

Estructura y elementos del lenguaje



Contenido

Estructura y elementos del lenguaje	3
Palabras reservadas	3
Variables	4
Tipos de Datos	4
Expresiones y sentencias	6
Operadores	6
Operadores aritméticos	7
Operadores Lógicos	7
Palabras, frases y texto	8
Estructuras de datos	8
Listas	8
Tuplas	9
Diccionarios	10
Control de Flujo	10
Expresiones condicionales	11
If, elif y else	11
Bucles y repetición de código	12
While	12
For	13
Conclusión	14

Introducción

En el módulo anterior, realizamos una presentación de **Python** y sus características más importantes. Ahora, llegó el momento de comenzar a ver cómo está compuesto este lenguaje de programación y cuáles son las principales estructuras y elementos que nos ayudarán a desarrollar nuestros programas y *scripts* de la mejor manera.

Todo lenguaje de programación¹ cuenta con ciertas palabras clave (*keywords*), **tipos de datos** y una sintaxis que hay que tener en cuenta para poder utilizarlo. De igual manera que cualquier idioma tiene una sintaxis y una semántica y al escribir debemos respetarla, los lenguajes de programación requieren que sigamos ciertos lineamientos básicos.

Comenzaremos por ver los tipos básicos de datos en Python para luego repasar de qué manera utilizarlos al escribir nuestros programas e ir avanzando con este lenguaje de programación.

Estructura y elementos del lenguaje

Cuando en un lenguaje de programación nos referimos a la estructura del lenguaje, estamos hablando acerca de la manera de utilizar los tipos de datos, operadores, condicionales, bucles y demás. Antes de escribir código en Python vamos a repasar cada uno de estos elementos por separado para luego usarlos en conjunto.

Antes de empezar a ver los elementos de Python vamos a hablar acerca de características propias de todos los lenguajes de programación.

Palabras reservadas

Dentro de cada lenguaje de programación existen una serie de palabras claves (*keywords*) que hay que respetar y son propias de cada uno. En Python, por ejemplo ya vimos una palabra reservada „**print**“, que le dice al lenguaje que lo que viene a continuación lo tiene que imprimir. En nuestro ejemplo, imprimimos por pantalla un mensaje y la función de esta palabra reservada es más que clara.

Otras de las *keywords* de Python que son necesarias conocer son:

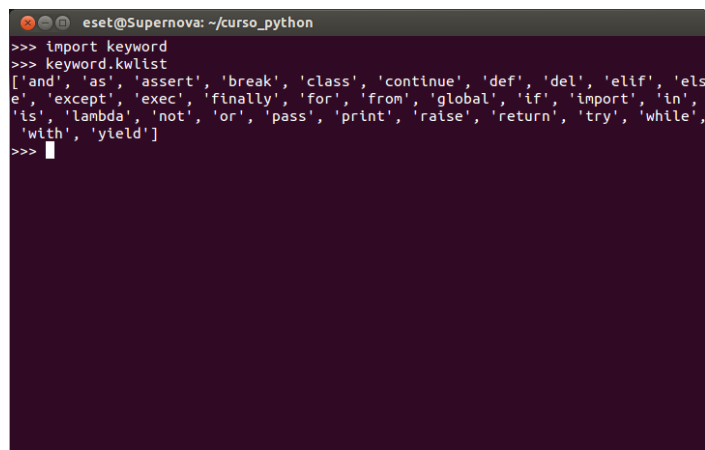
- *Import*
- *If*
- *For*
- *While*
- *Def*
- *Class*

Para poder ver un listado de todas las *keywords* de Python, pueden probar ejecutar el siguiente código en una consola y ver qué es lo que imprime:

```
>> import keyword
>> keyword.kwlist
```

Si lo prueban en una consola van a poder ver un listado como el siguiente:

¹ Lenguajes de Programación: http://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n

A screenshot of a terminal window with a dark background. The prompt is 'eset@Supernova: ~/curso_python'. The user has entered 'import keyword' and 'keyword.kwlist'. The output is a list of Python keywords in single quotes: ['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try', 'while', 'with', 'yield'].

```
eset@Supernova: ~/curso_python
>>> import keyword
>>> keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>>
```

Imagen 1 - Keywords en Python²

Variables

Uno de los elementos más importantes dentro de cualquier lenguaje de programación es el concepto de **variable**. ¿Qué es una variable? Según la Wikipedia³ una variable es un espacio en el sistema de almacenaje (memoria de la computadora) identificada por un nombre que nos permite guardar un valor o conjunto de valores.

Dicho en otras palabras, una **variable** nos permite identificar un valor dentro de nuestro programa que nosotros podemos utilizar cuando querramos cambiando su contenido, su tipo prácticamente en cualquier momento. Para identificar una variable es necesario definirle un nombre, en Python los nombres de las variables tienen que comenzar con una letra y luego pueden continuar con números, guiones (- o _).

Algunas convenciones son útiles al momento de programar y hacer que nuestro código sea más legible para otras personas, como así también para nosotros. Por ejemplo, se estila a que las variables en Python sean representadas en letra minúscula y si está formado por más de una palabra, el separador sea un „_“. Algunos nombres de variables pueden ser:

- a
- malware_familia
- numero
- archivo_de_entrada

Es posible encontrar una guía de estilo para Python⁴ desde la documentación oficial en dónde se explican las „mejores prácticas“ para los nombres de **variables**, **clases** y **métodos**. Una elección personal al momento de programar en Python es el lenguaje que vamos a utilizar para los nombres de variables, cada uno puede elegir si quiere utilizar nombres en español o inglés, queda a criterio persona.

Ahora que sabemos qué es una variable y tenemos una pequeña idea de para qué sirven vamos a ver los tipos de datos que tiene **Python** y cómo almacenarlos en nuestras variables.

Tipos de Datos

En Python existen algunos tipos de datos básicos que nos permiten representar números, cadenas de texto y demás. Los números son utilizados para cualquier tipo cálculo, las cadenas de texto nos permite expresarnos y mostrar información entre otras cosas. Vamos a ir creando algunas variables con distintos tipos de datos básicos:

```
>> a = 4
>> # Esto es un comentario
>> malware = "Win32/Dorkbot"
```

² Keywords en Python: <http://stackoverflow.com/questions/14595922/list-of-python-keywords>

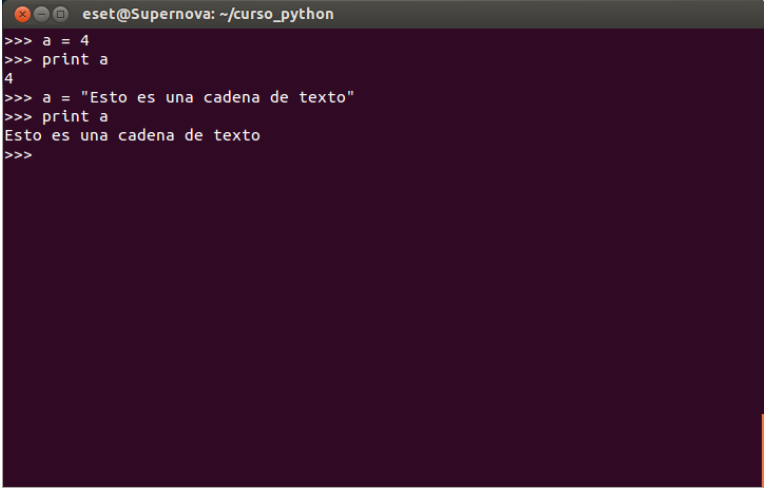
³ Variables de Programación: [http://es.wikipedia.org/wiki/Variable_\(programaci%C3%B3n\)](http://es.wikipedia.org/wiki/Variable_(programaci%C3%B3n))

⁴ Guía de estilo para Python: <http://www.python.org/dev/peps/pep-0008/>

```
>> b = 15.4
>> c = "Estoy aprendiendo Python"
```

En las líneas anteriores de código se declararon 2 variables numéricas, una entera y la otra real y dos cadenas de texto. Lo que vendría a ser los dos tipos de datos más básicos de cualquier lenguaje de programación. A diferencia de lenguajes como **C**, **Java** o **C#**, en Python no es necesario declarar ningún tipo de datos al crear una variable. Esto se debe a que tal y como comentábamos en el Módulo I, Python cuenta con tipado dinámico⁵.

En otras palabras cuando tenemos una variable, no es necesario especificar qué tipo de dato va a almacenar, como sí lo es en los lenguajes estáticamente tipados. A modo de ejemplo podemos ver las líneas de código en la siguiente imagen en donde se puede ver que la misma variable almacena dos tipos de datos diferentes:



```
eset@Supernova: ~/curso_python
>>> a = 4
>>> print a
4
>>> a = "Esto es una cadena de texto"
>>> print a
Esto es una cadena de texto
>>>
```

Imagen 2 - Tipos de variables en Python

En la primer línea se le asigna a la variable *a* el valor del número entero 4, luego se imprime el contenido en la siguiente línea. En el tercer comando que ingresamos, le asignamos a la variable *a* el texto “Esto es una cadena de texto”, por lo que la misma dejó de ser una variable de tipo numérica para pasar a ser una cadena de caracteres.

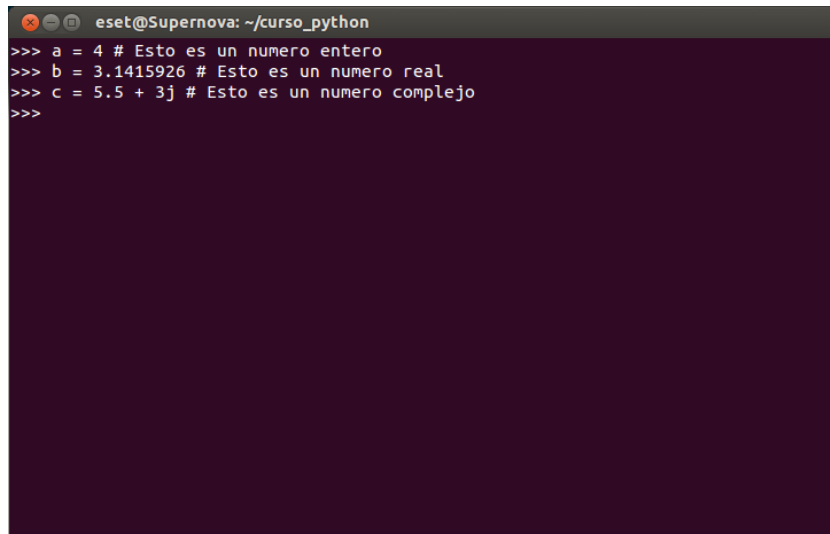
Como se pueden imaginar, este tipo de característica de los lenguajes interpretados como Python o Ruby, otorga a los programadores cierto dinamismo como así también algún que otro recaudo a tomar al momento de escribir su código. Entre los tipos de datos numéricos en Python podemos diferenciar tres tipos principales:

- **Enteros:** Los números enteros⁶ son el conjunto de números naturales tanto positivos como negativos, incluyendo el cero. En otras palabras no tiene parte decimal.
- **Reales:** La representación de los números reales en los lenguajes de programación se los conoce como Coma Flotante⁷, es una forma de notación científica y se pueden representar. Esto significa que los valores que podemos representar van desde $\pm 2,2250738585072020 \times 10^{-308}$ hasta $\pm 1,7976931348623157 \times 10^{308}$. De igual manera que vemos en el ejemplo un número real se compone de una parte entera, seguida por un punto y luego la parte decimal.
- **Complejos:** Los números complejos, tienen una parte real y una parte imaginaria. El motivo por el cual Python cuenta con números complejos se debe a que es ampliamente utilizado para cálculo numérico, ecuaciones diferenciales y otras actividades de cálculo complejos. Si bien no los vamos a utilizar a lo largo del curso, pueden ser un recurso de mucha utilidad llegado el caso.

⁵ Tipado dinámico: http://es.wikipedia.org/wiki/Tipado_din%C3%A1mico

⁶ Números enteros: http://es.wikipedia.org/wiki/N%C3%BAmero_entero

⁷ Coma flotante: http://es.wikipedia.org/wiki/Coma_flotante

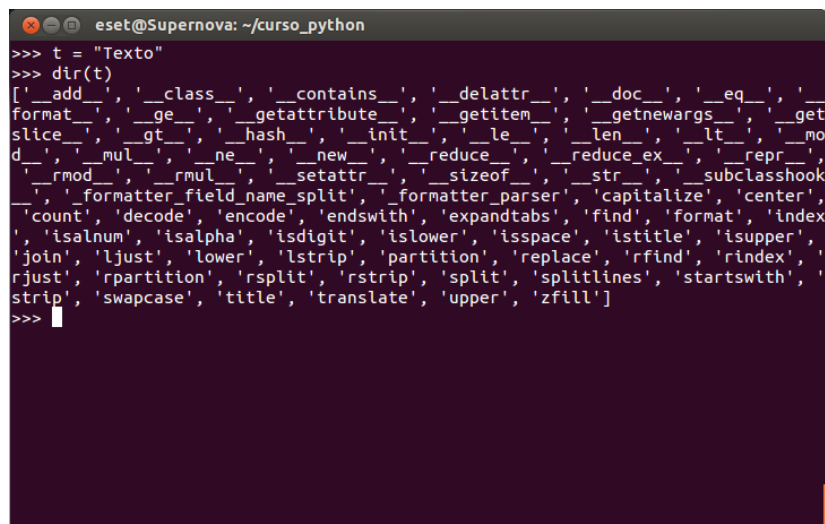
A terminal window with a dark background and light-colored text. The prompt is 'eset@Supernova: ~/curso_python'. The user has entered three lines of Python code: '>>> a = 4 # Esto es un numero entero', '>>> b = 3.1415926 # Esto es un numero real', and '>>> c = 5.5 + 3j # Esto es un numero complejo'. The prompt '>>>' is shown again on the next line.

```
eset@Supernova: ~/curso_python
>>> a = 4 # Esto es un numero entero
>>> b = 3.1415926 # Esto es un numero real
>>> c = 5.5 + 3j # Esto es un numero complejo
>>>
```

Imagen 3 - Tipos de variables numericas en Python

Las cadenas de texto también cuentan con una serie de funciones básicas que es necesario conocer, sobre todo para poder operar con ellas. Entre los métodos más comunes a conocer se encuentran algunos métodos que nos permiten saber si todos los caracteres son numéricos, caracteres, alfanumérico, pasarla a mayúsculas a minúsculas, etc.

En otras palabras, los tipos básicos de datos en Python incluyen una serie de métodos *built-in* para poder operar con ellos. Aprovechamos esta oportunidad para introducir un comando muy útil en Python; el comando **dir()**. Este comando nos permite listar todas las funciones que podemos ejecutar sobre una variable u objeto y su sintaxis es `dir(<variable>)`, por ejemplo en la siguiente captura utilizamos este método por una variable que contiene una cadena de texto:

A terminal window with a dark background and light-colored text. The prompt is 'eset@Supernova: ~/curso_python'. The user has entered two lines of Python code: '>>> t = "Texto"' and '>>> dir(t)'. The output of dir(t) is a long list of methods and attributes for a string object, including: ['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', 'format', '__ge__', '__getattr__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'formatter_field_name_split', 'formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill'].

```
eset@Supernova: ~/curso_python
>>> t = "Texto"
>>> dir(t)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', 'format', '__ge__', '__getattr__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'formatter_field_name_split', 'formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
```

Imagen 4 - Utilizando el comando dir()

En caso de que queramos leer la documentación más detallada también existe el comando `help()`. Los invitamos a probar cómo funciona y que listen la información de una cadena de texto.

Expresiones y sentencias

Ahora que ya conocemos los tipos básicos de datos en Python, comenzaremos a ver de qué manera podemos utilizarlos en un programa y cuáles son los operadores existentes para cada uno de ellos. En la presente sección veremos los operadores aritméticos, lógicos y otras funcionalidades del lenguaje.

Operadores

Operadores aritméticos

Los operadores aritméticos son aquellos que nos permite realizar operaciones por sobre los números. Son operaciones que utilizamos todos los días como la suma, la resta, la división y la multiplicación.

Descripción	Operador	Ejemplo
Suma	+	s = 4 + 9 # s es igual a 11
Resta	-	r = 15 - 3 # r es igual a 12
Multiplicación	*	m = 4 * 4 # m es igual a 16
División	/	d = 10 / 4 # d es igual a 2.5
División Entera	//	e = 5 // 2 # e es igual a 2
Exponente	**	p = 2 ** 5 # p es igual a 32
Módulo	%	z = 10 % 3 # z es igual a 1

Los operadores aritméticos no necesitan mucha explicación. En caso de que les haya quedado alguna duda sobre cómo funcionan o cuáles son los valores que retornan les recomendamos abrir una consola de Python y probar haciendo algunas cuentas. Además, de igual manera que sumamos y restamos números podemos realizar las mismas operaciones con variables:

```

eset@Supernova: ~/curso_python
>>> a = 4
>>> b = 5
>>> b * a
20
>>> a + b
9
>>> c = b / 2
>>> c
2
>>> 

```

Imagen 5 - Operadores aritméticos

Operadores Lógicos

Los operadores lógicos nos permiten evaluar una determinada condición. En otras palabras nos sirven para comprobar si se cumple una condición entre dos valores o variables, podemos preguntar si son iguales o no, y en el caso de los números evaluar si uno es mayor, menor o igual.

Descripción	Operador	Ejemplo
Menor	<	4 < 5 # True
Menor o Igual	<=	6 <= 4 # False
Igual	==	3 == (4 - 1) # True
Mayor o igual	>=	45 >= 34 # True
Mayor	>	11 > 11 # False

En una misma condición se pueden combinar uno o más operadores lógicos y en base al resultado de la condición ejecutar algún tipo de acción u otra. Más adelante en la sección de Control de Flujo veremos de qué manera tomar acciones diferentes según alguna condición lógica. Para poder combinar más de un operador para construir una condición compuesta existen otros operadores de conjunción y disyunción:

Descripción	Operador	Ejemplo
¿Se cumplen las dos condiciones?	and	a = True and False # a es False
¿Se cumple alguna de las dos condiciones?	or	a = False or True # a es True
Negación	not	a = not True # a es False

Como habrán observado, todos los resultados evaluados por algún operador lógico devuelven valores True o False, lo que se conoce como **booleanos**⁸. Los únicos dos valores posibles para una variable Booleana son **True** (Verdadero) o **False** (Falso) y como lo anunciamos son muy importantes para las expresiones condicionales y los bucles.

Palabras, frases y texto

El último de los tipos de datos que nos queda por ver un poco más en detalle son las cadenas de texto. En Python es posible definir una cadena de texto o conjunto de caracteres encerrando el texto entre comillas dobles ("string") o comillas simples ('string'). Por ejemplo:

```
>> a = "Esto es una cadena de texto"  
>> b = 'Esto es otra cadena de texto'
```

Además también dentro de una variable de texto en Python podemos incluir caracteres especiales, que son precedidos por una \ y nos permiten marcar el inicio de una nueva línea, tabulaciones, etc. También hay que tener en cuenta que las *strings* en Python cuentan métodos built-in que nos permiten buscar subcadenas de texto, reemplazar, o separar la cadena según un carácter y muchas otras opciones.

Intenten averiguar cómo hacer para declarar una variable que tenga más de una línea.

Estructuras de datos

En la sección anterior estuvimos viendo los tipos de dato básicos de Python como los números, las cadenas de texto y las variables booleanas. En Python también existen otro tipo de datos que nos permiten agrupar la información en diferentes conjuntos o colecciones.

Listas

Una Lista en Python es lo que en otros lenguajes de programación se conocen como vectores o arrays. Es un conjunto de datos del mismo o diferente tipo agrupados dentro de una misma estructura. Las listas son muy útiles para manejar información de una manera práctica y eficiente.

Para definir una lista en Python solo necesitamos indicar el grupo de variables a almacenarse dentro del array entre corchetes y separadas por comas:

```
>> mi_lista = [1, 25, "Troyano", "Gusano", True]
```

Una de las características más importantes de las Listas es que podemos acceder directamente a cualquiera de los elementos almacenados en ella a través del uso de índices. Cada una de las posiciones dentro de una Lista se referencia por un número empezando desde el 0. En otras palabras, si nosotros queremos acceder a la cadena "Troyano" podemos hacerlo directamente escribiendo **mi_lista[2]** desde la consola como se puede ver en la siguiente imagen:

⁸ Tipo de datos lógico: http://es.wikipedia.org/wiki/Tipo_de_dato_l%C3%B3gico


```
eset@Supernova: ~/curso_python
>>> mi_lista = [1,25, "Troyano", "Gusano", True]
>>> mi_lista[2]
'Troyano'
>>>
```

Imagen 6 - Listas en Python

A través del uso de los índices se puede acceder a los elementos de una lista, o como veremos más adelante, también es posible recorrer cada uno de sus elementos en un bucle.

Si queremos asociar a las listas con un concepto matemático, tenemos que relacionarlo con el concepto de vectores, además también podemos crear matrices de dos o más dimensiones y recorrer todos sus elementos. Si bien no nos vamos a centrar en esto, piensen para qué les podría servir poder representar matrices de distintas dimensiones.

Las Listas cuentan con varias funcionalidades para acceder a cada uno de sus elementos o una parte de ellos. Existe el concepto de slicing para definir cuántos elementos o desde qué posición se quieren extraer o leer. Para poder hacer esto, Python nos permite definir la posición de inicio y final. También, se pueden utilizar índices negativos para recorrer la lista de atrás para adelante. Algunos de estos ejemplos los pueden observar en la siguiente imagen:

```
eset@Supernova: ~/curso_python
>>> mi_lista = [1,25, "Troyano", "Gusano", True]
>>> mi_lista[1:3]
[25, 'Troyano']
>>> mi_lista[1:4]
[25, 'Troyano', 'Gusano']
>>> mi_lista[-2]
'Gusano'
>>>
```

Imagen 7 - Slicing Listas en Python

Los invitamos a probar las diferentes funciones y métodos que tienen las listas para ordenar, extraer y actualizar su contenido.

Tuplas

Las tuplas en Python son similares a las Listas, sin embargo, existe una diferencia importante en relación a las listas y refiere a que las Tuplas son inmutables. Una vez que se crearon no se pueden modificar su contenido y su tamaño es fijo. Para crear una tuplas lo elementos tiene que estar encerrados por paréntesis y los elementos separados por una coma:

```
>> mi_tupla = (1, "Malware")
>> mi_tupla[0]
```

1

Uno de los puntos a favor en relación a las Tuplas respecto a las Listas que es consumen menos memoria y por lo tanto son más rápidas que las listas, siempre teniendo en claro que su contenido no puede ser modificado.

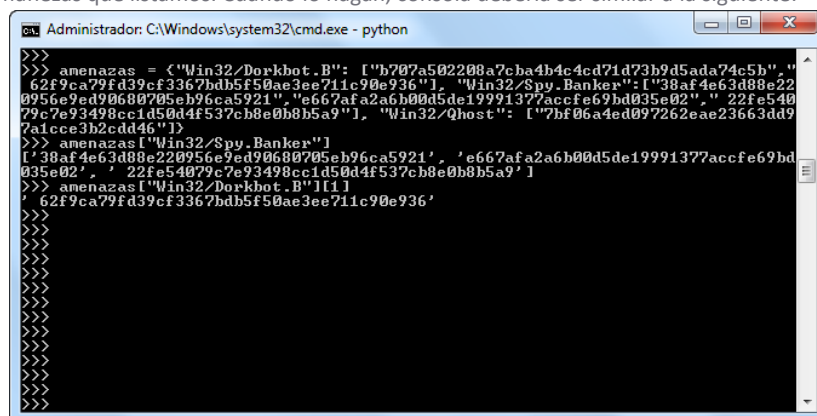
Diccionarios

Dentro de Python, los diccionarios son una de las estructuras de datos más avanzadas y con una gran utilidad. Se trata de una colección de elementos relacionados a través de una clave y un valor. Un diccionario nos permite organizar la información de la manera que nosotros queramos.

Con los diccionarios, podemos representar atributos de los elementos. Por ejemplo, podemos crear un diccionario que asocie distintos MD5 o SHA1 de un archivo con familias de códigos maliciosos. En este caso, nuestra clave o *key* sería el nombre de la detección y el valor un array que contenga los SHA1 de los archivos detectados:

```
>>> amenazas = {"Win32/Dorkbot.B": ["b707a502208a7cba4b4c4cd71d73b9d5ada74c5b",
62f9ca79fd39cf3367bdb5f50ae3ee711c90e936"],
"Win32/Spy.Banker": ["38af4e63d88e220956e9ed90680705eb96ca5921", "e667afa2a6b00d5de19991377accfe69bd035e02",
22fe54079c7e93498cc1d50d4f537cb8e0b8b5a9"], "Win32/Qhost": ["7bf06a4ed097262eae23663dd97a1cce3b2cdd46"]}
>>> amenazas["Win32/Spy.Banker"]
["38af4e63d88e220956e9ed90680705eb96ca5921", "e667afa2a6b00d5de19991377accfe69bd035e02",
22fe54079c7e93498cc1d50d4f537cb8e0b8b5a9"]
>>> amenazas["Win32/Dorkbot.B"][1]
"62f9ca79fd39cf3367bdb5f50ae3ee711c90e936"
```

Los invitamos a probar las líneas anteriores en una consola e intentar imprimir en pantalla los valores correspondientes a las distintas familias de amenazas que listamos. Cuando lo hagan, consola debería ser similar a la siguiente:



```
Administrador: C:\Windows\system32\cmd.exe - python
>>> amenazas = {"Win32/Dorkbot.B": ["b707a502208a7cba4b4c4cd71d73b9d5ada74c5b",
62f9ca79fd39cf3367bdb5f50ae3ee711c90e936"], "Win32/Spy.Banker": ["38af4e63d88e220956e9ed90680705eb96ca5921", "e667afa2a6b00d5de19991377accfe69bd035e02",
22fe54079c7e93498cc1d50d4f537cb8e0b8b5a9"], "Win32/Qhost": ["7bf06a4ed097262eae23663dd97a1cce3b2cdd46"]}
>>> amenazas["Win32/Spy.Banker"]
["38af4e63d88e220956e9ed90680705eb96ca5921", "e667afa2a6b00d5de19991377accfe69bd035e02",
22fe54079c7e93498cc1d50d4f537cb8e0b8b5a9"]
>>> amenazas["Win32/Dorkbot.B"][1]
62f9ca79fd39cf3367bdb5f50ae3ee711c90e936'
```

Imagen 8 - Diccionarios en Python

En este caso a diferencia de las anteriores, utilizamos una versión de Python en Windows, para demostrar que no hay diferencias. Los diccionarios puede ser muy útiles para darle forma a la información que queremos almacenar dentro de nuestro código. Entre los ejemplos más clásicos se ve el caso de películas o libros, pero en este caso, ver cómo se pueden armar estructuras útiles para categorizar archivos o amenazas también es útil. ¿Ustedes que creen?

Al igual que todas las otras estructuras que fuimos viendo hasta el momento, los diccionarios cuentan con una gran cantidad de métodos built-in que nos permiten manejar su información de una manera práctica y eficiente. A medida que vayamos avanzando con el curso vamos a ir

Control de Flujo

Al momento de programar es igual que en la realidad, debemos tomar elecciones. Basados en alguna condición será necesario realizar una u otra determinada acción. Para lograr esto en los lenguajes de programación se utilizan las sentencias condicionales que no solo nos permiten definir qué líneas de código se ejecutaran según el estado del programa sino que además sirve como condición de corte para los bucles, que veremos a continuación.

Expresiones condicionales

Las expresiones condicionales en los lenguajes de programación son una de las funcionalidades más importantes para el control de un programa. Nos permiten establecer cuándo se tomará una acción u otra según el estado de algunas variables.

Para poder utilizar y entender al máximo las expresiones condicionales tenemos que recordar lo que vimos con los **operadores lógicos** y las variables **booleanas**. Toda expresión condicional se evalúa como el resultado de una expresión booleana y eso determinará si se ejecuta una sección de código u otra. La expresión más conocida en prácticamente todos los lenguajes de programación para evaluar una condición es el **if**

If, elif y else

El **if** es la estructura condicional más simple que existe. Simplemente pregunta si se cumple una condición y en ese caso ejecuta o no alguna parte del código según el resultado. La sintaxis de un **if** en Python es la siguiente:

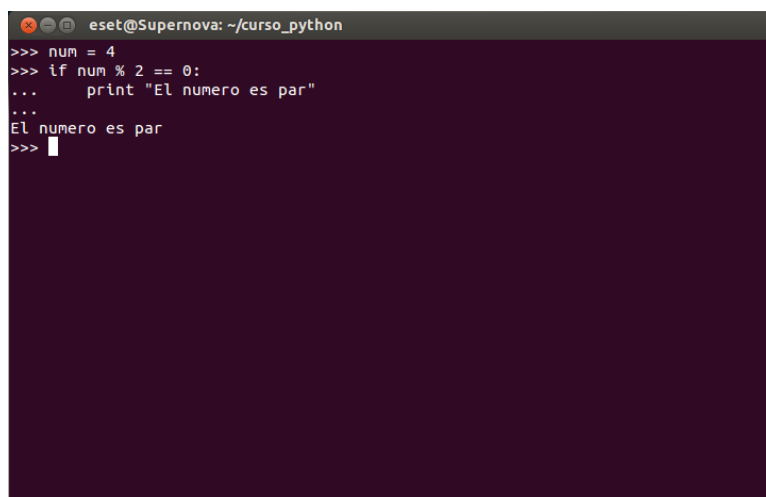
```
>>> if <condición>:  
    <código>
```

<condición> : Combinación de variables y operadores lógicos que tienen como resultado un booleano (*True* o *False*)

<código>: Instrucción a ejecutarse si se cumple la condición evaluada.

Un punto importante a remarcar y que es una de las características de Python, es que a diferencia de lenguajes como C, Java, C# o JavaScript, el código utiliza indentación (o tabulación) para marcar cuándo un conjunto de instrucciones se ejecuta dentro de un nivel. Puede que esto no quede muy claro al principio pero que Python utilice indentación para delimitar conjuntos de instrucciones hace que nuestro código sea más legible y ordenado.

Ahora nos vamos a detener con un clásico ejemplo y evaluar si un número es par o impar:



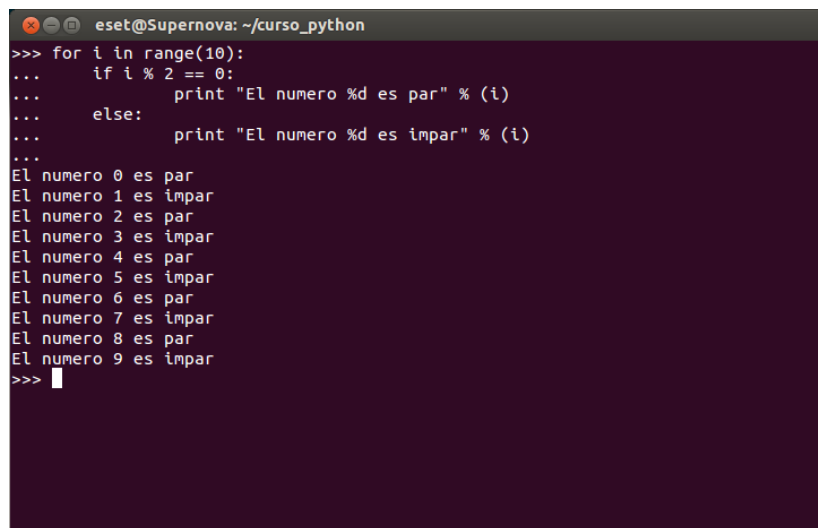
```
eset@Supernova: ~/curso_python  
>>> num = 4  
>>> if num % 2 == 0:  
...     print "El numero es par"  
...  
El numero es par  
>>>
```

Imagen 9 - Numero par o no

En las líneas de códigos anteriores se puede ver el resultado de ejecutar una sentencia condicional sobre el valor de la variable **num**. Dentro de la condición del **if**, se evalúa si el resto de la división de **num** por dos es igual a cero. Si esta condición se cumple entonces el resultado es verdadero (*true*) y por lo tanto se ejecuta la instrucción **print**.

Como podrán imaginar este código cambiará el flujo de la ejecución si y solo si se cumple la condición, en caso contrario lo hará nada. Llegado el caso de que se tenga que evaluar más de una condición se pueden anidar diferentes condiciones **if** a través del uso del **elif**, el “else if” de otros lenguajes de programación. El **else**, es muy útil cuando en caso de que no se haya cumplido ninguna condición, queremos ejecutar determinadas líneas de código.

Para ver un ejemplo con más condiciones, intenten ejecutar el siguiente script en una consola y evaluar su resultado:



```
eset@Supernova: ~/curso_python
>>> for i in range(10):
...     if i % 2 == 0:
...         print "El numero %d es par" % (i)
...     else:
...         print "El numero %d es impar" % (i)
...
El numero 0 es par
El numero 1 es impar
El numero 2 es par
El numero 3 es impar
El numero 4 es par
El numero 5 es impar
El numero 6 es par
El numero 7 es impar
El numero 8 es par
El numero 9 es impar
>>>
```

Imagen 10 - Números pares e impares con un if

¿Pudieron decifrar qué es lo que hacen las líneas de código de la imagen? ¿Qué otros ejemplos se les ocurren?

Cuando trabajamos con este tipo de expresiones condicionales es muy importante que estemos atentos a la indentación. Muchos de los IDE que puedan utilizar automáticamente indentarán el código luego de escribir el dos puntos(":"), sin embargo un error de este tipo puede causar más de un dolor de cabeza.

Bucles y repetición de código

Los bucles nos permiten ejecutar una determinada cantidad de veces una serie de instrucciones de código. En otras palabras, mientras se cumpla la condición que nosotros queremos se ejecutará el código y cuando se deje de cumplir, continuará con la ejecución de la siguiente parte del programa.

Las dos sentencias con las que cuenta Python para esto son el **while** y el **for**. La primera se ejecutará mientras la condición que nosotros dispongamos sea verdadera y la segunda ejecutará el código para uno y cada uno de los elementos por los cuales querramos iterar. Dicho en otras palabras, en un **for** queremos ejecutar una secuencia de instrucciones a una serie finita de elementos y en el caso del **while**, se van a **ejecutar las intrucciones mientras se cumple la condición**.

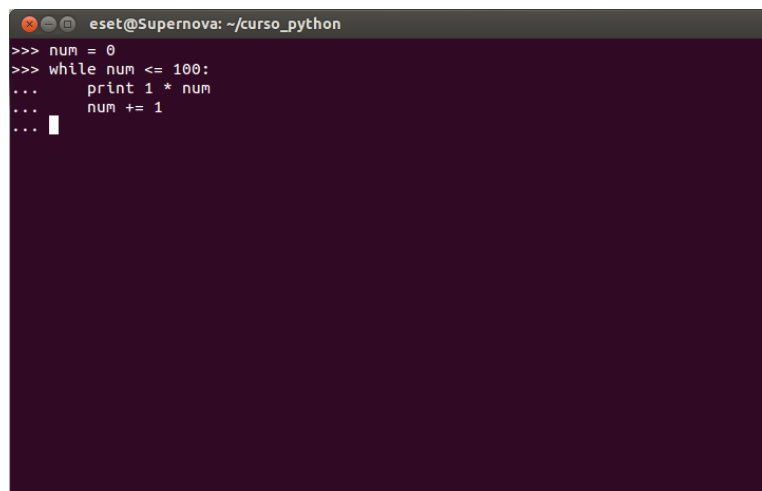
Es posible programar un **while** infinito, pero la cantidad de iteraciones de un **for** es finita.

While

Como definimos anteriormente, un **while** ejecuta un fragmento de código mientras se cumpla una condición. Su estructura es más que sencilla, lo ejemplificaremos con el siguiente código:

```
>> while <condición>:
    <código>
    <código>
    ...
    <código>
```

De esta manera podemos lograr que una sección de código se repita acorde a nuestras necesidades. Veamos un ejemplo sencillo para contar hasta 100:

A terminal window with a dark background and light text. The title bar reads 'eset@Supernova: ~/curso_python'. The code entered is: >>> num = 0, >>> while num <= 100:, ... print 1 * num, ... num += 1, ... The cursor is at the end of the third line.

```
eset@Supernova: ~/curso_python
>>> num = 0
>>> while num <= 100:
...     print 1 * num
...     num += 1
... 
```

Imagen 11 - Contando hasta 100 con un While

Al iniciarse, la variable **num** almacena el valor 0, luego irá iterando e imprimiendo por pantalla su valor para luego incrementar el valor en 1. De esta manera se ejecutarán esas dos líneas de código hasta que se deje de cumplir la condición o en otras palabras que num deje de ser menor o igual que 100.

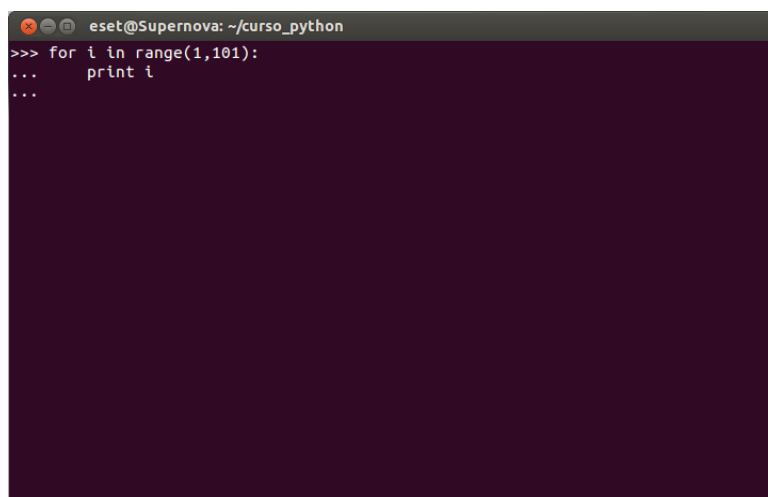
For

A diferencia del **while**, el for nos permite definir en lugar de una expresión condicional, un grupo o subgrupo de elementos a través del cual iterar. En otras palabras, se usa para repetir una secuencia de instrucciones una **i** cantidad de veces.

Su estructura general es la siguiente:

```
>>> for <variable> in [Conjunto de elementos]:
...     <código>
...     <código>
```

De igual manera que lo hicimos con el **while**, vamos un pequeño ejemplo para contar hasta 100, pero esta vez utilizando un **for**:

A terminal window with a dark background and light text. The title bar reads 'eset@Supernova: ~/curso_python'. The code entered is: >>> for i in range(1,101):, ... print i, ... The cursor is at the end of the third line.

```
eset@Supernova: ~/curso_python
>>> for i in range(1,101):
...     print i
... 
```

Imagen 12 - Contando hasta 10 con un for

En esta oportunidad, utilizamos un for para contar hasta 100. La principal diferencia es que en el caso del for no tenemos que ir sumando uno a la variable de corte. Ahora, en relación al range(1, 101), ¿qué creen ustedes que hace? ¿De qué manera se puede utilizar? Les recomendamos buscar un poco en internet para entender porqué pusimos el valor 101 para que corte en 100 y algunas otras consideraciones a tener en cuenta.

Conclusión

A lo largo del presente módulo se presentaron las variables, los tipos de datos, expresiones lógicas, condicionales, bucles y cómo controlar el flujo de la ejecución de nuestro programa. A grandes rasgos, estos son los elementos básicos de cualquier programa que vayamos a escribir y sus componentes centrales.

Con todo lo que vimos en este módulo, ustedes ya pueden comenzar a programar algunos algoritmos partes de programas o scripts que los ayuden a simplificar algunas tareas. Más adelante, veremos de qué manera leer archivos o acceder a sitios web para luego aplicar transformaciones que nos permitan explotar aún más las capacidades de Python.

A partir de ahora vamos a ir aplicando todos los puntos tratados aquí para crear objetos, funciones y scripts que nos van a permitir automatizar tareas, modificar archivos y muchas otras cosas más.