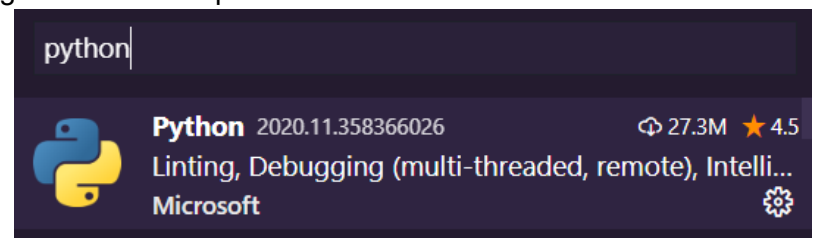
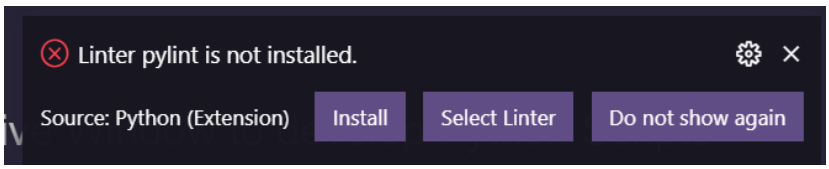
Pasos para instalar y utilizar Python en VS Code.

1) Descargar Python de https://www.python.org/downloads/ e instalar tildando la opción “Agregar Python al PATH”. Al finalizar tocar en “Disable path length limit”.

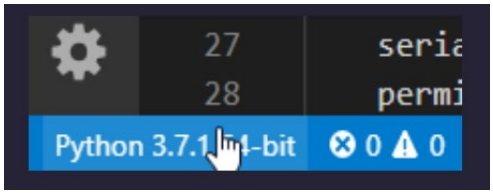
2) Descargar la extensión para VS Code.



3) Al crear un archivo .py es probable que muestre el siguiente mensaje, instalar.



4) Seleccionar el intérprete en la barra de status. (Probar primero correr un programa, muchas veces no es necesario).



Introducción a Python: Características del lenguaje

Fuente: http://www.tutorialesprogramacionya.com // https://j2logo.com

Python es un lenguaje de programación de alto nivel cuya máxima es la legibilidad del código. Las principales características de Python son las siguientes:

- Es multiparadigma, ya que soporta la programación imperativa, programación orientada a objetos y funcional.
- Es multiplataforma: Se puede encontrar un intérprete de Python para los principales sistemas operativos: Windows, Linux y Mac OS. Además, se puede reutilizar el mismo código en cada una de las plataformas.
- Es dinámicamente tipado: Es decir, el tipo de las variables se decide en tiempo de ejecución.

- Es fuertemente tipado: No se puede usar una variable en un contexto fuera de su tipo. Si se quisiera, habría que hacer una conversión de tipos.
- Es interpretado: El código no se compila a lenguaje máquina.

Expresiones y sentencias en Python

Expresión

Una expresión es una unidad de código que devuelve un valor y está formada por una combinación de operandos (variables y literales) y operadores. Los siguientes son ejemplos de expresiones (cada línea es una expresión diferente):
5 + 2 # Suma del número 5 y el número 2
a < 10 # Compara si el valor de la variable a es menor que 10
b is None # Compara si la identidad de la variable b es None
3 * (200 - c) # Resta a 200 el valor de c y lo multiplica por 3

Sentencia

Por su parte, una sentencia o declaración es una instrucción que define una acción. Una sentencia puede estar formada por una o varias expresiones, aunque no siempre es así. En definitiva, las sentencias son las instrucciones que componen nuestro programa y determinan su comportamiento.
Ejemplos de sentencias son la asignación = o las instrucciones if, if ... else ..., for o while entre otras.
❗Una sentencia está delimitada por el carácter Enter (\n).

Sentencias de más de una línea

Normalmente, las sentencias ocupan una sola línea. Por ejemplo:
a = 2 + 3 # Asigna a la variable <a> el resultado de 2 + 3
Sin embargo, aquellas sentencias que son muy largas pueden ocupar más de una línea (la guía de estilo PEP 8, recomienda una longitud de línea máxima de 72 caracteres).
Para dividir una sentencia en varias líneas se utiliza el carácter \. Por ejemplo:

```
a = 2 + 3 + 5 + \
  7 + 9 + 4 + \
  6
```

Además de la separación explícita (la que se realiza con el carácter \), en Python la continuación de línea es implícita siempre y cuando la expresión vaya dentro de los caracteres (), [] y {}.
Por ejemplo, podemos inicializar una lista del siguiente modo:

```
a = [1, 2, 7,
   3, 8, 4,
   9]
```

Bloques de código (Indentación)

Lo último que veremos sobre sentencias en esta introducción a Python es cómo se pueden agrupar en bloques de código.
Un bloque de código es un grupo de sentencias relacionadas bien delimitadas. A diferencia de otros lenguajes como JAVA o C, en los que se usan los caracteres {} para definir un bloque de código, en Python se usa la indentación o sangrado.
El sangrado o indentación consiste en mover un bloque de texto hacia la derecha insertando espacios o tabuladores al principio de la línea, dejando un margen a la izquierda.

Un bloque comienza con un nuevo sangrado y acaba con la primera línea cuyo sangrado sea menor. De nuevo, la guía de estilo de Python recomienda usar los espacios en lugar de las tabulaciones para realizar el sangrado. Yo suelo utilizar 4 espacios.

Veamos todo esto con un ejemplo:

```
def suma_numeros(numeros): # Bloque 1
  suma = 0              # Bloque 2
  for n in numeros:     # Bloque 2
     suma += n          # Bloque 3
     print(suma)        # Bloque 3
  return suma           # Bloque 2
```

Como te decía en la sección anterior, no hace falta todavía que entiendas lo que hace el ejemplo. Simplemente debes comprender que en la línea 1 se define la función suma_numeros. El cuerpo de esta función está definido por el grupo de sentencias que pertenecen al bloque 2 y 3. A su vez, la sentencia for define las acciones a realizar dentro de la misma en el conjunto de sentencias que pertenecen al bloque 3.

Convenciones de nombres en Python

A la hora de nombrar una variable, una función, un módulo, una clase, etc. en Python, siempre se siguen las siguientes reglas y recomendaciones:

- Un identificador puede ser cualquier combinación de letras (mayúsculas y minúsculas), números y el carácter guión bajo (_).
- Un identificador no puede comenzar por un número.
- A excepción de los nombres de clases, es una convención que todos los identificadores se escriban en minúsculas, separando las palabras con el guión bajo. Ejemplos: contador, suma_enteros.
- Es una convención que los nombres de clases sigan la notación Camel Case, es decir, todas las letras en minúscula a excepción del primer carácter de cada palabra, que se escribe en mayúscula. Ejemplos: Coche, VehiculoMotorizado.
- No se pueden usar como identificadores las palabras reservadas.
- Como recomendación, usa identificadores que sean expresivos. Por ejemplo, contador es mejor que simplemente c.
- Python diferencia entre mayúsculas y minúsculas, de manera que variable_1 y Variable_1 son dos identificadores totalmente diferentes.

Palabras reservadas de Python

Python tiene una serie de palabras clave reservadas, por tanto, no pueden usarse como nombres de variables, funciones, etc.
Estas palabras clave se utilizan para definir la sintaxis y estructura del lenguaje Python.
La lista de palabras reservadas es la siguiente:
and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, yield, while y with

Constantes en Python

Terminamos esta introducción a Python señalando que, a diferencia de otros lenguajes, en Python no existen las constantes.
Entendemos como constante una variable que una vez asignado un valor, este no se puede modificar. Es decir, que a la variable no se le puede asignar ningún otro valor una vez asignado el primero.
Se puede simular este comportamiento, siempre desde el punto de vista del programador y atendiendo a convenciones propias, pero no podemos cambiar la naturaleza mutable de las variables.

Entrada/Salida en Python

Para el ingreso de un dato por teclado y mostrar un mensaje se utiliza la función **input**, esta función retorna todos los caracteres escritos por el operador del programa:

lado=input("Ingrese la medida del lado del cuadrado:")

La variable lado guarda todos los caracteres ingresados pero no en formato numérico, para esto debemos llamar a la función int:

Un formato simplificado para ingresar un valor entero por teclado y evitarnos escribir las dos líneas anteriores es:

lado=int(input("Ingrese la medida del lado del cuadrado:"))

Procedemos a efectuar el cálculo de la superficie luego de ingresar el dato por teclado y convertirlo a entero:

superficie=lado*lado

Para mostrar un mensaje por pantalla tenemos la función print que le pasamos como parámetro una cadena de caracteres a mostrar que debe estar entre simple o doble comillas:

print("La superficie del cuadrado es")

Para mostrar el contenido de la variable superficie no debemos encerrarla entre comillas cuando llamamos a la función print:

print(superficie)

Algunas consideraciones

Python es sensible a mayúsculas y minúsculas, no es lo mismo llamar a la función input con la sintaxis: Input.

Los nombres de variables también son sensibles a mayúsculas y minúsculas. Son dos variables distintas si en un lugar iniciamos a la variable "superficie" y luego hacemos referencia a "Superficie"

Los nombres de variable no pueden tener espacios en blanco, caracteres especiales y empezar con un número.

Todo el código debe escribirse en la misma columna, estará **incorrecto** si escribimos:

```
lado=input("Ingrese la medida del lado del cuadrado:")
  lado=int(lado)
  superficie=lado*lado
print("La superficie del cuadrado es")
print(superficie)
```

Forma correcta:

```
print(superficie)
lado=input("Ingrese la medida del lado del cuadrado:")
lado=int(lado)
superficie=lado*lado
print("La superficie del cuadrado es")
print(superficie)
```

Definición de comentarios en el código fuente

Un programa en Python puede definir además del algoritmo propiamente dicho una serie de comentarios en el código fuente que sirvan para aclarar los objetivos de ciertas partes del programa.
Tengamos en cuenta que un programa puede requerir mantenimiento del mismo en el futuro. Cuando hay que implementar cambios es bueno encontrar en el programa comentarios sobre el objetivo de las distintas partes del algoritmo, sobretodo si es complejo.
Existen dos formas de definir comentarios en Python:

Comentarios de una sola línea, se emplea el caracter #:

```
#definimos tres contadores
```

```
conta1=0
conta2=0
conta3=0
```

Todo lo que disponemos después del caracter # no se ejecuta

Comentarios de varias líneas:

```
"""Definimos tres contadores
   que se muestran si son distintos a cero"""
conta1=0
conta2=0
conta3=0
```

Se deben utilizar tres comillas simples o dobles seguidas al principio y al final del comentario.

Docstrings

Los docstrings son un tipo de comentarios especiales que se usan para documentar un módulo, función, clase o método. En realidad son la primera sentencia de cada uno de ellos y se encierran entre tres comillas simples o dobles.
Los docstrings son utilizados para generar la documentación de un programa. Además, suelen utilizarlos los entornos de desarrollo para mostrar la documentación al programador de forma fácil e intuitiva.
Veámoslo con un ejemplo:

```
def suma(a, b):
   """Esta función devuelve la suma de los parámetros a y b"
   return a + b
```

Más sobre variables

Hasta este momento hemos visto cómo definir variables enteras y flotantes. Realizar su carga por asignación y por teclado.

Para iniciarlas por asignación utilizamos el operador **=**

```
#definición de una variable entera
cantidad=20
#definición de una variable flotante
altura=1.92
```

Como vemos el intérprete de Python diferencia una variable flotante de una variable entera por la presencia del caracter punto.

Para realizar la carga por teclado utilizando la función input debemos llamar a la función int o float para convertir el dato devuelto por input:

```
cantidad=int(input("Ingresar la cantidad de personas:"))
altura=float(input("Ingresar la altura de la persona en metros ej:1.70:"))
```

Cadenas de caracteres

A estos dos tipos de datos fundamentales (int y float) se suma un tipo de dato muy utilizado que son las **cadenas de caracteres**.

Una cadena de caracteres está compuesta por uno o más caracteres. También podemos iniciar una cadena de caracteres por asignación o ingresarla por teclado.

Inicialización de una cadena por asignación:

```
#definición e inicio de una cadena de caracteres
dia="lunes"
```

Igual resultado obtenemos si utilizamos la **comilla simple**:

```
#definición e inicio de una cadena de caracteres
dia='lunes'
```

Para la carga por teclado de una cadena de caracteres utilizamos la función input que retorna una cadena de caracteres:

```
nombre=input("ingrese su nombre:")
```

Como su nombre lo indica una cadena de caracteres está formada generalmente por varios caracteres (de todos modos podría tener sólo un caracter o ser una cadena vacía)
Podemos acceder en forma individual a cada caracter del string mediante un subíndice:

```
nombre='juan'
print(nombre[0])   #se imprime una j
if nombre[0]=="j": #verificamos si el primer caracter del string es una j
    print(nombre)
    print("comienza con la letra j")
```

Los subíndices comienzan a numerarse a partir del cero.
Si queremos conocer la longitud de un string en Python disponemos de una función llamada **len** que retorna la cantidad de caracteres que contiene:

```
nombre='juan'
print(len(nombre))
```

El programa anterior imprime un 4 ya que la cadena nombre almacena 'juan' que tiene cuatro caracteres.

Métodos propios de las cadenas de caracteres.

Los string tienen una serie de métodos (funciones aplicables solo a los string) que nos facilitan la creación de nuestros programas.

Los primeros tres métodos que veremos se llaman: lower, upper y capitalize.

- **upper()** : devuelve una cadena de caracteres convertida todos sus caracteres a mayúsculas.
- **lower()** : devuelve una cadena de caracteres convertida todos sus caracteres a minúsculas.
- **capitalize()** : devuelve una cadena de caracteres convertida a mayúscula solo su primer caracter y todos los demás a minúsculas.

Concatenación

```
var1 = 'Hola'
var2 = 'Python'
var3 = var1 + ' ' + var2
```

Operadores de comparación

Los operadores de comparación se utilizan, como su nombre indica, para comparar dos o más valores. El resultado de estos operadores siempre es True o False.

Operador	Descripción
>	Mayor que. True si el operando de la izquierda es estrictamente mayor que el de la derecha; False en caso contrario.
>=	Mayor o igual que. True si el operando de la izquierda es mayor o igual que el de la derecha; False en caso contrario.
<	Menor que. True si el operando de la izquierda es estrictamente menor que el de la derecha; False en caso contrario.
<=	Menor o igual que. True si el operando de la izquierda es menor o igual que el de la derecha; False en caso contrario.

==	Igual. True si el operando de la izquierda es igual que el de la derecha; False en caso contrario.
!=	Distinto. True si los operandos son distintos; False en caso contrario.

Operadores aritméticos

Operador	Descripción
+	Suma dos operandos.
–	Resta al operando de la izquierda el valor del operando de la derecha. Utilizado sobre un único operando, le cambia el signo.
*	Producto/Multiplicación de dos operandos.
/	Divide el operando de la izquierda por el de la derecha (el resultado siempre es un float).
%	Operador modulo. Obtiene el resto de dividir el operando de la izquierda por el de la derecha.
//	Obtiene el cociente entero de dividir el operando de la izquierda por el de la derecha.
**	Potencia. El resultado es el operando de la izquierda elevado a la potencia del operando de la derecha.

Operadores lógicos

Operación	Resultado	Descripción
a or b	Si a se evalúa a falso, entonces devuelve b, si no devuelve a	Solo se evalúa el segundo operando si el primero es falso
a and b	Si a se evalúa a falso, entonces devuelve a, si no devuelve b	Solo se evalúa el segundo operando si el primero es verdadero
not a	Si a se evalúa a falso, entonces devuelve True, si no devuelve False	Tiene menos prioridad que otros operadores no booleanos

Ejemplos:

```
x = True
y = False
x or y #resultado True
x and y #resultado False
not x #resultadoFalse
```

Operadores de asignación

El operador de asignación se utiliza para asignar un valor a una variable. Como te he mencionado en otras secciones, este operador es el signo =.
Además del operador de asignación, existen otros operadores de asignación compuestos que realizan una operación básica sobre la variable a la que se le asigna el valor.
Por ejemplo, x += 1 es lo mismo que x = x + 1. Los operadores compuestos realizan la operación que hay antes del signo igual, tomando como operandos la propia variable y el valor a la derecha del signo igual.
A continuación, aparece la lista de todos los operadores de asignación compuestos:

Operador	Ejemplo	Equivalencia

+=	x += 2	x = x + 2
-=	x -= 2	x = x – 2
*=	x *= 2	x = x * 2
/=	x /= 2	x = x / 2
%=	x %= 2	x = x % 2
//=	x //= 2	x = x // 2
**=	x **= 2	x = x ** 2

Operadores de pertenencia

Los operadores de pertenencia se utilizan para comprobar si un valor o variable se encuentran en una secuencia (list, tuple, dict, set o str).
Todavía no hemos visto estos tipos, pero son operadores muy utilizados.

Operador	Descripción
in	Devuelve True si el valor se encuentra en una secuencia; False en caso contrario.
not in	Devuelve True si el valor no se encuentra en una secuencia; False en caso contrario.

A continuación vemos unos ejemplos que son muy intuitivos:
lista = [1, 3, 2, 7, 9, 8, 6]

```
4 in lista       #False
3 in lista       #True
4 not in lista #True
```

Listas

En Python existe un tipo de variable que permite almacenar una colección de datos y luego acceder por medio de un subíndice (similar a los string)

Creación de la lista por asignación

Para crear una lista por asignación debemos indicar sus elementos encerrados entre corchetes y separados por coma.

```
lista1=[10, 5, 3]                    # lista de enteros
lista2=[1.78, 2.66, 1.55, 89,4]        # lista de valores float
lista3=["lunes", "martes", "miercoles"] # lista de string
lista4=["juan", 45, 1.92]              # lista con elementos de distinto tipo
```

Si queremos conocer la cantidad de elementos de una lista podemos llamar a la función len:

```
lista1=[10, 5, 3]   # lista de enteros
print(len(lista1))  # imprime un 3
```

Una lista en Python es una estructura mutable (es decir puede ir cambiando durante la ejecución del programa)
Hemos visto que podemos definir una lista por asignación indicando entre corchetes los valores a almacenar:

lista=[10, 20, 40]

Una lista luego de definida podemos agregarle nuevos elementos a la colección. La primera forma que veremos para que nuestra lista crezca es utilizar el método **append** que tiene la lista y pasar como parámetro el nuevo elemento:

```
lista=[10, 20, 30]
print(len(lista))    # imprime un 3
lista.append(100)
print(len(lista))    # imprime un 4
print(lista[0])      # imprime un 10
print(lista[3])      # imprime un 100
```

Definimos una lista con tres elementos:

```
lista=[10, 20, 30]
```

Imprimimos la cantidad de elementos que tiene la lista, en nuestro caso lo definimos con 3:

print(len(lista)) # imprime un 3

Agregamos una nuevo elemento al final de la lista llamando al método append:

lista.append(100)

Si llamamos nuevamente a la función len y le pasamos el nombre de nuestra lista ahora retorna un 4:

print(len(lista)) # imprime un 4

Imprimimos ahora el primer y cuarto elemento de la lista (recordar que se numeran a partir de cero):
print(lista[0]) # imprime un 10
print(lista[3]) # imprime un 100

Lo que hace tan flexible a esta estructura de datos es que podemos almacenar componentes de tipo LISTA.

notas=[[4,5], [6,9], [7,3]]

En la línea anterior hemos definido una lista de tres elementos de tipo lista, el primer elemento de la lista es otra lista de dos elementos de tipo entero. De forma similar los otros dos elementos de la lista notas son listas de dos elementos de tipo entero.

Listas: eliminación de elementos

Hemos visto que una lista la podemos iniciar por asignación indicando sus elementos.
lista=[10, 20, 30, 40]

También podemos agregarle elementos al final mediante el método append:
lista.append(120)

Si ahora imprimimos la lista tenemos como resultado:
[10, 20, 30, 40, 120]

Otra característica fundamental de las listas en Python es que podemos eliminar cualquiera de sus componentes llamando al método **pop** e indicando la posición del elemento a borrar:
lista.pop(0)

Ahora si imprimimos la lista luego de eliminar el primer elemento el resultado es:
[20, 30, 40, 120]

Otra cosa que hay que hacer notar que cuando un elemento de la lista se elimina no queda una posición vacía, sino se desplazan todos los elementos de la derecha una posición.
El método pop retorna el valor almacenado en la lista en la posición indicada, aparte de borrarlo.

lista=[10, 20, 30, 40]
print(lista.pop(0)) # imprime un 10

Estructura de datos tipo Tupla

Hemos desarrollado gran cantidad de algoritmos empleando tipos de datos simples como enteros, flotantes, cadenas de caracteres y estructuras de datos tipo lista.
Vamos a ver otra estructura de datos llamada Tupla.
Una tupla permite almacenar una colección de datos no necesariamente del mismo tipo. Los datos de la tupla son inmutables a diferencia de las listas que son mutables.
Una vez inicializada la tupla no podemos agregar, borrar o modificar sus elementos.

La sintaxis para definir una tupla es indicar entre paréntesis sus valores:

```
tupla=(1, 2, 3)
fecha=(25, "Diciembre", 2016)
punto=(10, 2)
persona=("Rodriguez", "Pablo", 43)
print(tupla)
print(fecha)
print(punto)
print(persona)
```

Como vemos el lenguaje Python diferencia una tupla de una lista en el momento que la definimos:

```
tupla=(1, 2, 3)
fecha=(25, "Diciembre", 2016)
punto=(10, 2)
persona=("Rodriguez", "Pablo", 43)
```

Utilizamos paréntesis para agrupar los distintos elementos de la tupla.
Podemos acceder a los elementos de una tupla en forma similar a una lista por medio de un subíndice:

```
print(punto[0]) # primer elemento de la tupla
print(punto[1]) # segundo elemento de la tupla
```

Es muy IMPORTANTE tener en cuenta que los elementos de la tupla son **inmutables**, es incorrecto tratar de hacer esta asignación a un elemento de la tupla:

```
punto[0]=70
```

Nos genera el siguiente error:

```
Traceback (most recent call last):
  File "C:/programaspython/ejercicio146.py", line 11, in
    punto[0]=70
TypeError: 'tuple' object does not support item assignment
```

Utilizamos una tupla para agrupar datos que por su naturaleza **están relacionados** y que **no serán modificados** durante la ejecución del programa.

Empaquetado y desempaquetado de tuplas.

Podemos generar una tupla asignando a una variable un conjunto de variables o valores separados por coma:

```
x=10
y=30
punto=x,y
print(punto)
```

tenemos dos variables enteras x e y. Luego se genera una tupla llamada punto con dos elementos.

```
fecha=(25, "diciembre", 2016)
print(fecha)
dd,mm,aa=fecha
print("Dia",dd)
print("Mes",mm)
print("Año",aa)
```

El desempaquetado de la tupla "fecha" se produce cuando definimos tres variables separadas por coma y le asignamos una tupla:

```
dd,mm,aa=fecha
```

Es importante tener en cuenta definir el mismo número de variables que la cantidad de elementos de la tupla.

Tuplas anidadas

Ahora que vimos tuplas también podemos crear tuplas anidadas.
En general podemos crear y combinar tuplas con elementos de tipo lista y viceversa, es decir listas con componente tipo tupla.

```
empleado=["juan", 53, (25, 11, 1999)]
print(empleado)
empleado.append((1, 1, 2016))
print(empleado)

alumno=("pedro",[7, 9])
print(alumno)
alumno[1].append(10)
print(alumno)
```

Por ejemplo definimos la lista llamada **empleado** con tres elementos: en el primero almacenamos su nombre, en el segundo su edad y en el tercero la fecha de ingreso a

trabajar en la empresa (esta se trata de una tupla) Podemos más adelante durante la ejecución del programa agregar otro elemento a la lista con por ejemplo la fecha que se fue de la empresa:

```
empleado=["juan", 53, (25, 11, 1999)]
print(empleado)
empleado.append((1, 1, 2016))
print(empleado)
```

Tenemos definida la tupla llamada alumno con dos elementos, en el primero almacenamos su nombre y en el segundo una lista con las notas que ha obtenido hasta ahora:

```
alumno=("pedro",[7, 9])
print(alumno)
```

Podemos durante la ejecución del programa agregar una nueva nota a dicho alumno:

```
alumno[1].append(10)
print(alumno)
```

Porciones de listas, tuplas y cadenas de caracteres

El lenguaje Python nos facilita una sintaxis muy sencilla para recuperar un trozo de una lista, tupla o cadena de caracteres.
Veremos con una serie de ejemplos cómo podemos rescatar uno o varios elementos de las estructuras de datos mencionadas.

```
lista1=[0,1,2,3,4,5,6]
lista2=lista1[2:5]
print(lista2) # 2,3,4
lista3=lista1[1:3]
print(lista3) # 1,2
lista4=lista1[:3]
print(lista4) # 0,1,2
lista5=lista1[2:]
print(lista5) # 2,3,4,5,6
```

Para recuperar una "porción" o trozo de una lista debemos indicar en el subíndice dos valores separados por el caracter "**:**".

Del lado izquierdo indicamos a partir de qué elementos queremos recuperar y del lado derecho hasta cual posición sin incluir dicho valor.
Por ejemplo con la sintaxis:

```
lista1=[0,1,2,3,4,5,6]
lista2=lista1[2:5]
print(lista2) # 2,3,4
```

Estamos recuperando de la posición 2 hasta la 5 sin incluirla.

También es posible no indicar alguno de los dos valores:

lista4=lista1[:3]
print(lista4) # 0,1,2

Si no indicamos el primer valor estamos diciendo que queremos recuperar desde el principio de la lista hasta la posición menos uno indicada después de los dos puntos.
En cambio si no indicamos el valor después de los dos puntos se recupera hasta el final de la lista:

lista5=lista1[2:]
print(lista5) # 2,3,4,5,6

Hay que tener en cuenta que el concepto de "porciones" se puede aplicar en forma indistinta a listas, tuplas y cadenas de caracteres.

Valores negativos
Ahora veremos que podemos utilizar un valor negativo para acceder a un elemento de la estructura de datos.

```
lista1=[0,1,2,3,4,5,6]
print(lista1[-1]) # 6
print(lista1[-2]) # 5
```

En Python podemos acceder fácilmente al último elemento de la secuencia indicando un subíndice -1:
print(lista1[-1]) # 6

Luego el anteúltimo se accede con la sintaxis:
print(lista1[-2]) # 5

Diccionarios

Hasta ahora hemos presentado dos estructuras fundamentales de datos en Python: listas y tuplas. Ahora presentaremos y comenzaremos a utilizar la estructura de datos tipo diccionario.
La estructura de datos tipo diccionario utiliza una clave para acceder a un valor. El subíndice puede ser un entero, un float, un string, una tupla etc. (en general cualquier tipo de dato inmutable)

Podemos relacionarlo con conceptos que conocemos:

- Un diccionario tradicional que conocemos podemos utilizar un diccionario de Python para representarlo. La clave sería la palabra y el valor sería la definición de dicha palabra.
- Una agenda personal también la podemos representar como un diccionario. La fecha sería la clave y las actividades de dicha fecha sería el valor.

- Un conjunto de usuarios de un sitio web podemos almacenarlo en un diccionario. El nombre de usuario sería la clave y como valor podríamos almacenar su mail, clave, fechas de login etc.

Hay muchos problemas de la realidad que se pueden representar mediante un diccionario de Python.
Recordemos que las listas son mutables y las tuplas inmutables. Un diccionario es una estructura de datos mutable es decir podemos agregar elementos, modificar y borrar.

Definición de un diccionario por asignación.

```
productos={"manzanas":39, "peras":32, "lechuga":17}
print(productos)
```

Como vemos debemos encerrar entre llaves los elementos separados por coma. A cada elemento debemos indicar del lado izquierdo del caracter : la clave y al lado derecho el valor asignado para dicha clave. Por ejemplo para la clave "peras" tenemos asociado el valor entero 32.
Para agregar elementos a un diccionario procedemos a asignar el valor e indicamos como subíndice la clave:

```
    nombre=input("Ingrese el nombre del producto:")
    precio=int(input("Ingrese el precio:"))
    productos[nombre]=precio
```

Si ya existe el nombre del producto en el diccionario se modifica el valor para esa clave.

Operador in con diccionarios

Para consultar si una clave se encuentra en el diccionario podemos utilizar el operador **in**:

```
if clave in diccionario:
   print(diccionario[clave])
```

Si no existe la clave produce un error al tratar de accederlo:

```
print(diccionario[clave])
```