1. Introducción

Python es un lenguaje interpretado, multiparadigma. Soporta programación orientación a objetos (oop), programación imperativa y funcional. Es de tipado dinámico, multiplataforma y multipropósito.

Interpretado:

Los lenguajes de programación se suelen agrupar en interpretados y compilados según la forma en la que son traducidos. Los lenguajes se inventaron para facilitar al programador el desarrollo aplicaciones. Por eso cuando nosotros escribimos un código en realidad lo que estamos haciendo es hablar un lenguaje más fácil de comprender para nosotros y que luego será traducido a lenguaje de máquina que es lo que puede entender el procesador.

Los lenguajes compilados son aquellos en los que el código del programador es traducido por completo de una sola vez mediante un proceso llamado "compilado" para ser ejecutado por un sistema predeterminado. Entre los más comunes encontramos "C", "C++", Java, etc.

Y los lenguajes Interpretados son aquellos en los que el código del programador es traducido mediante un intérprete a medida que es necesario. Entre los más comunes encontramos "Python", "Ruby", "Javascript", etc.

Multiparadigma:

Python es un lenguaje que soporta más de un paradigma, suponiendo paradigma como modelo de desarrollo (y cada lenguaje tiene el suyo). Estos son los de Python:

- Imperativo: La principal diferencia es que en la programación **imperativa** se describe paso a paso un conjunto de instrucciones que deben ejecutarse para variar el estado del programa y solucionar el problema. Mientras que en la declarativa sólo se procede a describir el problema que se quiere solucionar.
- Funcional: La programación funcional es un paradigma basado en el uso de funciones matemáticas que permite la variación del programa mediante la mutación de variables. Esto nos permite operar con datos de entrada y salida, ofreciendo la posibilidad de introducir datos que serán procesados para darnos otros datos de salida.
- Orientado a Objetos (POO): Los objetos son entidades que tienen un determinado estado, las entidades son propiedades que los diferencian. Los objetos manipulan otros objetos de entrada para la obtención de resultados (salida) específicos donde cada objeto nos ofrece una función específica y también nos permite la agrupación de bibliotecas o librerías

De tipado dinámico:

Una variable puede tomar diferentes valores de distintos tipos en diferentes momentos. En Python las variables son declaradas por su contenido y no por su contenedor, lo que nos va a permitir cambiar el valor y tipo de una variable durante la ejecución sin necesidad de volver a declarar.

Por ejemplo supongamos que *X* es una variable, y en este caso *X* = 1. *X* está almacenando un número, pero podemos almacenar una letra durante la ejecución del programa en la misma variable *X* mediante una instrucción, por ejemplo *X* = "a". En Python, si dentro de la variable colocamos números, la variable será de tipo numérica, si colocamos letras, la variable será de tipo texto y si colocamos un booleano (true o false), será de tipo booleano.

1. Instalación de Python

¿Que es un IDE?

Definición de IDE: Sus siglas significan Entorno de Desarrollo Integrado. Básicamente es un programa que nos permite editar código fuente de manera más sencilla sin extraviarnos en muchas líneas de código permitiéndonos una mayor organización en el desarrollo de una aplicación.

Aunque su función más importante es facilitarle el desarrollo al programador un IDE también tiene otra serie de ventajas: autocompletado de código, coloración sintáctica, navegación de clases, objetos y funciones, barras de herramientas, compilador, intérprete, etc.

Los más utilizados para Python 3 son Pycharm y SublimeText. Pero si ya dispones de un IDE puedes utilizarlo o elegir otro. Es una elección personal. Incluso puedes prescindir de él y trabajar con el block de notas como editor de textos y la consola de Windows como interfaz.

2. Sintaxis

¿Qué es la sintaxis en Python?

Con sintaxis nos referimos al correcto uso y orden de las palabras que utilizamos para comunicarnos. En Python si cometemos un error de sintaxis nos mostrará un mensaje que dice: Sintax Error. La sintaxis en python es el correcto orden y uso de las palabras para indicar una instrucción al intérprete.

¿Qué es una instrucción en Python?

Una instrucción es un conjunto de datos insertados en una secuencia estructurada para ordenar al intérprete realizar una operación determinada. Por ejemplo al declarar una variable (instrucción) se debe hacer de “cierta” forma y no puede ser de otra.

Tipos de instrucciones

Existen dos tipos de instrucciones en Python:

Instrucciones simples: son aquellas órdenes que Python comprende respetando su sintaxis que comienzan y culminan en una sola línea.

Instrucciones compuestas: son aquellas que Python comprende respetando su sintaxis comenzando con una cláusula de sentencia compuesta que enlaza con “:”, y continúan debajo con una indexación conformando así un bloque de código.

Instrucciones simples:

```
b = 12

c = (2 + 2 + 8)

a = 'Hola'

print (a)
```

Instrucciones compuestas:

```
if b == c:
    print (a)
```

Las instrucciones compuestas en Python ocupan más de una línea y el intérprete sigue un orden de ejecución normalmente estructurado de arriba abajo, comenzando, obviamente por la primera línea de código. Cuando se encuentra con una instrucción compuesta, que en nuestro ejemplo, sirve para comparar si b == c, y si esta condición se cumple y efectivamente b == c deberá descender a través de la indexación a la línea que vemos como siguiente print(a). Pero si esta condición no se cumple, se saltará esa línea y continuará hasta terminar.

El indexado en Python

El indexado en Python resulta crucial para definir donde comienza un bloque y termina otro, ya que en Python como habrás leído no existen caracteres de cierre como en otros lenguajes, aquí no utilizamos ni llaves {}, ni el tan histérico punto y coma. En Python todo esto lo define el indexado en estructuras o bloques y por supuesto los paréntesis dentro de cada instrucción simple.

Para indexar una línea basta con pulsar la tecla “Tabulador” o “tab” en algún IDE. También hay quienes hacen 4 espacios. Ya que una “Tabulación” corresponde a 4 espacios en blanco.

Por ende podemos definir la indexación en Python como una práctica que se utiliza para delimitar la estructura del programa estableciendo bloques de código.

Lo importante aquí es recordar lo siguiente:

- Una línea indexada pertenece a algún bloque de código y para ello debe existir una línea más arriba que termine en “:”.
- Si una línea esta indexada, pero no pertenece a un bloque de código obtendremos un error de indexación.
- Una instrucción es simple cuando no pertenece a un bloque de código, pero tampoco está indexada.
- Una instrucción es compuesta cuando termina en dos puntos (“:”) y generalmente conforma uno o más bloques de código.
- Se puede tener un bloque de código dentro de otro bloque de código, pero se debe intentar no indexar hasta más de tres niveles.

3. Funciones en python

Definición

Una función es un conjunto de líneas de código que realizan una tarea específica y pueden tomar argumentos para diferentes parámetros que modifiquen su funcionamiento y los datos de salida. Una función nos permite implementar operaciones que son frecuentemente utilizadas en un programa y así reducir la cantidad de código.

Las funciones en python serán una parte del código de nuestro programa encargadas de cumplir algún objetivo específico definido por nosotros o por el lenguaje, recibiendo ciertos “datos de entrada” (argumentos) en los llamados parámetros para procesarlos y brindarnos “datos de salida” o de retorno.

Ejemplo de una función en python:

Podemos hacer referencia a una calculadora. Puedes dar como entrada dos números (parámetros) por ej: 10 y 30 (argumentos). ¿Pero qué operación quieres realizar con estos números? ¿Sumarlos, restarlos, dividirlos o multiplicarlos?, en este caso según la operación que elijas obtendrás diferentes resultados. Podemos ver a los números (10 y 30) como argumentos

para los parámetros de la función, estos serán los “datos de entrada“, a la operación SUMA como una “función” y al resultado como un “dato de salida“.

Parámetros –>	Operación –>	Datos de Salida
Argumentos(Datos de entrada) –>	Función –>	Resultado
10, 40	Suma	50
10, 40	Resta	-30
10, 40	Multiplicación	400
10, 40	División	0,25

Declaración de una función

Por ejemplo vamos a declarar la función suma:

```
def Suma(parametro1, parametro2):
    Resultado = parametro1 + parametro2
```

Llamada de una función

```
Suma(argumento1, argumento2)
#En este caso podria ser: Suma(10, 40)
```

Pero si ahora yo quisiera sumar 2 y 5 lo único que tengo que hacer es cambiar los “argumentos” y cambiarían los resultados. No es necesario cambiar nada en la función. Por eso las funciones nos ahorran escribir mucho código, solo necesitamos definir una función y enviando diferentes argumentos a sus parámetros obtendremos diferentes resultados.

Tipos de funciones en python

Hay dos tipos de funciones, las que podemos crear nosotros y aquellas que ya vienen predefinidas por el lenguaje. Hay muchas funciones que podemos “Llamar” y utilizarlas sin nosotros declararlas (crearlas).

Primeramente nos vamos a enfocar en las predefinidas por el lenguaje que son funciones que ya vienen declaradas para facilitarnos las cosas y poder empezar a programar rápidamente.

Sintaxis de la Función Print

Una de las primeras funciones que debemos aprender es la función print. Esta función lo que hace sencillamente es imprimir en pantalla lo que nosotros le indiquemos como argumento.

Vamos a ver ahora la sintaxis de la función. Por ejemplo, un posible uso de esta función sería:

```
Print (“Hola, esto es una cadena de texto”)
```

Ejecutando este código estamos invocando la función predefinida print y nos debería mostrar en pantalla “Hola esto es una cadena de texto”. Puesto que la función toma como argumento para parámetro el texto que está entre comillas.

El texto siempre debe estar entre comillas porque si no, se interpretará a las palabras como algún objeto que por supuesto no existe y nos dará un error de sintaxis.

Respeta la sintaxis de la llamada de las funciones en python colocando después del nombre de la función los paréntesis. Aunque no lleve ningún argumento, debe llevar los paréntesis que serán los que indiquen que es una función. Por ejemplo: print() , no imprimirá nada, pero no nos dará ningún error.

El intérprete de Python como bien ya sabemos es el que va a procesar nuestro código y permitir que nuestro ordenador lo ejecute traduciendo nuestro lenguaje a código máquina, así que venga vamos a darle algunas órdenes.

Acceder al intérprete de Python

Windows

Debes ir a Inicio, ejecutar y allí escribes cmd que es el símbolo de sistema de Windows.

Una vez hecho esto escribe “python” (sin las comillas) y deberías estar dentro del intérprete viendo el símbolo “>>>” que es el prompt de Python.

Linux

Debes abrir un terminal y allí escribir “python3” sin las comillas, eso si has instalado la versión 3, de lo contrario solo escribe “Python”

Otra alternativa es que lo tengas integrado en tu IDE y en tal caso, cada IDE tiene su forma de acceso al intérprete

Si has accedido correctamente al intérprete deberías ver el símbolo “>>>” que es el prompt del intérprete de Python y significa que está listo para recibir nuestras instrucciones.

Nuestro intérprete va a ser como si tuviéramos un sistema operativo en modo comando y le daremos funciones codificadas en Python para que las ejecute. ¿Cómo podríamos darle al intérprete una serie de órdenes consecutivas sin tener que darlas de una en una? La respuesta es la misma que si lo hiciéramos para un sistema operativo: por medio de un script. Programar en Python simplemente será generar scripts. Los archivos .py corresponden a un archivo de script de Python. Es normal encontrar por la red numerosos scripts que podremos descargar en “.py” para ejecutarlos con alguna finalidad específica.

4. Datos estructurados en python

Un dato puede definirse en la vida real diaria como representación de un hecho o acción. Por ejemplo una entrada a un recital, es un dato que brinda la información de compra y (si está cortada o no, su asistencia al mismo), entre otras cosas.

Partiendo de esto podemos decir que este dato se convierte en información cuando resulta relevante para el usuario que lo necesita. Entonces podemos destacar aquí la diferencia entre dato e información siendo esta entrada por ejemplo, útil tanto para el fan que asistirá al evento, como para el portero para verificar la autenticidad de compra, entre otras cosas.

En informática la definición de dato no está muy lejos de ser parecida. Nosotros lo definimos como una representación simbólica (la cual podemos simbolizar con números, verdadero o falso, letras, caracteres, etc.) de un atributo o variable cuantitativa (de cantidad) o cualitativa (de cualidad).

Datos aislados vs Información.

Estos datos, solos, es decir, aislados no significan nada. Pero en determinado entorno, con determinadas necesidades de un usuario, pueden significar mucho.

Y es aquí cuando estos datos aislados, pasan a convertirse en información valiosa para nuestro usuario. Y esto es importante para todo programador: conocer la relevancia de los datos; las necesidades del usuario del programa. Con esto, evitamos servir al usuario datos irrelevantes

y/o innecesarios que solo recargarían las bases de datos y serían excesivamente molestos. El mundo avanza rápido y con ello también la gestión de los datos.

Los datos en programación pueden subdividirse en datos del tipo simple en su estado puro y del tipo estructurado o complejo ordenado.

Datos simples en python

Podemos definir un dato simple como aquel que permite que una variable pueda almacenarlo como un único valor de ese único tipo. Y presta especial atención a "valor", "único valor" y "único tipo". Pues no podría ser un dato simple un conjunto de valores (porque es más de uno), de "diferentes tipos".

Ejemplo de datos simples asignados a una variable:

```
a = 23
```

```
a = 233
```

```
a = 2.33
```

La variable "a" toma por valor al inicio el valor 23 que es un dato simple (número entero), y luego el valor 233 que también es un dato simple (número entero), finalmente a toma como valor 2.33 (número decimal o flotante) que también es un dato simple. Por lo que la variable siempre permanece simple.

Datos estructurados en python

Los datos estructurados en python son aquellos que permiten que una variable pueda almacenarlos como más de un valor, dato o tipo de dato. Como puede ser un conjunto de números, una cadena de caracteres, letras, lista, secuencia, etc. Ejemplificamos:

```
a = (2, 3, 4)
```

```
a = 'Marcos'
```

```
a = """Versículo de la biblia"""
```

La variable a no almacena un único dato por ende se puede decir que almacena datos estructurados.

Porque aunque 'Marcos' parezca un solo dato, no lo es, es una cadena de caracteres. Como podría ser también un conjunto de números, obviamente es un conjunto. Entonces no es un único valor.

¿Y si fuera solo una letra?. Pues tampoco, porque el tipo de variable permite que se guarde más de un carácter y por ende permite "más de un único valor de ese tipo". Así que una letra, será dato estructurado, complejo, aunque no lo quieras.

Tipos de datos simples en python

Como dijimos en Python una variable almacena un valor del tipo simple cuando establecemos un valor único. Por ejemplo un número almacenado en la variable "n" será un tipo de dato simple cuantitativo. Y esta variable por supuesto será del tipo Int (Entero), o sea una variable simple.

En el caso de las variables simples en python existen tres tipos: Int (Enteros), Bool (True, false), float (reales).

Recordemos que en python, el tipo de variable estará determinado por el tipo de dato que almacena; y si se modifica dicho dato y tipo se modificará automáticamente el tipo de variable.

Si una variable “a” almacena un número entero ejemplo: 2 y luego cambiamos este número 2 por la palabra “dos”; esta variable cambiará su tipo. Ha cambiado el tipo de dato que contiene de simple (int) a estructurado (str). Ejemplificamos:

Tipos de datos estructurados en python

En programación las variables que almacenen varios valores (del mismo tipo o no) al mismo tiempo serán consideradas variables del tipo estructurado o complejas. Sucede lo mismo con Python, donde podemos encontrar como en el ejemplo anterior cadena de caracteres.

Podemos encontrar diferentes tipos de datos estructurados, que serán almacenados en diferentes tipos de variables estructuradas o complejas. Y que a su vez pueden clasificarse en mutables (se pueden modificar en tiempo de ejecución) o inmutables (no se pueden modificar en tiempo de ejecución).

Es importante que distingamos los tipos de datos y por ende los tipos de variables que los van a almacenar aunque Python lo determine automáticamente.

Simples vs Estructurados en Python

Los tipos de datos simples son:

- Números: del tipo Entero, Flotante, Long, Complex
- Booleanos: True o False (Verdadero o falso)

Y los tipos de datos estructurados son:

- Secuencias: Listas, Tuplas, String, unicode, xrange, range
- Mapeos: Diccionarios
- Conjuntos: set, frozenset
- Otros: File, None, Notimplementedtype

Puedes saber si un dato es simple o estructurado aplicando la función Type() en Python brindando como argumento la variable que lo almacena, si es diferente de Int, Bool, Float. Es probable que almacene un dato estructurado.

No hay que olvidar que los datos estructurados son aquellos que conservan cierta estructura, orden, organización y radica allí la importancia de aprender a diferenciarlos en parte por su utilidad y aplicación. No es lo mismo una variable que almacena diez números ordenados (dato estructurado), que diez variables que almacenan un número cada una (datos simples).

5. Tipado de variables

Definición

Una variable es un espacio en un sistema de almacenaje que recibe un identificador y contiene información. Podría verse como una caja donde vamos a almacenar e intercambiar datos de determinados tipos.

Declarando variables

En python al declarar una variable no necesitamos especificar el tipo de dato que vamos a almacenar en ella como en otros lenguajes de programación. Declaremos 4 en el intérprete de la siguiente manera.

caja1 = "Hola pin pon"

caja2 = 18

caja3 = 18.1

caja4 = True

Sintaxis

Si el dato a almacenar es texto, éste debe ir entre comillas.

Los nombres de variables son secuencias de letras y números ya sea en mayúscula o minúscula pero preferentemente siempre comenzando por una letra, también podemos usar guion bajo "_".

Tipos de variables

Si ya has declarado las antes mencionadas puedes saber su tipo utilizando la función predefinida Type() que nos permitirá saber el tipo de una variable en un momento en específico, porque recuerda que puedes cambiar el contenido de alguna de ellas durante la ejecución del programa y así, variar su tipo.

Para llamar la función type() sólo hay que escribir en el intérprete la misma y dentro de los paréntesis le das como parámetro el nombre de la variable, así: type(caja1)

Imprimiendo variables

Vamos a utilizar la función print para imprimir estas variables antes declaradas en pantalla y ver así su contenido colocando como argumento nuestras variables a la función:

print (caja1)

print (caja2)

print (caja3)

print (caja4)

O bien:

print (caja1, caja2, caja3, caja4)

Si queremos imprimir estas variables y seguido su tipo:

print (caja1, type(caja1))

print (caja2, type(caja2))

print (caja3, type(caja3))

print (caja4, type(caja4))

Como se puede ver en este código anterior estamos colocando la función type dentro de un print, separado por comas y dentro de los paréntesis pertenecientes a la función print.

6. Listas, tuplas y range

Las Listas y Tuplas en python se refieren a una zona de almacenamiento contiguo donde podemos guardar numerosos elementos. Es como si almacenamos varios datos dentro de una misma variable. Pueden ser también mutables o inmutables.

Mutables son aquellas que se pueden editar en tiempo de ejecución, es decir, mientras nuestro programa o script está funcionando. E inmutables, todo lo contrario: podremos editarlas pero no mientras el programa se ejecute.

Listas en python

Como dijimos anteriormente podemos definir una lista como una zona de almacenamiento contigua donde podemos almacenar numerosos y diferentes tipos de datos, la misma puede editarse, modificarse mediante métodos siendo mutable.

Un conjunto de datos que pueden ser simples o estructurados agrupados dentro de una variable a la que python le asigna el tipo lista.

Sintaxis de Listas[ ]

Para crear una lista simplemente lo hacemos como si declaramos una variable pero almacenando en ella dentro de corchetes los elementos o ítems:

Objetos = ["Casa", "coche", "puerta"]

Numeros = [111, 222, 333, 444, 555, 666]

También pueden ser creadas sin especificar los elementos que contienen, porque al ser mutables los podemos agregar después. No es así en el caso de las tuplas que obligadamente debemos especificar al momento de crearla cuáles serán los elementos que vamos a almacenar en ella.

Para crear una vacía lo hacemos de igual modo pero sin elementos, siempre respetando la sintaxis y colocando los debidos “corchetes”: List1 = []

Tuplas en python

Las tuplas son variables donde se almacenan diferentes tipos de datos estructurados que deben estar ordenados desde el comienzo. Las tuplas son inmutables, no se pueden modificar durante la ejecución del programa. Por ende al crearla se deben colocar los elementos que se van a almacenar. Veamos ahora un ejemplo de tuplas y su sintaxis:

Sintaxis de Tuplas( ):

Cosas = ("casa", "puerta", "reloj", "mesa", "silla", "banco", "cuadro", "alfombra")

Numeros = (1, 2, 3, 4, 55)

Diferencias y similitudes entre listas y tuplas:

Listas en Python	Tuplas en Python
Son Dinámicas	Son Estáticas
Utilizamos Corchetes [ ]	Utilizamos Paréntesis ( )
Elementos separados por coma	Elementos separados por coma
List = [1, 2]	Tupla = (1, 2)
Acceso a elementos [Índice]	Acceso a elementos [Índice]

Entre las similitudes podemos apreciar que en ambos tipos el acceso a los elementos es a través de un índice entre corchetes, pese a que las tuplas se definan entre paréntesis. El conjunto de índices son lo número naturales, comenzando por el 0.

Cortando listas y tuplas en Python

Para acceder a un elemento de ambos tipos de datos se coloca su Índice entre corchetes. y, ¿si queremos mostrar un "pedazo", "parte" o rebanada de la lista o tupla y no sólo un elemento?. Para eso podemos especificar un índice de inicio y un índice de final y entonces nos mostrará una parte:

Numeros[0:3]

Resultado: 111, 222, 333

En este caso accedemos a la Lista "Numeros" que creamos anteriormente y mostramos del indice "0" al "3". Donde el primer número entre corchetes marca el comienzo y el segundo (tras los dos puntos (:)) el final (no incluido). Si ahora cambiamos el número del inicio (cero) por el (uno) comenzará a mostrar desde el índice 1:

Numeros [1:3]

Resultado: 222, 333

También podemos indicarle que nos la muestre completa excepto los últimos elementos de ella, indicando como final un número negativo, por ejemplo:

Numeros[:-2]

Resultado: 111, 222, 333

Métodos de listas en Python (modificarlas)

Vamos a ver como añadir o quitar elementos mediante los métodos append, insert, remove. Porque por supuesto puedes añadir todos los elementos que quieras en el código fuente. Pero en este caso vamos a agregar código para que cambie mientras nuestro programa se ejecuta.

Append: Sirve para agregar un elemento al final de una lista. Se coloca el nombre de ella que vamos a modificar seguido de un punto, el método (append en este caso) y luego entre paréntesis el elemento a agregar:

List1 = ['Marcos', 'Roberto', 'Celeste', 'Margarita']

print (List1)

Lista1.append ('Alex')

print (List1)

Resultado del 1er print: ['Marcos', 'Roberto', 'Celeste', 'Margarita']

Resultado del 2º print: ['Marcos', 'Roberto', 'Celeste', 'Margarita', 'Alex']

Extend: Agrega varios elementos al final de una lista o también añadir una nueva lista. Usando el método extend nos permite extenderla:

List1 = ['Marcos', 'Roberto', 'Celeste', 'Margarita']

```
print (List1)

List1.extend('JOSE')

print (List1)
```

Resultado del 1er print: [‘Marcos’, ‘Roberto’, ‘Celeste’, ‘Margarita’]

Resultado del 2º print: [‘Marcos’, ‘Roberto’, ‘Celeste’, ‘Margarita’, ‘J’, ‘O’, ‘S’, ‘E’]

```
Lista1 = [1, 2, 3]

Lista2 = [4, 5, 6]

Lista1.extend(Lista2) #Añadimos la lista2 como extensión de Lista1

print (Lista1)
```

Resultado del print: [1, 2, 3, 4, 5, 6]

Insert: Para añadir un elemento en el puesto i, indicando el nombre de la lista, seguido de un punto, el método (Insert en este caso) y luego entre paréntesis indicamos el índice el elemento separados por una coma:

```
List1 = ['Marcos', 'Roberto', 'Celeste', 'Margarita']

print (List1)

List1.insert (1, 'Alex')

print (List1)
```

Resultado del 1er print: [‘Marcos’, ‘Roberto’, ‘Celeste’, ‘Margarita’]

Resultado del 2º print: [‘Marcos’, ‘Alex’, ‘Roberto’, ‘Celeste’, ‘Margarita’]

Como podemos ver nuestro elemento “Alex” se añadió en el puesto uno, como lo indicamos y se le asignó otro índice a los siguientes.

Pop: Para eliminar un elemento de la lista, debemos invocar a la lista con el nombre, seguido del punto y el nombre del método (pop) y entonces entre paréntesis indicamos el índice del elemento a eliminar.

```
List1 = ['Marcos', 'Roberto', 'Celeste', 'Margarita']

print (List1)

List1.pop (1)

print (List1)
```

Resultado del 1er print: [‘Marcos’, ‘Roberto’, ‘Celeste’, ‘Margarita’]

Resultado del 2º print: [‘Marcos’, ‘Celeste’, ‘Margarita’]

Si al método pop no le indicamos índice nos eliminará el último elemento de nuestra lista.

Remove: Nos sirve para eliminar un elemento de listas en llamándolo por su valor, o por su índice:

print (List1) #en este caso continuamos con la lista como había quedado antes

List1.remove ('Marcos')

print (List1)

Resultado del 1er print: [‘Marcos’, ‘Celeste’, ‘Margarita’]

Resultado del 2º print: [‘Celeste’, ‘Margarita’]

El tipo Range

A partir de Python 3, Range es un tipo de dato estructurado y no una función. Es de fácil confusión pues range() lo que hace básicamente es crear un rango de valores, según se le especifique como argumentos, desde dónde comienza a seleccionar, hasta dónde y con qué intervalos; como si fuera una función que crea así una sucesión aritmética consecutiva como si fuera una lista inmutable que puede imprimirse, guardarse, etc.

Sintaxis de Range()

range(*n*): Donde “*n*”, que está entre paréntesis como un argumento, será el número hasta el cual se crea la sucesión de valores terminando en “n-1”. Es decir que terminará antes del número que especificaste, no incluyéndolo.

Si combinamos la función que crea listas con este tipo de datos estructurado podremos ordenarle al intérprete lo siguiente:

x = list(range(5))

x será una variable - > List() la función que convierta a range(5) en una lista.

#Con esto, la sucesión creada por range se almacenará en una lista que será la variable x

Una vez hecho esto solo nos queda imprimir x para comprobar si la lista se creó.

print (x)

Resultado:

[0, 1, 2, 3, 4]

Lo que hicimos fue indicarle al intérprete en la variable “x” quiero que almacenes una lista creada a partir de un rango de 5.

Pero, si en vez de una lista del 0 al 5, ¿queremos una del 4 al 20, pero saltándonos los números impares?

range acepta 3 argumentos. El número inicial a partir de cual comienza a contar (i), el número de paso (s), y el final (f) que indica hasta dónde llega este rango.

Entonces, la lista que debía ser desde el 2 al 20 saltando los impares lo hacemos así:

x = list(range(4, 20, 2))

x será una variable - > List() la función que convierta a range() en una lista.

En range indicamos I = 2

En range indicamos f = 20

En range indicamos s = 2

#Inicia del 2 al final 20, saltando de 2 en 2.

#Imprimimos la lista x

print (x)

Resultado:

[4, 6, 8, 10, 12, 14, 16, 18]

7. Diccionarios

Son estructuras de datos utilizadas para asignar etiquetas arbitrarias a valores.

Con estructura nos referimos a un orden predeterminado que nos permite relacionar (crear una base de datos ordenada gráficamente) valores mediante pares "etiqueta – valor".

De esta forma podremos guardar "Valores" (Juan, 55, Masculino, Si, No, 4, $16000) en "etiquetas" (Nombre, Edad, Genero, Hijos, Casado, Cuantos Hijos, Ingresos) de una manera más ordenada y accesible.

Sintaxis de un Diccionario en python

Para crearlo hacemos igual que si declaramos una lista. El nombre del diccionario seguido del identificador " = "y luego entre llaves { } agrupamos etiqueta : valor separadas por comas, así:

MiDiccionario = {etiqueta1 : valor, etiqueta 2 : valor, etiqueta : valor }

Accediendo a un diccionario

Para acceder a un valor y que se refleje por pantalla a través de la función print lo deberíamos conseguir con: print (Midiccionario ['etiqueta'])

Si el valor de esa etiqueta fuera una lista, se podría especificar qué valor de esa lista quiero conseguir:

print (Midiccionario ['clave'][0:2])

En este caso indicamos que nos imprima los elementos de la lista con índices desde el 0 al 2.

Agregar, reemplazar y modificar elementos

Para agregar en un diccionario una nueva etiqueta o cambiar el valor que tuviera asignado, si ya existía previamente, simplemente lo tenemos que asignar:

NombreDelDiccionario[etiqueta] = valor

Si quisiéramos cambiar una clave por otra

NombreDelDiccionario[etiqueta _nueva] = NombreDelDiccionario.pop(etiqueta _vieja)

Para eliminar un par (etiqueta : valor) podemos recurrir a la función **del**:

del(NombreDelDiccionario[etiqueta])

Métodos de los diccionarios

Método **get**: Sirve para obtener el valor de una etiqueta determinada:

MiDiccionario.get(etiqueta, "valor por defecto, por si la etiqueta no tiene uno")

Resultado: valor

El método keys() nos servirá para imprimir solo las etiquetas del diccionario

Y el método value() nos permite imprimir solo los valores

etiquetas = MiDiccionario.keys()

Resultado: dict_keys([etiqueta1, etiqueta2, …, etiquetan])

valores = MiMarga.values()

Resultado: dict_values([valor1, valor2, …, valorn])

8. Condicionales

Los condicionales if, else, elif en python se utilizan para ejecutar una instrucción en caso de que una o más condiciones se cumplan. Un condicional es como el momento en que se debe tomar una decisión en nuestro programa o script. Dependiendo de la decisión que se tome ocurrirá una cosa u otra, o ninguna. La comprensión de los condicionales es un elemento clave en la programación pues es lo que determinará que un programa sea dinámico y cambie según diferentes condiciones.

Sintaxis de los condicionales:

Para utilizar el if solo basta con agregarlo en el siguiente orden:

IF + (Condición) + ":"

Y debajo de los dos puntos, indexada adecuadamente la instrucción a realizar en caso de que la condición se cumpla.

```
a = 2 + 2

if a == 4: #condicion si a es exactamente cuatro, entonces(:)

   print ("a es igual a cuatro")
```

Como vemos en el trozo de código anterior definimos primero el valor de la variable "a" como la suma de dos números. Luego colocamos una condición if que determina que si "a" es igual a "4" entonces nos imprime "a es igual a cuatro"

Pero si cambiamos el valor de "a" indicando que la variable "a" es igual a la suma de los números "2 + 3"

```
a = 2 + 3

if a == 4: #condicion si a es exactamente cuatro, entonces(:)

   print ("a es igual a cuatro")

#Resultado:
```

En este caso no se cumpliría la condición porque 2 más 3 no es 4. Es 5 y la condición se cumple solo si “a” es igual a cuatro. Así que no sucederá nada.

Después de una sentencia if podemos agregar un else para que se ejecutara otro código en caso que la condición no se cumpliera y no nos pasará como en el último ejemplo.

```
a = 2 + 3

if a == 4: #condicion si a es exactamente cuatro, entonces(:)

   print ("a es igual a cuatro") # Imprimir

else:

   print ("No se cumple la condicion")

#Resultado: "No se cumple la condicion"
```

En este caso agregaremos dos elif al mismo código para comprobar si a fuera igual a cinco o a seis:

```
a = 2 + 3

if a == 4: #condicion si a es exactamente cuatro, entonces(:)

   print ("a es igual a cuatro") # Imprimir

elif a == 5:

   print ("a es igual a cinco")

elif a == 6:

   print ("a es igual a seis")

else:

   print ("No se cumple la condición")

#Resultado: "a es igual a cinco"
```

Al no cumplirse la condición IF el intérprete seguirá comprobando las elif y se encuentre con la primera donde coincide que “a” es igual a cinco y por ende ejecutará esta instrucción imprimiendo “a es igual a cinco”

La estructura es siempre IF luego ELIF y finalmente ELSE (que se ejecuta en caso que no se cumpla ninguna de las anteriores IF/ELIF)

Si experimentas y cambias el valor de “a” a por ejemplo (2 + 5) se ejecutará el else, porque no se cumple ninguna condición anterior en los IF/ELIF

9. Operadores

Definición

Los operadores son símbolos matemáticos que llevan a cabo una operación específica entre los operandos. Los operadores pueden recibir operandos variables.

Podríamos entenderlo más fácilmente con un ejemplo:

En el caso de una SUMA el operador utilizado es el símbolo “más” (+) y podemos sumar tanto números, como letras o variables como hemos visto anteriormente. Estos números o variables serían los operandos y cuando decimos que pueden ser dinámicos nos referimos a variables, que pueden cambiar su valor.

Tipos de operadores en python:

Aritméticos

Los operadores aritméticos son aquellos que se utilizan para realizar operaciones matemáticas sencillas:

Operador	Función	Ejemplos	Resultado
“+”	Sumar	2 + 2	4
“-“	Restar	3 - 2	1
“*”	Multiplicar	2 * 2	4
“/”	Dividir	4 / 2	2
“%”	Módulo: Devuelve el resto de la operación	4 % 2	0
“**”	Potencia	3 ** 2	9
“//”	División entera: Devuelve la parte entera del cociente	9//4	2

Comparación

Los operadores de comparación son aquellos que se utilizan para comparar valores y nos devolverá True / False como resultado de la condición.

Operador	Función	Ejemplos	Resultado
“==”	Si dos valores son exactamente iguales devuelve True	2 == 2 2 == 3	True False
“!=“	Si dos valores son diferentes devuelve True	2 != 5 2 != 2	True False
“>”	Si el valor de la izquierda es mayor que el de la derecha devuelve True	4 > 2 1 > 2	True False
“<”	Si el valor de la izquierda es menor que el de la derecha devuelve True	1 < 2 4 < 2	True False
“>=”	Si el valor de la izquierda es mayor o igual que el de la derecha devuelve True	4 >= 2 4 >= 4 1 >= 2	True True False
“<=”	Si el valor de la izquierda es menor o igual que el de la derecha devuelve True	1 <= 2 4 <= 4 4 <= 2	True True False

Asignación

Los operadores de asignación son aquellos que utilizamos para asignarle un valor a variable, lista, tupla, conjunto, etc. Y no sólo existe el símbolo igual “=” como operador de asignación, hay también otras combinaciones de símbolos al igual que las anteriores que vimos que nos van a permitir ahorrar código.

Operador	Función	Ejemplo	Resultado
“=”	Asigna un valor a un elemento. Puede ser variable, lista, diccionario, tupla, etc	a = 2 + 2	“a” vale 4
“+= “	El primer elemento es igual a la suma del primer elemento con el segundo.	b += 1	b = b + 1 Cada vez que se ejecute esta instrucción se le sumará 1 a “b”

“-=”	El primer elemento es igual a la resta del primer elemento con el segundo.	b -= 1	b = b – 1 Cada vez que se ejecute esta instrucción se le restara 1 a “b”
“*=”	El primer elemento es igual a la multiplicación del primer elemento con el segundo.	b *= 2	b = b * 2 Cada vez que se ejecute esta instrucción “b” se multiplicará por dos y se le asignará el valor del resultado.
“/=”	El primer elemento es igual a la división del primer elemento con el segundo.	b /= 2	b = b / 2 Cada vez que se ejecute esta instrucción “b” se dividirá por dos y se le asignará el valor del resultado.
“%=”	El primer elemento es igual al resto de la división del primer elemento con el segundo.	b %= 2	b = b % 2 Cada vez que se ejecute esta instrucción “b” se dividirá por dos y se le asignará el resto del resultado.
“**=”	El primer elemento es igual al resultado de la potencia del primer elemento con el segundo.	b **= 2	b = b ** 2 Cada vez que se ejecute esta instrucción “b” se elevará a dos y se le asignará el valor del resultado.

Lógicos

Los operadores lógicos son and (y) or (o) not (no) y sirven para comprobar si dos o más operandos son ciertos (True) o falsos (false) y nos devolverá como resultado True o False. Normalmente los solemos utilizar mucho en los condicionales para devolver un booleano comparando varios elementos.

Operador	Función	Ejemplos	Resultado
“AND”	Si y sólo si todos los elementos son True dará por resultado True. Si no, False	True AND True True AND False False AND True False AND False	True False False False
“OR”	Si algún elemento es True dará por resultado True. Si no, False	True OR True True OR False False OR True False OR False	True True True False
“NOT”	El operador “not” es unario, de negación por lo que sólo dará True si opera con False y viceversa.	NOT False NOT True	True False

Especiales

Existen otros operadores especiales que veremos y utilizaremos comúnmente en bucles, para comprobar si una variable es exactamente igual a otra o no, o para saber si un elemento se encuentra dentro de otro, etc..

Operador	Función	Ejemplos	Resultado
“In”	El operador “in” devuelve True si un elemento se encuentra dentro de otro.	a = [3, 4] 3 in a	True Porque “3” se encuentra en “a”
“Not in”	El operador “not in” devuelve True si un elemento no se encuentra dentro de otro.	a = [3, 4] 5 not in a	True Porque “5” no se encuentra en “a”

“Is”	El operador “is” devuelve True si los elementos son exactamente iguales.	x = 10 y = 10 x is y	True Porque ambas variables tienen el mismo valor: son iguales.
“Not Is”	El operador “not is” devuelve true si los elementos no son exactamente iguales.	x = 10 y = 111 x not is y	True Porque las variables no tienen el mismo valor: son diferentes.

10. Funciones

Funciones predefinidas en python

Vamos a ver una tabla de funciones internas (o funciones predefinidas en python) más utilizadas y conocidas. La sintaxis de dichas funciones es la misma que aprendimos con la función print. Debemos indicarle los argumentos para los parámetros con los que dicha función va a operar:

Funciones de cadenas en python

Función	Utilidad	Ejemplo	Resultado
print()	Imprime en pantalla el argumento.	print (“Hola”)	“Hola”
len()	Determina la longitud en caracteres de una cadena.	len(“Hola Python”)	11
join()	Convierte en cadena utilizando una separación	Lista = [‘Python’, ‘es’] ‘-‘.join(Lista)	‘Python-es’
split()	Convierte una cadena con un separador en una lista	a = (“hola esto sera una lista”) Lista2 = a.split() print (Lista2)	[‘hola’, ‘esto’, ‘sera’, ‘una’, ‘lista’]
replace()	Reemplaza una cadena por otra	texto = “Manuel es mi amigo” print (texto.replace (‘es’, ‘era’))	Manuel era mi amigo
upper()	Convierte una cadena en mayúsculas	texto = “Manuel es mi amigo” texto.upper()	‘MANUEL ES MI AMIGO’
lower()	lower() Convierte una cadena en minúsculas	texto = “MaNueL eS mI AmIgO” texto.lower()	‘manuel es mi amigo’

Funciones numéricas en python

Función	Utilidad	Ejemplo	Resultado
range()	Crea un rango de números	x = range (5) print (list(x))	[0, 1, 2, 3, 4]
str()	Convierte un valor numérico a texto	str(22)	‘22’
int()	Convierte a valor entero	int(’22’)	22
float()	Convierte un valor a decimal	float(‘2.22’)	2.22
max()	Determina el máximo entre un grupo de números	x = [0, 1, 2] print (max(x))	2
max()	Determina el mínimo entre un grupo de números	x = [0, 1, 2] print (max(x))	0
sum()	Suma el total de una lista de números	x = [0, 1, 2] print (sum(x))	3

Otras funciones útiles en python

Función	Utilidad	Ejemplo	Resultado
list()	Crea una lista a partir de un elemento	x = range (5) print (list(x))	[0, 1, 2, 3, 4]
tuple()	Convierte un valor numérico a texto	Crea o convierte en una tupla print(tuple(x))	(0, 1, 2, 3, 4)
open()	Convierte a valor entero	with open("Ejercicios/Ejercicio.py", "w") as variables: variables.writelines("Eje")	Crea el archivo "Ejercicio.py" con el contenido "Eje"
ord()	Devuelve el valor ASCII de una cadena o carácter.	print(ord('A'))	65
round()	Redondea después de la coma de un decimal	print (round(12.723))	13
type()	Devuelve el tipo de un elemento	type(x)	<class 'range'>
input()	Permite la entrada de datos al usuario	y = int(input("Introduzca el número")) print (y)	3 3

Éstas son sólo algunas funciones predefinidas más utilizadas de Python 3, pero hay bastantes más. Puedes consultar más en al manual de Python. Veamos ahora cómo crear e invocar nuestras propias funciones.

Como crear tus propias funciones

En Python, para crear una función recurrimos a la instrucción **def**, seguido del nombre que queramos dar a la función. Continuamos con paréntesis y finalizamos con dos puntos de la siguiente manera:

```
def func(parámetro, parámetro):

    #código de la función <-- Identación

    return
```

Para usar esta función, debemos llamarla después de haberla definido. Nunca podemos llamar una función que no fue creada anteriormente.

```
variable = func(argumento, argumento)
```

Al incluir un "return" le estamos diciendo a python que retorne inmediatamente el valor de resultado de la función y use la consiguiente expresión como un valor de retorno.

11. Bucles

Un bucle o ciclo en programación es la ejecución continua de un determinado bloque de código mientras una condición asignada se cumple. Existen diferentes tipos de bucles en python y nos van a servir para ejecutar nuestro código hasta obtener determinado resultado.

Bucles For

En python iteran sobre los ítem de cualquier secuencia (lista, cadena, diccionario) en el orden en que aparecen en la secuencia. Este bloque de código en el bucle se suele llamar “cuerpo del bucle” y cada repetición se suele llamar iteración.

Para no complicarnos tanto podemos entender al bucle for como el encargado de recorrer una secuencia buscando esos elementos que cumplen determinada condición y realizando alguna instrucción según sea especificada.

Sintaxis

En el caso de un bucle se coloca la sentencia seguida de la variable donde se almacenarán los ítem y después del operador “in” el elemento a iterar:

```
for variable in elemento_iterable:
    cuerpo de la repetición
```

Una vez que se termina de iterar el elemento el bucle se detiene, es decir que se ejecutará mientras se cumpla la condición.

Por ejemplo, vamos a iterar esta lista para contar los caracteres de cada palabra que contiene utilizando la función interna de cadenas len():

```
Palabras = ['Peine', 'Pelo', 'Ventana', 'Refrigerador', 'Adolescente', 'Dentista', 'Asesino']

for caracteres in Palabras: #Creamos el bucle para iterar Palabras y almacenar en caracteres
    print ((caracteres), ('###Longitud:'), (len(caracteres)))
```

Resultado:

```
Peine ###Longitud: 5

Pelo ###Longitud: 4

Ventana ###Longitud: 7

Refrigerador ###Longitud: 12

Adolescente ###Longitud: 11

Dentista ###Longitud: 8

Asesino ###Longitud: 7
```

Bucles While

Estos se ejecutan mientras la condición sea verdadera, es decir, si la condición se cumple se ejecutará el cuerpo de dicho bucle y al finalizar se volverá a comprobar la condición. Si continúa siendo verdadera, se ejecutará nuevamente. En caso contrario omitirá la misma y continuará la ejecución normal del programa. Veamos cómo crear un bucle while y para que nos ha de servir.

Sintaxis

Es realmente muy sencilla, escribimos la palabra reservada while seguida de la condición y dos puntos, debajo de esta se encontrara el cuerpo del mismo.

```
while (condición):
    Cuerpo de la repetición
```

Por ejemplo, podemos crear un contador e imprimir el resultado hasta que la condición deje de cumplirse, en este caso contaremos del 1 al 3.

```
i = 0 #Declaramos el valor de la variable "i"
while (i <= 2): #Bucle con la condición de ejecutarse mientras "i" sea menor o igual que "2"
    i +=1     #Cuerpo de la repetición = sumamos uno a i (i es igual a i mas uno)
    print (i) #Imprimimos "i"
```

Resultado:

```
1
2
3
```

Control de bucles, break, continue y pass en python

A veces podemos no querer que un bucle se continúe repitiendo infinitamente cuando ya ha cumplido su función, para esto existe el control de bucles mediante estas tres instrucciones.

Break

Esta instrucción se utiliza para finalizar un bucle, es decir, salir de él y continuar con la ejecución

```
while True: #Creamos el ciclo infinito
    opcion = (input("""Elige una fruta para tu desayuno:
    1- Manzanas
    2- Bananas
    3- Nada
    """))#Creamos un input para
        #que el usuario ingrese su opción y la almacenamos en "opcion"
    if opcion == '1': #Condicionales según la opción que eligió!
        print ("Has seleccionado manzanas")
        break #Rompemos el bucle
    elif opcion == '2':
        print ("Has seleccionado bananas")
        break #Rompemos el bucle
    elif opcion == '3':
        print ("Has seleccionado nada")
```

```
        break #Rompemos el bucle
    else:
        print ("Debes seleccionar una opcion (1, 2 o 3)")
print("Programa terminado, que disfrutes tu desayuno") # FUERA DEL BUCLE
```

El break lo colocamos al final del bloque que queremos que dé la salida, en este caso, después de que el usuario elija la cualquiera de las opciones. El usuario solo podrá salir de la repetición eligiendo la opción correcta.

Continue

La instrucción continue dentro de un bucle obliga al intérprete a volver al inicio del bucle obviando todas las instrucciones o iteraciones debajo de él.

Pass

La instrucción Pass es como lo indica su nombre una expresión nula, no hace nada. Es casi como si no existiera, pero nos permite crear un bucle sin colocar código en su cuerpo para añadirlo más tarde utilizándolo como un relleno temporal.

La diferencia con continue es que este termina la iteración actual pero continúa con la siguiente iteración del bucle, volviendo al inicio. En cambio pass no hace nada y continúa con las siguientes instrucciones de éste sin volver al inicio.

12. Excepciones

Llamamos excepciones en python a los errores generados por nuestro código fuente. Si alguna función de nuestro programa genera un error y ésta no lo maneja, el mismo se propaga hasta llegar a la función principal que la invocó y genera que el programa se detenga.

Manejar los errores nos va a permitir evitar que nuestro programa deje de funcionar de golpe y nos va a dar la posibilidad de mostrar un error personalizado al usuario en vez de los clásicos errores del intérprete python.

El bloque try / except:

Para esto vamos a tratar de ubicar el código que pueda ser capaz de producir un error dentro de un bloque Try. En el caso de generarse un error, éste buscará alguna especificación para el mismo en el bloque except.

Es decir que mediante el uso de Try/Except se logra establecer una condición en caso de que se generará un error dentro del bloque Try. En ese caso debería dispararse el código que se encuentra en el Except. En caso de no haber error se continuaría normalmente.

Except:

Se pueden crear múltiples except para cada error particular lo que nos permitirá anticiparnos a estos y generar una advertencia o corrección para cada uno.

Finally:

El bloque finally nos permite indicar las sentencias de finalización con la particularidad de que se ejecutan independientemente de que haya surgido o no un error. Si hay un bloque finally no es necesario el except y podemos aplicar un try, solo con un finally.

```
def sumar(): #Definimos la función sumar
```

```
    x = a + b
    print (("Resultado"), (x))
def restar():#Definimos la función restar
    x = a - b
    print (("Resultado"), (x))
def multiplicar():#Definimos la función multiplicar
    x = a * b
    print (("Resultado"), (x))
def dividir():#Definimos la función dividir
    x = a / b
    print (("Resultado"), (x))
while True: #Creamos un bucle
    try: #Intentamos obtener los datos de entrada
        a = int(input("Ingresa el primer numero: \n")) #Solicitamos el 1er numero al usuario
        b = int(input("Ingresa el segundo numero: \n"))#Solicitamos el 2do numero al usuario
        print (("Que calculo quieres realizar entre"), (a), ("y"), (b), ("?\n")) #Preguntamos el calc
        op = str(input(""" #Ofrecemos las opciones de cálculo
        1- Sumar
        2- Restar
        3- Multiplicar
        4- Dividir \n"""))
        if op == '1':#Si el usuario elige opción 1 llamamos a sumar
            sumar()
            break
        elif op == '2':#Si el usuario elige opción 1 llamamos a restar
            restar()
            break
        elif op == '3':#Si el usuario elige opción 1 llamamos a multiplicar
            multiplicar()
            break
        elif op == '4':#Si el usuario elige opción 1 llamamos a dividir
```

```
            dividir()
            break
        else:
            print ("""Has ingresado un numero de opcion erroneo""") #entrada erronea
    except ZeroDivisionError:
        print ("Nuestro calculador no permite dividir por cero, intenta otro calculo!")
    except:
        print ("Error")
        op = '?'
    finally:
        print ("Gracias por utilizar nuestra calculadora")
```

Aserciones

Una aserción es una comprobación de validez que puedes activar o desactivar cuando hayas terminado de probar el programa. Una expresión es probada y si el resultado es falso una excepción se activa.

Casi podría decirse que hacemos una comprobación y en caso de ser falsa se genera un error que podemos personalizar para detectar el fallo rápidamente.

Sintaxis de las aserciones

Las aserciones son llevadas a cabo a partir de la sentencia “assert“, vamos a ver un ejemplo, en este caso utilizando condiciones if para comprobar las monedas de las que dispone un usuario, pero también queremos comprobar que no sea un número negativo, y en caso de que lo sea, detener el programa enviando un error.

```
monedas = -19
if monedas >= 12:
    print (("Tienes"), (monedas), (",que es más que 12"))
elif monedas <= 12:
    print (("Tienes"),(monedas),(",que es menos que 12, lo siento consigue más monedas"))
    assert monedas > 0, "Error de comprobación, monedas es negativo"
```

Como vemos se coloca la sentencia “assert” seguido de la condición en la que se ejecutará y tras la coma entre comillas el error que será mostrado.

A menudo la sentencia assert se utiliza para comprobar errores de programación. Si se está realizando un programa con mucho código y se quiere ir comprobando que todo funcione como es correcto, se puede hacer utilizando aserciones, que permitirá detectar rápidamente los errores.

13. Ficheros

Para crear un archivo en python desde nuestro código existen diferentes maneras. Empezaremos explicando que los expertos aconsejan por su seguridad y eficiencia en la modificación de ficheros.

Crear archivos en python usando With

Esta cláusula establece automáticamente un contexto para la ejecución segura debido a que crea un punto de partida de configuraciones iniciales y al finalizar recupera los valores anteriores. Y en el caso de la apertura de un archivo, ésta automáticamente ha de cerrarlo una vez terminada la función que lo involucre, así como también se realiza una limpieza de la memoria utilizada.

De esta manera se utiliza un gestor de contexto "Context Manager" que son objetos que tienen definido los métodos __enter__ y __exit__ para inicializar el contexto y para terminarlo al igual que los objetos file, esto nos asegura que se ha de cerrar correctamente.

Abrir archivos en python mediante el objeto file

Podemos manipular una variable como un objeto file (archivo) asignándole un archivo mediante la función integrada open e indicando la ruta del mismo seguido del modo en el cual vamos a abrirlo. Entonces la variable toma como valor el archivo y se convierte en un objeto file permitiéndonos trabajar con ella utilizando los métodos.

Combinándolo con la cláusula with as lo hacemos de la siguiente manera:

```
#Forma clásica de crear un archivo:

f = open("archivotext.txt", "w") #Creamos el archivo

f.write("Creando archivo de texto en python de forma clásica") #<-Escribimos en el

f.close() #Cerramos el archivo

#Utilizando With-As

with open ("archivotext.txt", "w") as f: #Creamos el archivo

   f.write("Creando archivo de texto en python usando whit as") #<-Escribimos en el

   f.close()
```

Sintaxis

```
with open ("Nombre", "modo de apertura") as Objeto:

    Objeto.Write() #<----Método escribir del objeto
```

Métodos del objeto file en python

Sabemos que cuando abrimos un fichero existe un puntero, que es donde se posiciona el intérprete sobre el archivo. Normalmente lo hace al inicio, pero si quisiéramos abrirlo con el puntero en determinada ubicación podríamos hacerlo con los métodos de los que dispone el objeto file:

Método Seek(byte)

Este método se corresponde al puntero mencionado y junto al método tell() nos permitirá ubicarnos al final o al principio de los archivos. Pero el método seek() fue diseñado para archivos binarios, no para archivos de texto.

Este método moverá el puntero hacia el byte indicado como argumento.

seek (desplazamiento, desde_donde)

Si colocamos un cero '0' como argumento a seek nos posiciona al inicio del archivo!

Método Tell()

Devuelve el puntero a una posición en un momento dado. Junto con seek() nos permitirán posicionarnos al principio o al final.

Tell() nos devuelve un entero que indica la posición del puntero.

Método Read([bytes])

El método read nos permite leer un archivo completo, salvo que indiquemos los bytes entonces solo leerá hasta los bytes determinados.

```
with open ("pirata.txt", "r") as f: #Abrimos de modo Read (r)
    contenido = f.read() #<-Lo abrimos utilizando el metodo read
print (contenido)          #Imprimimos la variable que tiene el contenido
```

Prestemos atención a la letra "r" que colocamos como modo luego del nombre, esto significa que lo abre sólo para ser leído. Por ende no puede ser editado. Y si colocas una "w" de "Write" se abrirá en modo escribir en él. Y se borrara todo el contenido.

Método Readline([bytes])

El método readline lee una línea por vez. Si colocamos un solo readline nos leerá una sola línea y si no especificamos a partir de dónde será la primera por defecto. Veamos el código

```
with open ("pirata.txt", "r") as f: #Abrimos el archivo de modo read (r)
    linea1 = f.readline() #<-Almacenamos en variable linea1 el método readline()
    linea2 = f.readline() #<-Volvemos a almacenar en variable linea2 el método readline()
    print (linea1)
    print (linea2)
```

Cada vez que hagamos un readline() tomará la siguiente línea.

Método Readlines()

Lee todas las líneas en forma de lista separando las líneas con el carácter de escape para los saltos de línea "/n"

Método Write()

Sirve para escribir una cadena dentro del mismo. Para esto debemos abrirlo con modo de re-escribir lo cual se hace agregando "r+" en el modo de apertura. De esta manera nuestra cadena se añadirá al final.

Método Writelines(secuencia)

Nos permite escribir una secuencia (elemento iterable) dentro de él, línea a línea.

Método Close()

El método close se encarga de cerrar el archivo de forma segura.

Modos de apertura de archivos en python

El modo de apertura de nuestros archivos es importante debido a que determinan varios factores a tener en cuenta y permiten abrir archivos tanto para solo lectura como para edición, o reemplazo completo.

Modo de Solo lectura (Read) ("r")

Para especificar que lo abrimos en modo de lectura lo hacemos añadiendo la letra "r" luego del nombre. Como lo especifica su nombre solo permite al intérprete trabajar con él en modo lectura por lo que lo cargará en memoria pero no podrá editarlo.

Modo de Escritura (Write) ("w")

Al abrir un archivo en este modo el mismo será borrado y vuelto a escribir, por eso es importante distinguir este modo como de truncado. Si el archivo no existe, lo creara. Pero hay que tener en cuenta el hecho de que si el mismo existe será reescrito.

Modo de Solo escritura al final ("a")

Este modo nos permite escribir en el archivo posicionándonos al final del mismo. Si abrimos un archivo utilizando este modo siempre estaremos escribiendo a continuación de lo que ya se encuentra en el mismo sin necesidad de posicionar el puntero. También, al igual que write, si el archivo no existe, será creado.

Modo de Lectura- Escritura (+) (r+ , w+, a+)

Si a cualquiera de los anteriores le añadimos un "+" se convierte en modo de escritura y lectura. Pero ambos ("r+") y ("w+") trabajan de manera diferente. Al utilizar ("w+") si el archivo existe será truncado, es decir, se borrará todo su contenido y se sobrescribirá en él. No es así el caso de ("r+"). (a+) permanece igual, solo que al agregarle un + si el fichero no existe será creado.

Trabajo a realizar durante la práctica

Instalar un IDLE de Python en el ordenador que vayas a usar para las prácticas, o si no quieres el IDLE, ten un intérprete de Python. Si no tienes experiencia con Python practica un poco el uso de librerías externas y la programación. Para demostrar tu dominio de los primeros conceptos impartidos en clase de teoría y del lenguaje Python tienes que implementar el sistema de cifrado Vigenère:

- Escribe un informe en el que describas en pseudocódigo los tres algoritmos del sistema: $(\mathcal{K}, \mathcal{E}, \mathcal{D})$.

- La función de cifrado trabajará sobre el lenguaje castellano y las entradas a la función será a través de dos ficheros: plain.txt para el mensaje plano y clave.txt. dando como salida el fichero cifrado.txt.

- Por otro lado la función de descifrado trabajará con los ficheros: cifrado.txt y clave.txt. dando como salida el fichero descifrado.txt.

- Se deberá entregar a través de Moodle dos ficheros:

    a. El informe descriptivo de la práctica.

    b. El código fuente.

- **Trabajo Extra voluntario:** Implementar una función de criptoanálisis del sistema basado en los descritos en las clases de teoría. La entrega sería incluida en la entrega del trabaja anterior con dos ficheros más:

    a. El informe descriptivo del algoritmo de criptoanálisis usado.

    b. El código fuente.