

Ahora ¿Qué pasa si necesito en cada iteración imprimir el número de elemento dentro de la lista?

En ese caso tenemos 3 formas típicas de resolverlo

- Iterando el índice
- Iterando el índice y el elemento
- Iterando el elemento y generando el índice dentro de la iteración

Veamos cada una en el ejemplo

Iterando el índice

```
listado = ["AAPL", "AMZN", "NFLX", "FB"]

for i in range(len(listado)):
    print(listado[i], ' -- ', i)
```

AAPL -- 0
AMZN -- 1
NFLX -- 2
FB -- 3

Iterando el índice y el elemento

Para esto usamos el `enumerate()` que me genera por cada elemento otro que es su índice, por lo tanto asignamos el índice Y el elemento a la parte desempaquetada del FOR (antes del “in `enumerate()`”)

```
listado = ["AAPL", "AMZN", "NFLX", "FB"]

for i, elemento in enumerate(listado):
    print(elemento, ' -- ', i)
```

AAPL -- 0
AMZN -- 1
NFLX -- 2
FB -- 3

Iterando el elemento y generando el índice dentro de la iteración

Esta sinceramente no es una forma amigable de hacerlo, no es performante tampoco, pero bueno, es una opción mas a tener en cuenta que según las circunstancias, cuando ya el bloque iterador vino armado de esa forma, podemos recordar este ejemplito trivial:

```
listado = ["AAPL", "AMZN", "NFLX", "FB"]

for elemento in listado:
    print(elemento, ' -- ', listado.index(elemento))

AAPL -- 0
AMZN -- 1
NFLX -- 2
FB -- 3
```

Iterando una Tupla

Como vimos las tuplas se comportan igual que las listas, solo que son inutables, pero esto no afecta en nada a las diversas formas de ser iteradas que vimos

Veamos un ejemplo similar al anterior pero con una tupla originalmente:

```
listado = ("AAPL", "AMZN", "NFLX", "FB")

for elemento in listado:
    print(elemento)

AAPL
AMZN
NFLX
FB
```

Tuplas de 1 elemento en Python

Este tema lo pongo por acá porque me han consultado muchos a mi cuenta de twitter por errores relacionados a este tema, resulta que Python tiene un tema de notación o sintaxis con las tuplas de 1 solo elemento medio especial

El punto es el siguiente, cuando definimos una lista de 1 solo elemento podemos poner directamente:

```
listado = ["elemento_unico"]
```

Pero con tuplas esto no genera una tupla en la variable listado, sino un string:

```
listado = ("elemento_unico")
```

Comprobemos esto que les estoy diciendo:

```
listado = ['AAPL']
type(listado)
```

list

```
listado = ('AAPL')
type(listado)
```

str

WTF?

Y si, que le vamos a hacer? Ahora les muestro como solucionarlo si quisieramos una tupla de un solo elemento pero antes reflexionen ustedes mismos por que esta diferencia en las siguientes iteraciones

Lista de 1 elemento

```
listado = ['AAPL']

for elemento in listado:
    print(elemento)
```

AAPL

¿Tupla de 1 elemento?

```
listado = ('AAPL')

for elemento in listado:
    print(elemento)
```

A
A
P
L

Como habrán deducido, Python trata a los strings como una lista de caracteres, con lo cual, el segundo bloque es lógico que haya iterado letra por letra porque como dijimos, ese listado de aparentemente una tupla con el elemento “AAPL” en realidad no es una tupla, es simplemente el string “AAPL”

Ahora, si quisiera entonces una tupla de 1 solo elemento ¿como hago?

Simple, le agrego una coma luego del elemento, y ahí si la variable “listado” en este caso será una tupla:

```
listado = ('AAPL',)

for elemento in listado:
    print(elemento)
```

AAPL

Bueno, como se imaginarán esto empieza a tomar vuelo cuando puedo iterar dentro de una iteración, o cosas así, por ejemplo iterar una lista de listas, y dentro de cada iteración iterar cada sublista, o iterar una lista de diccionarios, y en cada iteración hacer algo con las claves y valores de cada diccionario, etc..

Pero tranca, vamos de a poco

Iterando una lista de tuplas

Vamos primero con un ejemplo sencillo de pares de tickers, si iteramos la variable “pares” que es una lista que adentro tiene tuplas, cada iteración me devolverá la tupla (par) en cada paso

Ahora como cada par es a su vez una colección (en este caso de 2 elementos) si quiero el primero o el segundo tengo que referirme con el slice[] correspondiente, en este caso el primero elemento es par[0] y el segundo par[1]

Veamos esto en el código:

```
pares = [ ("AAPL", "AMZN"), ("NFLX", "FB") ]  
for par in pares:  
    print(f"{par[0]} es par con {par[1]}")
```

AAPL es par con AMZN

NFLX es par con FB

Iterando dentro de una iteración

Como les anticipé, esto empieza a tomar vuelo a medida que puedo tratar con poco código mas y mas cantidad de info, ya veremos cosas mas complejas pero a modo de muestra trivial les dejo este ejemplito en el que iteramos dentro de otra iteración:

```
tuplas = [ ("AAPL", "AMZN", "MSFT", "NFLX", "FB"), ("SPY", "QQQ", "TLT", "GLD", "EWT") ]  
  
for tupla in tuplas:  
    print(f"iterando tupla {tupla}")  
    for elemento in tupla:  
        print(elemento)  
  
iterando tupla ('AAPL', 'AMZN', 'MSFT', 'NFLX', 'FB')  
AAPL  
AMZN  
MSFT  
NFLX  
FB  
iterando tupla ('SPY', 'QQQ', 'TLT', 'GLD', 'EWT')  
SPY  
QQQ  
TLT  
GLD  
EWT
```

Iterando un Diccionario

Como se estarán imaginando hay varias formas de iterar un diccionario:

- Por sus claves
- Por sus valores
- Por sus ítems

Así que vamos uno por uno, son muy similares igual, no hay mucha ciencia

Iterar diccionario por sus claves

En este caso el elemento iterado son las claves, así que si quisiera los valores, debo acudir al diccionario[clave]

```
cartera = {"AAPL":240,"AMZN":1700,"NFLX":298,"FB":146}
```

```
for elemento in cartera:  
    print(f"{elemento} $ {cartera[elemento]}")
```

```
AAPL $ 240  
AMZN $ 1700  
NFLX $ 298  
FB $ 146
```

Como podrán corroborar es lo mismo iterar el diccionario por sus claves, ya que por default la colección iterable de un diccionario son sus claves

```
cartera = {"AAPL":240,"AMZN":1700,"NFLX":298,"FB":146}
```

```
for elemento in cartera.keys():  
    print(f"{elemento} $ {cartera[elemento]}")
```

```
AAPL $ 240  
AMZN $ 1700  
NFLX $ 298  
FB $ 146
```

Iterar diccionario por sus valores

En este caso se complica un poco encontrar las claves adentro de la iteración, ya que dada una clave es fácil encontrar el valor de un diccionario, pero el proceso inverso es un poco más vueltero porque tenemos que recurrir al método index() de las listas, pero como partimos de un diccionario y no de listas, debemos generar previamente las listas de los valores y claves por separado, medio engorroso pero tampoco nada del otro mundo

Veámoslo mejor con un ejemplo:

```

cartera = {"AAPL":240,"AMZN":1700,"NFLX":298,"FB":146}

claves = list(cartera.keys())
valores = list(cartera.values())

for valor in valores:
    elemento = claves[valores.index(valor)]
    print(f"{elemento} $ {valor}")

```

AAPL \$ 240
AMZN \$ 1700
NFLX \$ 298
FB \$ 146

Iterar diccionario por sus ítems

Esta es la forma mas completa ya que accedemos de una a ambas partes de cada elemento del diccionario de movida al iterarlo, veámoslo directo en el ejemplo;

```

cartera = {"AAPL":240,"AMZN":1700,"NFLX":298,"FB":146}

for clave, valor in cartera.items():
    print(f"{clave} $ {valor}")

```

AAPL \$ 240
AMZN \$ 1700
NFLX \$ 298
FB \$ 146

A mi gusto, por la simpleza del código resultante es mucho mejor esta forma

Operadores de asignación

Perdón por lo descolgado de este tema acá, por un lado no sabía bien donde ponerlo así que lo metí acá porque a medida que empezamos a usar ciclos FOR es cuando empezamos por lo general a usar mucho los operadores de asignación especiales o de acumulación.

Veamos, en principio el primer operador de asignación es el simple “igual” es un operador que asigna lo que está a la derecha a la variable que definamos a la izquierda del operador con cierta lógica, en el caso del igual asigna exactamente lo mismo, es decir que la expresión

numero = 7

Asigna a la variable “numero” un valor exactamente igual a 7

Ahora ¿qué pasa si por ejemplo quisiera sumarle 1 a lo que ya tiene la variable “numero”?

```
numero = numero + 1
```

Con esa expresión decimos que sea lo que sea que tenga “numero” le voy a sumar 1 y le asigno ese nuevo valor a “numero”, bueno, esa misma expresión se puede escribir con un “operador de asignación” diferente al “=” que sería el “+=”

Con lo cual la siguiente expresión es equivalente a la anterior, lo que hace es una asignación destructiva, es decir, sobreescribe el valor que tenía la variable “numero” por ese mismo valor incrementado en 1

```
numero += 1
```

Este tipo de asignaciones son especialmente útiles en las iteraciones para ir contando acumuladores y esas cosas, por ejemplo, supongamos que tenemos una lista de precios diarios de GGAL que es la siguiente para el último mes:

```
ggal_precios = [8.54, 8.35, 8.18, 7.94, 8.03, 8.38, 8.12, 7.97, 8.15, 8.09,  
8.26, 8.08, 7.81, 7.79, 8.02, 7.96, 8.1, 8.21, 8.2, 8.55]
```

Y lo que queremos es una serie base 100, es decir que el primer elemento sea 100 y el resto me reflejen el mismo comportamiento de variación porcentual respecto a la base.

```
ggal_precios = [8.54, 8.35, 8.18, 7.94, 8.03, 8.38, 8.12, 7.97, 8.15, 8.09,  
8.26, 8.08, 7.81, 7.79, 8.02, 7.96, 8.1, 8.21, 8.2, 8.55]  
  
ggal_base100 = [round(100* precio/ggal_precios[0],2) for precio in ggal_precios]  
print(ggal_base100)
```

```
[100.0, 97.78, 95.78, 92.97, 94.03, 98.13, 95.08, 93.33, 95.43, 94.73, 96.72, 94.61, 91.45,  
91.22, 93.91, 93.21, 94.85, 96.14, 96.02, 100.12]
```

Como ven una simple lista por comprensión en donde a todos los elementos de los precios originales los multiplico por 100 y los divido por el primero precio de la lista.

Bueno, ahora vamos a llegar a lo mismo, pero le vamos a dar una vuelta de rosca mas que nada para practicar, supongamos que tenemos de nuevo la lista de precios, y queremos generar con ella una lista de variaciones porcentuales día a día, es decir que para cada valor me calcule la variación porcentual respecto al valor anterior (expresado como factor multiplicativo, es decir $1.02 = +2\%$, o bien $0.97 = -3\%$ etc.. es decir calculamos el % de variación/100 sumado a la unidad)

```
ggal_pct_change = [ggal_precios[i]/ggal_precios[i-1] for i in range(1,len(ggal_precios)) ]  
print(ggal_pct_change)
```

```
[0.9778, 0.9796, 0.9707, 1.0113, 1.0436, 0.969, 0.9815, 1.0226, 0.9926, 1.021, 0.9782, 0.9666,  
0.9974, 1.0295, 0.9925, 1.0176, 1.0136, 0.9988, 1.0427]
```

Bien, fíjense que de nuevo, armo una lista por comprensión donde divido a cada elemento (precio) por su valor anterior, nada del otro mundo, usando todos los “i” desde el 1, ya que el “0” no tiene valor anterior

Ahora con estos ggal_pct_change voy a calcular la serie base 100, para ello parto de una lista cuyo primer valor sea 100, y luego hago un acumulado multiplicativo usando un operador de asignación, lo meto en un FOR en lugar de lista por comprensión para que les sea más legible:

```
1 valor = 100
2 ggal_base100 = [valor]
3
4 for pct_change in ggal_pct_change:
5     valor *= pct_change
6     ggal_base100.append(round(valor,2))
7
8 print(ggal_base100)
```

```
[100, 97.78, 95.78, 92.97, 94.03, 98.13, 95.08, 93.33, 95.43, 94.73, 96.72, 94.61, 91.45,
91.22, 93.91, 93.21, 94.85, 96.14, 96.02, 100.12]
```

Como ven, en la línea 5, utilice el operador de asignación “`*=`” para ir expresando el producto acumulado de la base por el factor multiplicativo dado por el `pct_change`.. Y obviamente llego al mismo resultado que por el método anterior, igualmente fue una simple vuelta de tuerca más para mostrar ejemplos y que, por supuesto, sugerirles que lo codeen ustedes, y lo modifiquen y practiquen.

Listado de operadores de asignación

Hay alguno más pero estos son los más útiles, fíjense lo que es Python que uno se va pythonizando hasta en las tareas mas triviales, en este caso en vez de escribir y empezar a copypastear los ejemplo, dejo que lo haga Python con un sencillo ciclo FOR y la función eval() que como ven evalúa una expresión escrita como string

```
operadores = ['+=', '-=', '*=', '/=', '//=', '%=', '**=']

for operador in operadores:
    x = 7
    x = eval(f"x {operador[:-1]} 2")
    print(f"Siendo x=7, x {operador} 2 me devuelve x =", x)
```

```
Siendo x=7, x += 2 me devuelve x = 9
Siendo x=7, x -= 2 me devuelve x = 5
Siendo x=7, x *= 2 me devuelve x = 14
Siendo x=7, x /= 2 me devuelve x = 3.5
Siendo x=7, x //= 2 me devuelve x = 3
Siendo x=7, x %= 2 me devuelve x = 1
Siendo x=7, x **= 2 me devuelve x = 49
```

Ciclos definidos usando la función range

La función **range ()** admite 3 argumentos

- Desde
- Hasta
 - incremento

El orden de los argumentos de la función range es así **range (desde, hasta, incremento)**

Veamos ejemplos

```
for i in range(3):  
    print(i)
```

0
1
2

Veámoslo con un listado predefinido

```
listado = ["AAPL", "AMZN", "NFLX", "FB", "GOOGL", "TSLA"]  
for i in range(3):  
    print(listado[i])
```

AAPL
AMZN
NFLX

- Usemos un rango (desde, hasta)

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

2
3
4

- Usamos un rango (desde, hasta, intervalo)

En este caso vamos a saltar de a 3 elementos, arrancando del 4 hasta llegar al 10 exclusive:

```
for i in range(4,10,3):  
    print(i)
```

4
7

¿Y para que nos puede servir esto?

Supongamos que tenemos un listado de precios cada 1 minuto, pero nos interesa ver en pantalla los cierres cada 3 minutos, en ese caso tendríamos algo similar a esto

```
precios= [112,113,111,114,110,109,110,111,112,115,112]
for i in range(0,len(precios),3):
    print(precios[i])
```

112
114
110
115

Obviamente son ejemplos super simplificados, como para ir haciéndose ideas de usos

Creando listas con ciclos definidos

Como dije desde un comienzo siempre hay varias maneras de escribir el mismo código q en realidad derivan de varias maneras de pensar la solución

Vamos a hacer un script para llenar una lista con los elementos de la tabla de multiplicar de cualquier número "n" dado

En este primer ejemplo defino la tabla como una lista la cual va a estar compuesta por los elementos resultantes del recorrido de un ciclo FOR definido

```
n = 7
tabla = list(i*n for i in range(1,11))
tabla
```

[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]

```
n = 7
tabla = [i*n for i in range(1,11)]
tabla
```

[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]

Sin embargo, acá lo pensamos diferentes, primero definimos la tabla como una lista vacía, y luego creamos un ciclo FOR que lo que hará es ir llenando la tabla elemento por elemento

```

n = 7
tabla = []
for i in range(1,11):
    tabla.append(i*n)
tabla

```

[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]

Ciclos Indefinidos Finitos

Son ciclos en los que a priori no sabemos cuántas veces se van a ejecutar. Para esto tenemos los comandos FOR y WHILE

En el caso del **FOR** el ciclo se ejecuta siempre para todos los elementos luego del **IN** así que vamos a tener que en algún momento indicarle dentro de un **IF** cuando queremos que deje de ejecutar, para ello vamos a usar el comando **BREAK**

```

precios = [112,113,111,114,110,109,110,111,112,115,112]
stopLoss = 110
for precio in precios:
    if(precio > stopLoss):
        print("Hold $" + str(precio))
    else:
        print("--- StopLoss at $" + str(precio) + " ---")
        break

```

Hold \$112
 Hold \$113
 Hold \$111
 Hold \$114
 --- StopLoss at \$110 ---

Si quisieramos replicar el mismo código usando la sentencia **WHILE**, la pregunta que definirá si seguimos ejecutando dentro del ciclo es lo que viene justo después de la sentencia **WHILE**

Es decir que no hace falta poner un IF, sin embargo, si queremos ejecutar alguna línea al salir del ciclo, podemos poner un **ELSE** a su vez como no necesariamente tenemos que iterar un listado como en el FOR, lo que hacemos es usar un índice y un incrementador para ir refiriéndonos a cada elemento de la lista

```
precios = [112,113,111,114,110,109,110,111,112,115,112]
stopLoss = 110
i=0

while precios[i] > stopLoss:
    print("Hold $" + str(precios[i]))
    i += 1

else:
    print("--- StopLoss at $" + str(precios[i]) + " ---")

Hold $112
Hold $113
Hold $111
Hold $114
--- StopLoss at $110 ---
```

Ciclos Infinitos

En estos casos no hay un elemento inicial a iterar como un listado de precios dado o un listado de tickers o cualquier listado predefinido, sino que por ejemplo dependemos de recibir o no un data feed, es decir que el ciclo se ejecutará cada vez que ingresa un nuevo dato al programa, por ahora vamos a hacer el ejemplito con inputs de usuario para no complicarla, más adelante veremos ejemplos reales tomando data en tiempo real

```
stopLoss = 110
dato = float("inf")
while dato > stopLoss:
    dato = float(input("Nuevo precio recibido $"))
    print("Hold $" + str(dato))

else:
    print("--- StopLoss at $" + str(dato) + " ---")
```

```
Nuevo precio recibido      $123
Hold $123.0
Nuevo precio recibido      $111
Hold $111.0
Nuevo precio recibido      $109
Hold $109.0
--- StopLoss at $109.0 ---
```

Corte de ciclos infinitos con corte por TimeOut

Como se darán cuenta la función anterior puede no cortar nunca en la vida, porque si infinitamente recibe datos y esos datos recibidos son siempre mayores al stopLoss la función nunca cortará, si bien esto en nuestro objetivo podría tener sentido, en realidad tenemos que pensar que podría haber eventos inesperados y esto podría tornarse peligroso

Hay miles de maneras de cortar un algo por "cosas raras" una muy típica son cuelgues del sistema de feed de datos, es decir si pasa una determinada cantidad de tiempo que no recibimos datos, nuestro bot puede quedar operando a ciegas con datos viejos pensando que este todo ok y en realidad dejamos de recibir datos porque se colgó algo y el mercado está colapsando.

Una manera muy típica de evitar esto es con los típicos TIMEOUT, que detectan cuando pasa un determinado tiempo de ejecución de una función o script

```
import time
stopLoss = 110
dato = float("inf")
timeout = time.time() + 5
while dato > stopLoss:
    if time.time() > timeout:
        print("Ojota que se colgó todo")
        break
    else:
        timeout = time.time() + 5
        dato = float(input("Nuevo precio recibido $"))
        print("Hold $" + str(dato))
else:
    print("--- StopLoss at $" + str(dato) + " ---")
```

```
Nuevo precio
recibido $123 Hold
$123.0
Ojota que se colgó todo
```

Corte de Ciclos infinitos con Centinela

El centinela es un valor particular (que lo definimos nosotros) o bien puede ser una "lógica de secuencia de valores" que harán que nuestro ciclo corte

Por ejemplo, si lo que esperamos es un precio, podríamos definir que si recibimos el valor 0 se corte inmediatamente el script ya que muy probablemente estemos en presencia de algún tipo de cuelgue

```
stopLoss = 110
dato = float("inf")

while dato > stopLoss:
    dato = float(input("Nuevo precio recibido $"))
    if dato != 0:
        print("Hold $" + str(dato))
    else:
        print("Se corta el programa por recibir dato centinela")
        break
else:
    print("--- StopLoss at $" + str(dato) + " ---")
```

```
Nuevo precio
recibido $123 Hold
$123.0
Nuevo precio recibido $0
Se corta el programa por recibir dato centinela
```

Otra manera de escribir exactamente lo mismo es usando la sentencia continúe, en este caso no queda muy razonable, pero habrá casos en los que el código sea más entendible usando esta sentencia

```
stopLoss = 110
dato = float("inf")
while dato > stopLoss:
    dato = float(input("Nuevo precio recibido $"))
    if dato == 0:
        print("Se corta el programa por recibir dato centinela")
        break
    else:
        continue
    print("Hold $" + str(dato))

else:
    print("--- StopLoss at $" + str(dato) + " ---")
```

```
Nuevo precio
recibido $123 Nuevo
precio recibido $0
Se corta el programa por recibir dato centinela
```

Importación de datos de una planilla

Que es un archivo CSV

Antes que nada, vamos a repasar brevemente que es un archivo CSV

La sigla CSV significa "Valores separados por comas" en inglés obviamente por eso está al revés, bueno, es un estándar,

Ventajas:

- 1- Son estándares, o sea todos los programas son capaces de entenderlos
- 2- Son simplemente archivos de texto con una separación con lo cual son livianos
- 3- Los puedo leer sin tener ningún software pago instalado en mi máquina
- 4- Sirven para guardar tablas de datos (matrices, como en un Excel)
- 5- Es compatible con Excel, con Google sheets, y con todos los gestores de base de datos

Librería CSV

Es una librería estándar de python

Esta librería como se podrán imaginar nos va a permitir abrir y leer (también escribir) archivos CSV Vamos a ir viendo estas funciones directamente de unos ejemplos

En primer lugar, vamos a leer un archivo "SPY.csv" que tiene las cotizaciones de los últimos 20 años del ETF que replica el S&P500

Vemos lo que hace el ejemplo línea por línea

- 1- Importamos la librería CSV que tiene un método para leer justamente este tipo de archivos
- 2- Usamos el método Reader de la librería csv, y pasamos de argumento el archivo y el delimitador
- 3- Creamos una lista vacía
- 4- Armamos un ciclo definido que recorra todas las filas del archivo
- 5- En cada procesamiento del ciclo insertamos la fila a la lista creada
- 6- Imprimimos las primeras 5 filas

Este sería el código:

```
import csv  
data = csv.reader(open('SPY.csv'), delimiter=';')  
lista = []  
for fila in data:  
    lista.append(fila)  
lista[0:5]
```

```
[['timestamp', 'open', 'high', 'low', 'close', 'adjusted_close', 'volume'],  
 ['19/3/2020', '239.25', '247.38', '232.22', '240.51', '240.51', '288124389'],  
 ['18/3/2020', '236.25', '248.37', '228.02', '240', '240', '324845381'],  
 ['17/3/2020', '245.04', '256.17', '237.07', '252.8', '252.8', '260566279'],  
 ['16/3/2020', '241.18', '256.9', '237.36', '239.85', '239.85', '295019288']]
```

En el ejemplo indicamos un CSV con el separador ";" si no se indica nada el estándar es "," en nuestro caso usamos este ejemplo para mostrarlo ya que ambos son muy comunes

Probamos acceder a una sola fila

```
lista[1]  
['19/3/2020', '239.25', '247.38', '232.22', '240.51', '240.51', '288124389']
```

Y también podemos probar acceder a un elemento único de una determinada fila, en este caso al 3º elemento de la 2º fila

Recordemos siempre que en las listas el primer elemento siempre es el 0

```
lista[1][2]  
'247.38'
```

Ahora vamos a ver solo a modo de ejemplo, como imprimiríamos en pantalla la fecha y el precio de cierre de, por ejemplo, los primeros 5 datos de la serie

```
primeros5=lista[1:6]  
for ohlc in primeros5:  
    print(ohlc[0] + " Cierre $" + str(ohlc[4]))
```

```
19/3/2020 Cierre $240.51  
18/3/2020 Cierre $240  
17/3/2020 Cierre $252.8  
16/3/2020 Cierre $239.85  
15/3/2020 Cierre $269.32
```

Guardamos una columna en una lista

Como se habrán dado cuenta esto de tener todas las filas guardadas como filas no nos servirá mucho porque cada fila tiene datos que no tienen nada que ver entre sí, por lo general va a ser más útil agrupar datos de columnas de este tipo de series, veamos cómo puedo hacer eso

Partiendo de la base que tengo en la variable **lista** todas las filas, les recuerdo el código para ello

```
import csv
data = csv.reader(open('SPY.csv'), delimiter=';')
lista = []
for fila in data:
    lista.append(fila)
```

Lo que vamos a hacer es recorrer esas filas e ir llenando una lista con solo los valores de las fechas en este caso

Para ello usamos las siguientes líneas:

- 1- Definimos la lista de fechas vacía
- 2- Luego armamos el ciclo definido
- 3- Dentro del bloque del ciclo agregamos a la lista vacía
- 4- Por último, imprimimos las primeras 5 fechas a ver si este todo ok

Cabe aclarar que vamos a recorrer desde **1** (y no desde 0) hasta **len(lista)**, porque la fila 0 contiene el encabezado y solo queremos los valores

```
fechas = []
for i in range(1,len(lista)):
    fechas.append(lista[i][0])
fechas[0:5]
```

```
['19/3/2020', '18/3/2020', '17/3/2020', '16/3/2020', '13/3/2020']
```

La manera abreviada de llenar la lista es definirla y en el mismo paso, adentro de los corchetes, la lleno Para llenarla, ponemos el valor y luego la iteración que define al valor

```
fechas = [ lista[i][0] for i in range(1,len(lista)) ]
fechas[0:5]
```

```
['19/3/2020', '18/3/2020', '17/3/2020', '16/3/2020', '13/3/2020']
```

Pero se preguntarán por que usamos la lista de filas y no hicimos esto directamente cuando recorrimos el CSV original

No sé si se lo preguntaron, pero si fue así, perfecto, excelente pregunta

La realidad hice esos pasos porque me pareció más pedagógico, pero claramente podríamos directamente cuando leemos el CSV armar la o las columnas que necesitaremos, veamos cómo sería

```
import csv
data = csv.reader(open('SPY.csv'), delimiter=';')
cierres_aj = []
for fila in data:
    cierres_aj.append(fila[5])
cierres_aj[0:4]
```

['adjusted_close', '240.51', '240', '252.8']

Como verán la única diferencia es que en este caso estamos guardando también el encabezado de la columna, lo podríamos quitar luego con la función del ()

```
del(cierres_aj[0])
cierres_aj[0:4]
```

['240.51', '240', '252.8', '239.85']

Como ven la función del (), lo que hizo fue borrar el primer valor (el que tenía el encabezado) pero luego "reseteó" el lugar 0 ya que al pedirle que me devuelva los primeros 4 valores arranca también desde 0 pero ya sin el encabezado

Por último, les muestro aquí una forma más abreviada de obtener una determinada columna con solo 5 líneas en total desde el CSV

```
import csv
data = csv.reader(open('SPY.csv'), delimiter=';')
cierres_aj = [ fila[5] for fila in data ]
del(cierres_aj[0])
cierres_aj[0:4]
```

['240.51', '240', '252.8', '239.85']

Ojo, no es necesario que sepan usar todos los métodos para nada, yo acá trato de mostrarles el mayor abanico posible por cuestiones didácticas.

Usen el que sientan más cómodo de leer y de interpretar, lo más razonable es simplificar la vida y no andar recorriendo gratuitamente una lista de datos que no vamos a necesitar muchas veces, ni de armar variables provisorias como la variable data que habíamos armado llenando la memoria innecesariamente, pero ojo, que también puede que prefieran hacerlo porque crean que a futuro necesitaran esa data completa para obtener otra columna sin tener que volver a recorrer el CSV

Como dijimos al principio de todo con las premisas de la programación, todo dependerá de las necesidades puntuales y también por qué no, de su comodidad, a veces lo prioritario es que el código sea legible o que simplemente sean comandos fáciles de recordar su uso

Generamos una columna nueva a partir de los datos dados

Claramente una de las principales funciones de un script va a ser calcular cosas nuevas a partir de datos dados. Por ejemplo, partiendo de la data de cierres calcular la columna de variaciones diarias

Como la serie de datos está ordenada del más reciente primero hasta el más antiguo al final, el % de variación lo calcularemos como el cociente de: cierre_aj[i] / cierre_aj[i+1]

y luego usaremos el método float () para transformar el dato que me viene encomillado es decir como string a decimal, y también usaremos el método round para limitar la cantidad de decimales del resultado

```
import csv
data = csv.reader(open('SPY.csv'), delimiter=';')
cierres_aj = [ fila[5] for fila in data ]
del(cierres_aj[0])

variaciones = []
for i in range(0,len(cierres_aj)):
    try:
        var = round( (float(cierres_aj[i])/float(cierres_aj[i+1]) -1)*100 , 2 )
        variaciones.append(var)
    except:
        pass

variaciones[0:4]
```

[0.21, -5.06, 5.4, -10.94]

En el ejemplo anterior usamos el método try/except para evitar dos tipos de errores:

- 1- El error de "Out of range" es decir cuando llegue al final de la serie que no me tire error por querer buscar el elemento siguiente (que no existe, obvio)
- 2- El error potencial de alguna división por cero en caso que tenga un 0 en algún cierre (por defecto de la data feed)

De CSV a Diccionario

En realidad, no es muy útil usar diccionarios para calcular parámetros y hacer análisis, el uso de diccionarios es más limitado a definiciones más bien estáticas, pero de todos modos de manera didáctica les dejo ejemplo de cómo armar un diccionario clave valor de fechas y cierres partiendo de un CSV

```
import csv
data = csv.reader(open('SPY.csv'), delimiter=';')
fechas_cierres = []
for fila in data:
    dicc = {}
    dicc[fila[0]] = fila[5]
    fechas_cierres.append(dicc)
fechas_cierres[0:4]

[{'timestamp': 'adjusted_close'},
 {'19/3/2020': '240.51'},
 {'18/3/2020': '240'},
 {'17/3/2020': '252.8'}]
```

Lectura de balances desde un CSV

Obvio que en realidad podemos leer cualquier tipo de dato que tengamos en un Excel o un CSV, en este caso los preparé de ejemplo un archivo con los 10 últimos balances trimestrales y anuales de los 4000 tickers más líquidos en USA que cotizan hace más de dos años.

Veamos primero que me trae en la primera fila

```
import csv
data = csv.reader(open('balances.csv'), delimiter='; ')
filas = [ fila for fila in data ]
filas[0]

['symbol', 'tipo', 'date', 'intangibleAssets', 'totalLiab', 'totalStockholderEquity',
'otherCurrentLiab', 'totalAssets', 'commonStock', 'otherCurrentAssets',
'retainedEarnings', 'otherLiab', 'otherAssets', 'totalCurrentLiabilities',
'otherStockholderEquity', 'propertyPlantEquipment', 'totalCurrentAssets',
'longTermInvestments', 'netTangibleAssets', 'shortTermInvestments', 'netReceivables',
'longTermDebt', 'inventory', 'accountsPayable']
```

Como verán nos devuelve los encabezados de las columnas, veamos ahora un ejemplo para traer solo los blancos de FB

```
import csv
data = csv.reader(open('balances.csv'), delimiter='; ')
balances = list()
for fila in data:
    if fila[0]=="FB":
        balances.append(fila)
print("La cantidad de balances es",len(balances),"El primero es", balances[0])
```

La cantidad de balances es 20

El primero es ['FB', 'trimestral', '31/12/2019', '894000000', '3232200000 0',
'101054000000', '10854000000', '133376000000', 'NULL', '8000000', '556
92000000', '7327000000', '2759000000', '15053000000', '-489000000', '44783
000000', '66225000000', 'NULL', '81445000000', '35776000000', '951800000 0',
'NULL', 'NULL', '1363000000']

Vamos ahora a mostrar solo el activo total (totalAssets) de los balances anuales (expresado en millones)

Para ello, lo que vamos a hacer es primero definir un diccionario vacío, y luego lo vamos a ir llenando (recuerden, clave=>valor) con las fechas como clave y la columna totalAssets como valor (la 8va columna, es decir la de índice 7 contando desde el 0)

```
activos_anuales = {}
for balance in balances:
    if balance[1]=="anual":
        activos_anuales[balance[2]] = round(int(balance[7]) / 1000000)
activos_anuales

{
'31/12/2019': 133376,
'31/12/2018': 97334,
'31/12/2017': 84524,
'31/12/2016': 64961,
'31/12/2015': 49407,
'31/12/2014': 40184,
'31/12/2013': 17895,
'31/12/2012': 15103,
'31/12/2011': 6331,
'31/12/2010': 2990
}
```

Calculemos ahora para FB las ratios anuales de Activo Corriente sobre Pasivo Corriente

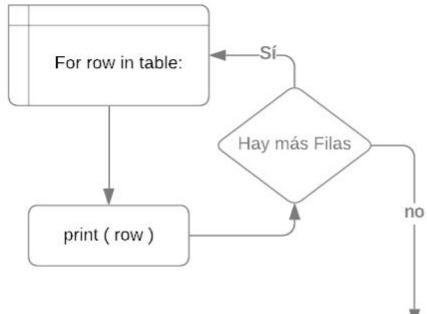
Ídem al procedimiento anterior solo que en lugar de guardar en el valor un dato, guardamos el cociente entre dos datos de la misma columna (el mismo balance)

```
ratio = {}
for balance in balances:
    if balance[1]=="anual":
        try:
            ratio[balance[2]] = round(int(balance[16]) / int(balance[13]),2)
        except:
            ratio[balance[2]] = "No se pudo calcular"
ratio

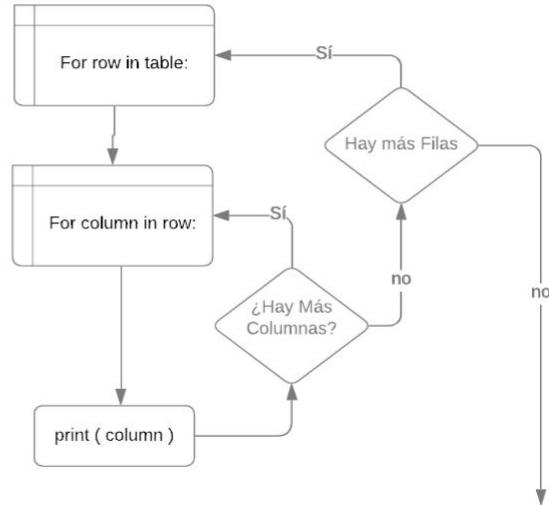
{
'31/12/2019': 4.4,
'31/12/2018': 7.19,
'31/12/2017': 12.92,
'31/12/2016': 11.97,
'31/12/2015': 11.25,
'31/12/2014': 9.4,
'31/12/2013': 11.88,
'31/12/2012': 10.71,
'31/12/2011': 'No se pudo calcular',
'31/12/2010': 'No se pudo calcular'
}
```

Bucles Anidados

Flujo del Ciclo FOR simple



Flujo del dos ciclos FOR anidados



El ejemplo básico que vemos en el diagrama de flujo de dos ciclos anidados es cuando queremos recorrer una tabla (filas y columnas) e imprimir por separado cada valor de la tabla

Para ello entramos a un primer ciclo FOR y recorremos las filas, y adentro del bloque del ciclo FOR entramos a otro ciclo FOR que recorra las columnas

```
table = [["f1c1","f1c2","f1c3"],["f2c1","f2c2","f2c3"]]
for row in table:
    for column in row:
        print(column)
```

f1c1
f1c2
f1c3
f2c1
f2c2
f2c3

Ejemplos prácticos con bucles anidados

Bueno, ahora es donde se empieza a poner interesante el poder del scripting, si bien después vamos a ver librerías que nos facilitarán este tipo de recorrido de tablas, no siempre convendrá usarlas o no siempre las tendremos a disposición, por lo que siempre es conveniente saber hacerlo sin la ayuda de esas "librerías mágicas"

Tomando el archivo con los balances de las empresas, hagamos un diccionario con la ratio Activo Corriente / Pasivo Corriente, pero solo del último balance disponible (teniendo en cuenta que vienen ordenados de más reciente a más antiguo siempre) y de las siguientes empresas como para comparar la foto actual entre ellas

```
empresas = ["AAPL", "AMZN", "FB", "TSLA", "KO", "NFLX"]
```

Bueno, para esto tenemos que meter dos bucles anidados, en este caso vamos a recorrer primero la lista de empresas, y para cada empresa recorreremos las filas del CSV (podría hacerse al revés también)

Cuando recorramos las filas del CSV vamos a buscar para cada empresa el último balance y le calcularemos la ratio y lo pondremos en un diccionario clave=>valor donde la clave sea el ticker y el valor la ratio

```
import csv
data = csv.reader(open('balances.csv'), delimiter=';')
balances = [ fila for fila in data ]

empresas = ["AAPL", "AMZN", "FB", "TSLA", "KO", "NFLX"]
screener = {}
for empresa in empresas:
    for balance in balances:
        if balance[0]==empresa and balance[1]=="anual":
            try:
                screener[empresa] = round(int(balance[16]) / int(balance[13]),2)
            except:
                screener[empresa] = "No se pudo calcular"
            break
print(screener)
```

```
{
'AAPL': 1.54,
'AMZN': 1.1,
'FB': 4.4,
'TSLA': 1.13,
'KO': 0.76,
'NFLX':0.9
}
```

Bien, veamos un poco esto más en detalle, para cada empresa recorro todos los balances listados, después viene una pregunta con un IF y un AND, veamos

LINEA 9, el IF: lo que busco acá en cada fila es ver si la fila coincide con la empresa que estoy en ese punto del FOR, para ello pregunto por balance [0] es decir el primer valor de la fila balance, y justamente ese primer valor es el ticker o symbol como figura en el CSV. Bien, pero también pregunto si la siguiente columna de ese balance (balance [1]) es igual a "anual" ya que queremos ver la ratio solo en los balances anuales

Línea 10 el bloque Try/Except: Acá lo que hago, es calcular el ratio siempre que se pueda calcular y devolver un texto de que no se pudo llegar al caso pero que es lo de la línea 14?

Línea 14, break: Esta instrucción lo que hace es salirse del ciclo de menor jerarquía es decir en este caso el de los balances y no el de las empresas, y esto es necesario porque si no pongo esta instrucción va a seguir calculando el ratio con los balances subsiguientes y va a sobrescribir el valor, y recordemos que el ejercicio pedía calcular el ratio del último balance y nos decían que vienen ordenados del más reciente primero al más antiguo al final, así que solo lo tengo que calcular en la primera coincidencia y salirme de ese ciclo

¿Alguien pensó en la eficiencia de esa forma de resolver el ejercicio?

Si alguien se dio cuenta que eso que hicimos era sencillo de entender, pero poco eficiente a nivel esfuerzo computacional muy bien observado el punto

El razonamiento de la solución anterior es decir lo siguiente:

- 1- Agarro la primera empresa que quiero
- 2- Recorro toda la lista de balances hasta encontrar ESA empresa
- 3- Hago lo que tengo que hacer, calcular etc.
- 4- Sigo agarrando, la siguiente empresa de la lista y repito el punto 2 y 3

Ya de por sí es obvio que algo acá no es necesario, por sentido común, nosotros no pensamos así, diríamos vamos recorriendo los 63k balances y donde encontramos una empresa de la lista hacemos lo que tenemos que hacer y seguimos adelante, de esta forma recorreremos una sola vez los 63k balances y no una vez por cada empresa que necesite

El punto es que generalmente los ciclos anidados suelen ser muy poco eficientes, por lo general suele haber formas alternativas de resolver el tema sin necesidad de hacer ciclos anidados, de hecho, este es uno de esos casos, por lo general también no nos importa mucho la eficiencia del algoritmo en cuanto a su esfuerzo computacional, pensemos un poco este ejemplo

Mi CSV tenía 63k filas, y mi listado de empresas era de 6, es decir que la solución de los ciclos anidados hacia $63k * 6 = 378k$ ciclos, porque para cada empresa volvía a recorrer otra vez los 63k balances. Si nuestro screener tenía 1000 empresas esa forma de resolverlo requeriría 63 millones de ciclos cuando se podría resolver en 63mil, es decir un algo 1000 veces más eficiente en "esfuerzo computacional"

Dicho todo eso entonces, resolvamos esto de una forma más sensata desde el punto de vista eficiencia

Para ello primero armo un diccionario con el nombre "calculado" en el que para cada empresa le asigno el booleano False

Este diccionario me va a servir para que una vez que empiezo a recorrer los balances si encuentro uno de una empresa listada, hago lo que tengo que hacer y le cambio ese booleano por True para indicar que esa empresa ya hice lo necesario y no me sobrescriba la ratio con los balances más viejos

```
import csv
data = csv.reader(open('balances.csv'), delimiter='; ')
balances = [ fila for fila in data ]

empresas = ["AAPL", "AMZN", "FB", "TSLA", "KO", "NFLX"]
calculado = {}
for empresa in empresas:
    calculado[empresa] = False

screener = {}
for balance in balances:
    empresa = balance[0]
    if empresa in empresas and balance[1]=="anual" and calculado[empresa] == False:
        try:
            screener[empresa] = round(int(balance[16]) / int(balance[13]),2)
        except:
            screener[empresa] = "No se pudo calcular"
    finally:
        calculado[empresa] = True
print(screener)
```

```
{
'AAPL': 1.54,
'AMZN': 1.1,
'FB': 4.4,
'KO': 0.76,
'NFLX': 0.9,
'TSLA': 1.13
}
```

Archivos necesarios para los ejercicios

- balances.csv
- estado_resultados.csv

Se pueden descargar de: <http://bit.ly/39Am64T> (<http://bit.ly/39Am64T>)

Ejercicios con CSVs

Dado el archivo SPY.csv que contiene en un CSV las cotizaciones de SPY hasta la fecha que fue generado:

1 - Generar un script que me devuelva la cantidad de subas diarias mayores al +5%, basados en los cierres ajustados

2 - Generar un script que arme un listado con los gaps de apertura (variación % del precio de apertura respecto al cierre anterior)

Imprimir en pantalla los primeros 5 para verificar que estén correctos

3 - Basados en la lista de gaps anterior. Contabilizar del listado de gaps anterior la cantidad de gaps al alza, cantidad gaps a la baja y los neutros

4 - Contrastar los resultados de gaps al alza y gaps a la baja con la cantidad de ruedas que el intradiario (cierre sobre apertura) fue positivo vs los que fue negativo

5 - En función de las listas construidas en los ejercicios anteriores, armar un diccionario con los siguientes valores:

- El promedio de los gaps positivos y el promedio de los negativos
- El % de Gaps Positivos y Negativos sobre el total de Gaps
- El promedio de los intradiario positivos y el promedio de los negativos
- El % de Intras Positivos y Negativos sobre el total de Intras

6 - Tomando la lista de variaciones entre cierres ajustados, armar un diccionario con los siguientes valores:

- El promedio de las variaciones positivas y el promedio de las negativas
- El % de variaciones Positivas, Negativas y neutras sobre el total
- Inferir si el mayor movimiento del índice se da entre cierres, durante las ruedas o mientras el mercado está cerrado

Dado el archivo balances.csv y el archivo estado_resultados.csv que contiene los últimos 10 estados de resultado anuales y trimestrales de las 4000 acciones más líquidas de USA:

7- Mostrar las columnas de la planilla

8- Mostrar los 4 últimos EBIT de AAPL trimestrales en millones de USD

9- Mostrar el último ratio costos/ingresos anuales de FB, AMZN, AAPL, NFLX y GOOGL

10- Mostrar el listado de empresas cuya ratio costos/ingresos anuales esté por debajo de 0.5, y que hayan invertido +5 B en R+D en el último ejercicio anual

11- Mostrar el promedio de inversión en R+D en Billones americanos de AAPL en los últimos 10 trimestres

12- Contar la cantidad de empresas que tuvieron el último balance anual una inversión en I+D de más de 1B americano

13- Calcular el valor promedio (en millones de dólares) que las empresas invirtieron en I+D el último balance anual

14- Elegir 5 empresas al azar de las que hayan invertido más de 400 millones de dólares

Respuestas ejercicios con CSVs

```
#----- #
# Rta Ejercicio 1 #
#----- #

import csv

data = csv.reader(open('SPY.csv'), delimiter=';')

lista = []

for fila in data:
    lista.append(fila)

cierres_aj = [ lista[i][5] for i in
range(1,len(lista)) ] variaciones = []

for i in range(0,len(cierres_aj)):

    try:
        var = round( (float(cierres_aj[i])/float(cierres_aj[i+1]) -1)*100, 2 )
    except:
        var = 0

    if variacion > 5:
        variaciones.append(var)

len(variaciones)
```

13

```
#----- #
# Rta Ejercicio 2 #
#----- #

cierres = [ lista[i][4] for i in range(1,len(lista)) ]
aperturas = [ lista[i][1] for i in range(1,len(lista)) ]
gaps = []
for i in range(0,len(aperturas)):

    try:
        gap = round( (float(aperturas[i])/float(cierres[i+1]) -1)*100, 2 )
    except:
        gap = 0

    gaps.append(gap)

gaps[0:5]
```

[-0.31, -6.55, 2.16, -10.45, 6.04]

```

#-----#
# Rta Ejercicio 3 #
#----- #

gaps_pos, gaps_neg, gaps_neutros = ([] for i in range(3))
for gap in gaps:
    if gap > 0 :
        gaps_pos.append(gap)
    elif gap < 0:
        gaps_neg.append(gap)
    else:
        gaps_neutros.append(gap)

print ( "Gaps Pos:", len(gaps_pos), " Gaps Neg:", len(gaps_neg), " Gaps Neutros:", len(gaps_neutros) )

```

Gaps Pos: 2672 Gaps Neg: 2270 Gaps Neutros: 90

```

#-----#
# Rta Ejercicio 4 #
# ----- #

intras, intras_pos, intras_neg, intras_neutros = ([] for i in range(4))

for i in range(0,len(aperturas)):
    intra = round( (float(cierres[i])/float(aperturas[i]) -1)*100, 2 )
    intras.append(intra)
    if intra > 0 :
        intras_pos.append(intra)
    elif intra < 0:
        intras_neg.append(intra)
    else:
        intras_neutros.append(intra)

print ( "Intras Pos:", len(intras_pos), "Intras Neg:", len(intras_neg),
        "Intras Neutros:", len(intras_neutros) )

```

Intras Pos: 2640 Intras Neg: 2340 Intras Neutros: 52

```

#-----#
# Rta Ejercicio 5 #
# -----#

d = {}

d.update ( {"GapsPos Media" : round(sum(gaps_pos)/len(gaps_pos),2) } )
d.update ( {"GapsNeg Media" : round(sum(gaps_neg)/len(gaps_neg),2) } )
d.update ( {"GapsPos %" : round( 100 * len(gaps_pos)/len(gaps) ,2 ) } )
d.update ( {"GapsNeg %" : round( 100 * len(gaps_neg)/len(gaps) ,2 ) } )
d.update ( {"GapsNeutros %" : round( 100 * len(gaps_neutros)/len(gaps) ,2 ) } )
d.update ( {"IntraPos Media" : round(sum(intras_pos)/len(intras_pos),2) } )
d.update ( {"IntraNeg Media" : round(sum(intras_neg)/len(intras_neg),2) } )
d.update ( {"IntrasPos %" : round( 100 * len(intras_pos)/len(intras) ,2 ) } )
d.update ( {"IntrasNeg %" : round( 100 * len(intras_neg)/len(intras) ,2 ) } )
d.update ( {"IntrasNeutros %" : round( 100 * len(intras_neutros)/len(intras) ,2 ) } )

d

```

```

{'GapsPos Media': 0.42,
'GapsNeg Media': -0.46,
'GapsPos %': 53.1,
'GapsNeg %': 45.11,
'GapsNeutros %': 1.79,
'IntraPos Media': 0.63,
'IntraNeg Media': -0.71,
'IntrasPos %': 52.46,
'IntrasNeg %': 46.5,
'IntrasNeutros %': 1.03}

```

```

#----- #
# Rta Ejercicio 6 #
#----- #

vars_pos, vars_neg, vars_neutros = ([] for i in range(3))
for var in variaciones:
    if var > 0 :
        vars_pos.append(var)
    elif var < 0:
        vars_neg.append(var)
    else:
        vars_neutros.append(var)

dv = {}
dv.update({ "VarsPos Media" : round(sum(vars_pos)/len(vars_pos),2) })
dv.update({ "VarsNeg Media" : round(sum(vars_neg)/len(vars_neg),2) })
dv.update({ "VarsPos %" : round( 100 * len(vars_pos)/len(variaciones),2 ) })
dv.update({ "VarsNeg %" : round( 100 * len(vars_neg)/len(variaciones),2 ) })
dv.update({ "VarsNeutros %" : round( 100 * len(vars_neutros)/len(variaciones),2 ) })
dv

```

```

{'VarsPos Media': 0.76,
'VarsNeg Media': -0.86,
'VarsPos %': 54.01,
'VarsNeg %': 45.12,
'VarsNeutros %': 0.87}

```

```

#-----#
# Rta Ejercicio 7 #
#----- #

import csv
data = csv.reader(open('estado_resultados.csv'), delimiter=';')
estados = [ fila for fila in data ]
estados[0]

['symbol',
'tipo',
'date',
'researchDevelopment',
'effectOfAccountingCharges',
'incomeBeforeTax',
'minorityInterest',
'netIncome',
'sellingGeneralAdministrative',
'grossProfit',
'ebit',
'nonOperatingIncomeNetOther',
'operatingIncome',
'otherOperatingExpenses',
'interestExpense',
'extraordinaryItems',
'nonRecurring',
'otherItems',
'incomeTaxExpense',
'totalRevenue',
'totalOperatingExpenses',
'costOfRevenue',
'totalOtherIncomeExpenseNet',
'discontinuedOperations',
'netIncomeFromContinuingOps',
'netIncomeApplicableToCommonShares',
'preferredStockAndOtherAdjustments']

```

```

#-----#
# Rta Ejercicio 8 #
#----- #

import csv
data = csv.reader(open('estado_resultados.csv'), delimiter=';')
estados = [ fila for fila in data ]
q = 0
respuesta = []
for estado in estados:
    if estado[0]=="AAPL" and estado[1]=="trimestral" and q < 4 :
        respuesta.append(round(int(estado[10])/1000000))
        q = q+1
respuesta

```

[25569, 15625, 11544, 13415]

```

#----- #
# Rta Ejercicio 9 #
#----- #

import csv
data =
csv.reader(open('estado_resultados.csv'), delimiter=';')
estados = [ fila for fila in data ]
empresas = [ "FB", "AMZN", "AAPL", "NFLX" , "GOOGL"]

calculado = {}
for empresa in empresas:
    calculado[empresa] = False

screener = {}

for estado in estados:
    empresa = estado[0]
    if empresa in empresas and estado[1]=="anual" and calculado[empresa] == False:
        try:
            screener[empresa] = round(int(estado[21])/int(estado[19]),2)
        except:
            screener[empresa] = "No se puede calcular"
        finally:
            calculado[empresa] = True
screener

{
'AAPL': 0.62,
'AMZN': 0.59,
'FB': 0.18,
'GOOGL': 0.44,
'NFLX': 0.62
}

```

```

#-----#
# Rta Ejercicio 10 #
#----- #

import csv
data = csv.reader(open('estado_resultados.csv'), delimiter=';')
estados = [ fila for fila in data ]
empresas = [estado[0] for estado in estados]

calculado = {}
for empresa in empresas:
    calculado[empresa] = False

screener = {}
for estado in estados:
    empresa = estado[0]

    if estado[1]=="anual" and calculado[empresa] == False :

        try:
            ratio = round(int(estado[21])/int(estado[19]),2)
            rd_bln = int(estado[3]) / 1000000000
            if ratio < 0.5 and rd_bln > 5:
                screener[empresa] = ratio

        except:
            ratio = "No se puede calcular"

    finally:
        calculado[empresa] == True

screener

```

```

{'ABBV': 0.24,
'AZN': 0.15,
'BMY': 0.28,
'BTI': 0.24,
'CELG': 0.08,
'CSCO': 0.33,
'DEO': 0.42,
'F': 0.4,
'FB': 0.14,
'GILD': 0.21,
'GOOG': 0.38,
'GOOGL': 0.3,
'GSK': 0.2,
'INTC': 0.35,
'JNJ': 0.31,
'LLY': 0.16,
'MRK': 0.4,
'MSFT': 0.2,
'NVS': 0.23,
'ORCL': 0.11,
'PFE': 0.24,
'QCOM': 0.4,
'SNY': 0.24,
'TEF': 0.48}

```

```

#-----#
# Rta Ejercicio 11 #
#----- #

import csv
data = csv.reader(open('estado_resultados.csv'), delimiter=';')
estados = [ fila for fila in data ]
rd_lista = []

for estado in estados:
    empresa = estado[0]
    if estado[1]=="trimestral" and empresa == "AAPL" :
        try:
            rd_bln = int(estado[3]) / 1000000000
            rd_lista.append(rd_bln)
        except:
            pass

rd_promedio = sum(rd_lista)/len(rd_lista)
rd_promedio

```

3.7901

```

#-----#
# Rta Ejercicio 12 #
#----- #

import csv
data = csv.reader(open('estado_resultados.csv'), delimiter=';')
estados = [ fila for fila in data ]
empresas = [estado[0] for estado in estados]
calculado = {}

for empresa in empresas:
    calculado[empresa] = False

listado = []
for estado in estados:
    empresa = estado[0]
    if estado[1]=="anual" and calculado[empresa] == False:

        try:
            rd_bln = int(estado[3]) / 1000000000
            calculado[empresa] = True

            if rd_bln > 1:
                listado.append(rd_bln)

        except:
            pass

len(listado)

```

```

#-----#
# Rta Ejercicio 13 #
#----- #

import csv
data = csv.reader(open('estado_resultados.csv'), delimiter=';')
estados = [ fila for fila in data ]
empresas = [estado[0] for estado in estados]
listado = []
calculado = {}

for empresa in empresas:
    calculado[empresa] = False

for estado in estados:
    empresa = estado[0]
    if estado[1]=="anual" and calculado[empresa] == False:
        try:
            rd_bln = int(estado[3]) / 1000000
            calculado[empresa] = True
            listado.append(rd_bln)
        except:
            pass

sum(listado)/len(listado)

```

181.6195679740683

```

#-----#
# Rta Ejercicio 14 #
#----- #

import csv, random

data = csv.reader(open('estado_resultados.csv'), delimiter=';')
estados = [ fila for fila in data ]
empresas = [estado[0] for estado in estados]
calculado = {}

for empresa in empresas:
    calculado[empresa] = False

listado = []
for estado in estados:
    empresa = estado[0]
    if estado[1]=="anual" and calculado[empresa] == False:
        try:
            rd_bln = int(estado[3]) / 1000000
            calculado[empresa] = True
            if rd_bln > 400:
                listado.append(empresa)
        except:
            pass
random.sample(listado,10)

```

['MLNX', 'BIDU', 'ON', 'UTX', 'CRM', 'TEVA', 'TMO', 'SAP', 'BMY', 'NFLX']

Ejercicios de cálculo con ciclos

La siguiente es una lista con los últimos 50 precios EOD de AAPL

(últimos 50 del día que bajé esos datos 😊)

```
precios_AAPL = [131.01, 126.6, 130.92, 132.05, 128.98, 128.8, 130.89, 128.91, 127.14, 127.83,  
132.03, 136.87, 139.07, 142.92, 143.16, 142.06, 137.09, 131.96, 134.14, 134.99, 133.94, 137.39,  
136.76, 136.91, 136.01, 135.39, 135.13, 135.37, 133.19, 130.84, 129.71, 129.87, 126.0, 125.86,  
125.35, 120.99, 121.26, 127.79, 125.12, 122.06, 120.13, 121.42, 116.36, 121.09, 119.98, 121.96,  
121.03, 123.99, 125.57, 124.76]
```

1- Dada la lista de precios de AAPL, armar una lista nativa mediante un FOR que llene los retornos porcentuales diarios de AAPL, e imprimir los últimos 3 valores al final

2- Calcular el desvío estándar de la distribución a mano

3- Asumiendo una distribución normal con el mismo mu y sigma de esos últimos 50 retornos diarios calculados en los ejercicios previos simular unos 1000 retornos aleatorios y guardarlos en una variable llamada simulación, usando listas nativas y la librería random.

Hacer la simulación fijando la semilla numero 211

Imprimir los primeros 10 valores de la simulación con el formato XX.XX %

4- Armar una lista con la media móvil de 5 días, con los últimos 45 de los 50 días dados en el dato

Imprimir los últimos 3 valores de media móvil

Para finalizar el libro permítanme un único ejercicio fuera del área de finanzas, pero que me parece muy piola por lo sencillo y útil para aprender a pensar en forma lógica, la idea del ejercicio es que puedan pensar las instrucciones para que Python haga muchos términos repetitivos de una serie o productorio, parece muy sencillo entender la lógica de como calcular PI en cada caso, de hecho no pongo la fórmula simbólica para que no sea chocante porque nuestro cerebro es perfectamente capaz de armar esa fórmula general es decir de generalizar el problema dada una cantidad de términos razonable de la serie

Lo loco es que cuando queremos traspasar esa lógica a instrucciones, la cosa no parece tan fácil, a decir verdad si lo es, es un problema muy fácil, pero por ser de los primeros problemas de este tipo algorítmicos con que se topan, es normal que lo sientan muy difícil, lo bueno es que a dos páginas de distancia tienen las soluciones, con lo cual no se desesperen y disfruten de pensar y probar todo lo que se les ocurra hasta que salga, es mi consejo, así que sin más, al problema:

5 - Dadas las siguientes fórmulas para la aproximación de pi, armar un script de Python que haga los ciclos FOR en cada caso con 10.000 términos de cada fórmula

5.1 Euler

$$\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \dots$$

5.2 Leibniz

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

5.3 Wallis

$$\frac{\pi}{2} = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdots$$

5.4 Viete

$$\pi = 2 \cdot \frac{2}{\sqrt{2}} \cdot \frac{2}{\sqrt{2 + \sqrt{2}}} \cdot \frac{2}{\sqrt{2 + \sqrt{2 + \sqrt{2}}}} \cdots$$

PD: Si piensan que es difícil resolverlo, imaginen los pobres diablos que calcularon esos 10.000 términos con lapiz y papel (posta que hubo mucha gente loca haciendo ese tipo de cosas)

Respuestas a ejercicios de cálculo con ciclos

```
#=====
#      Respuesta Ejercicio 1      #
#=====

px_i = precios_AAPL[:-1]
px_f = precios_AAPL[1:]

retornos = []
for i in range(len(px_i)):
    retornos.append(px_f[i]/px_i[i]-1)

retornos[-3:]
```

```
[0.02445674626125749, 0.012742963142188923, -0.0064505853308910455]
```

```
#=====
#      Respuesta Ejercicio 2      #
#=====

promedio = sum(retornos)/len(retornos)

dif = 0
for value in retornos:
    dif += (value - promedio)**2

varianza = dif/len(retornos)
desvio = varianza**0.5

promedio, desvio
```

```
(-0.000767824425837304, 0.021444236218944465)
```

```
#=====
#      Respuesta Ejercicio 3      #
#=====

import random
random.seed(211)

simulacion = []
for i in range(1000):
    simulacion.append(random.normalvariate(mu=promedio, sigma=desvio))

for s in simulacion[-10:]:
    print(f"{s:.2%}", end='   ')
```

```
-0.57%  1.02%  2.78%  -0.44%  2.84%  -5.42%  1.60%  0.17%  3.33%  -4.69%
```

```
#=====
#      Respuesta Ejercicio 4      #
=====

medias_moviles = []
for i in range(4, len(precios_AAPL)):
    precios = precios_AAPL[i-4:i+1]
    promedio = sum(precios)/5
    medias_moviles.append(round(promedio,2))

medias_moviles[-3:]
```

[121.61, 122.51, 123.46]

```
#=====
#      Ejercicio 5.1      #
=====

# Euler

iteraciones=10**5
pi = 0

for i in range(1,iteraciones):
    pi += 1/(i**2)

pi = (pi*6)**0.5
pi
```

3.141583104230963

```
#=====
#      Ejercicio 5.2      #
=====

# Leibniz

iteraciones=10**5
pi = 0
for i in range(iteraciones):
    pi += ((-1)**i) / (2*i+1)

pi *= 4
pi
```

3.1415826535897198

```

#=====
#      Ejercicio 5.3      #
#=====

# Wallis

iteraciones=10**5
pi = 1
for i in range(1,iteraciones):
    n1, n2 = 2*i / (2*i-1), 2*i/(2*i+1)
    pi *= n1*n2

pi *=2
p

```

3.1415847995784807

```

#=====
#      Ejercicio 5.4      #
#=====

# Viette

iteraciones=10**5

pi = 2
d = 0
for i in range(iteraciones):
    d = (2 + d) ** 0.5
    pi *= (2/ d)

pi

```

3.1415926535897944

Palabras Finales

Este tomo en su formato ebook, a partir de Septiembre de 2021 ya es de distribución gratuita, y mi plan es a medida que vaya haciendo las correcciones y sacando los tomos nuevos que faltan, ir liberando de a uno todos los tomos de esta colección para que queden de dominio público.

Espero que este libro los haya motivado a seguir aprendiendo, o que al menos hayan disfrutado del camino si es que aún no están convencidos de seguir aprendiendo estas cosas.

Si llegados al final de este libro creen que la programación no es lo suyo porque los aburre o simplemente no les parece algo interesante o no los atrae, está perfecto.

Ahora si creen que no es lo suyo porque es muuuuy difícil, por favor les pido que para lo primero se den otra oportunidad, así como en las finanzas, acá hay que manejar mucho la parte psicológica al tratar con lenguajes de programación, sé que es horriblemente frustrante cuando no te sale nada y ni siquiera podes putear a nadie porque del otro lado hay una estúpida computadora jaja, pero tranca, es normal, es parte del oficio, como cuando tenés esos días en el mercado o el trabajo que no te sale absolutamente nada, son los menos, a la larga se superan y son solo recuerdos y anécdotas hasta graciosas si uno sigue en carrera.

Si llegados acá creen que la programación en finanzas no es lo suficientemente útil, tengan paciencia, esto es recién el primer paso tomo 0 de 15, la cosa se pone infinitamente mas interesante luego, se los aseguro, pero como todo, por algo hay que empezar y esta primera parte es medio un embole.

Si se traban mucho en algún lado me pueden escribir mensajes directos a mi twitter @johnGalt_is_www No es ninguna molestia que me escriban sus dudas, de hecho, messirve mucho que me consulten porque me da un feedback muy copado para hacerle correcciones a esta obra y también aprendo mucho de la interacción, esto es un aprendizaje continuo, las cosas cambian todos los días y nada mejor que estar actualizado siempre intercambiando con gente en los mismos intereses.

Para cerrar, como siempre digo, la planificación financiera es una maratón y no una carrera de 100 metros, y lo más importante son los últimos kilómetros y no los primeros. Así que a seguir disfrutando del camino que esto recién empieza..