

Sumario

UT 03: Elementos de un programa informático.....	2
1 Introducción: ¿por qué debería aprender Python?.....	2
2 Elementos de un lenguaje de programación.....	4
3 Comenzando a programar en Python. La función "print()".....	5
4 Elementos de un programa informático.....	6
4.1 Literales y tipos de datos.....	6
4.2 Operadores aritméticos.....	10
4.3 Operadores de comparación.....	11
4.4 Operadores lógicos y de control.....	12
4.5 Resumen de operadores (del curso oficial de Cisco NetAcad).....	13
4.6 Variables.....	14
4.7 Entrada de datos.....	15
4.8 Constantes.....	16
4.9 Palabras reservadas.....	16
4.10 Reglas y convenciones de nombres.....	17
4.11 Sentencia del.....	17
4.12 Conversiones de tipo.....	18
4.13 Comentarios.....	19



Documentación oficial de Python: <https://docs.python.org/3/>

Referencia **w3schools**: https://www.w3schools.com/python/python_reference.asp



Realizado bajo licencia [Creative Commons Reconocimiento-NoComercial CC-BY-NC 4.0](https://creativecommons.org/licenses/by-nc/4.0/)

UT 03: Elementos de un programa informático

1 Introducción: ¿por qué debería aprender Python?

En esta Unidad de Trabajo vamos a introducir los conceptos de programación desde el enfoque de un lenguaje concreto, en este caso Python.

Python es un lenguaje omnipresente en multitud de dispositivos, aunque pueda pasar desapercibido. A lo largo de la vida de este lenguaje se han desarrollado miles de millones de líneas de código, lo que significa muchas oportunidades para la reutilización de código. Además cuenta con una comunidad muy activa, siempre dispuesta a ayudar.



También hay varios factores que hacen de Python un lenguaje excelente para aprender:

- Es fácil de aprender: el tiempo necesario para aprender Python es más corto que para muchos otros lenguajes; Esto significa que es posible iniciar la programación real más rápido;
- Es fácil y rápido generar código nuevo.
- Es fácil de obtener, instalar e implementar
- Es gratuito, abierto y multiplataforma; No todos los lenguajes pueden presumir de ello.

Si no conoces con otros lenguajes de programación, Python es ideal para comenzar, ya que permite adquirir una base sólida que te permitirá aprender otros lenguajes de programación (por ejemplo, C ++, Java o C) de manera mucho más fácil y rápida.

¿Para qué se usa realmente Python?

Python es el principal software de desarrollo utilizado en aplicaciones estadísticas, matemáticas y de diseño, pero también en utilidades de todo tipo:

- Juegos gratuitos de código abierto, por ejemplo, OpenRTS, PySol, Metin 2, Frets On Fire, juegos similares a Guitar Hero escritos en pygame.
- Portales web de servicios utilizan el lenguaje Python: Dropbox, UBER, Spotify, Pinterest, BitTorrent, etc...
- CAD / CAM 3D (FreeCAD, Fandango, Blender, Vintech RCAM)
- Aplicaciones empresariales (Odoo, Tryton, Picalo, LinOTP 2, RESTx)
- Aplicaciones de imagen (Gnofractal 4D, Gogh, imgSeek, MayaVi, VPython)
- Aplicaciones móviles (Aarlogic C05 / 3, AppBackup, Pyroute)

- Aplicaciones de oficina (calibre, caras, Notalon, pyspread)
- Gestores de información personal (BitPim, Narval, Priorizar, Coach de tareas, WikidPad)
- Desarrollo web (frameworks Django y Pyramid, microframes Flask y Bottle)
- Computación científica y numérica (p. Ej., SciPy, una colección de paquetes con fines matemáticos, científicos y de ingeniería; Ipython, un shell interactivo que presenta edición y grabación de sesiones de trabajo)
- Educación (¡es un lenguaje ideal para enseñar programación!)
- GUI de escritorio (por ejemplo, wxWidgets, Kivy, Qt)
- Desarrollo de software (control de construcción, gestión y pruebas: Scons, Buildbot, Apache Gump, Roundup, Trac)
- Aplicaciones empresariales (ERP, sistemas de comercio electrónico - Odoo, Tryton)

Y muchos, muchos otros proyectos y herramientas de desarrollo.

¿Cómo nació Python?

Python fue un proyecto personal de un programador holandés, llamado Guido van Rossum. En diciembre de 1989 escribió un intérprete para un lenguaje con ciertas características básicas:

- Un lenguaje fácil e intuitivo tan poderoso como el de los principales competidores.
- Código abierto, para que cualquiera pueda contribuir a su desarrollo.
- Código tan inteligible como el inglés simple.
- Adecuado para tareas cotidianas, que permita tiempos de desarrollo cortos.



Como el proyecto fue inicialmente un trabajo personal en su tiempo de ocio, Guido eligió el nombre Python para darle un aire informal, inspirado por el grupo británico de humor Monty Python. Casi treinta años después, está claro que sus objetivos se han cumplido. Algunos dicen que Python es el lenguaje de programación más popular del mundo.

Python 2 vs Python 3

En realidad hay dos versiones diferentes. Python 2 se mantiene (para dar cobertura a proyectos ya desarrollados), su evolución se detuvo. Las nuevas funcionalidades que necesitaba el lenguaje eran incompatible con la versión anterior. Python 3 no es una evolución, sino que tiene nuevas reglas y convenciones, siendo incompatible con el anterior, aunque formalmente es muy parecido. Los nuevos proyectos deberían desarrollarse en Python 3.

2 Elementos de un lenguaje de programación

Todo lenguaje de programación cuenta con los siguientes elementos:

1. Un **alfabeto** o conjunto de símbolos utilizados para construir palabras
2. Un **diccionario** o conjunto de palabras que el idioma ofrece a sus usuarios (por ejemplo, la palabra "computadora" proviene del diccionario de idioma inglés, mientras que "cmoptrue" no; la palabra "chat" está presente en los diccionarios de inglés y francés , pero sus significados son diferentes)
3. Una **sintaxis** o conjunto de reglas (formales o informales, escritas o sentidas intuitivamente) utilizadas para determinar si una determinada cadena de palabras forma una oración válida (por ejemplo, "Soy una pitón" es una frase sintácticamente correcta, mientras que "Pitón una yo soy" no lo es).
4. Una **semántica** o conjunto de reglas que determinan si cierta frase tiene sentido (por ejemplo, "comí una tarta" tiene sentido, pero "una tarta me comió" no)

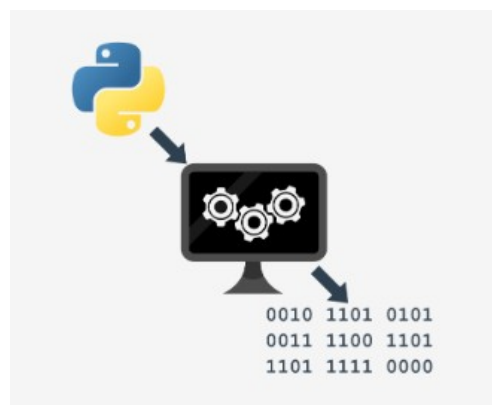
El IL (**Instruction List**) es el alfabeto de un lenguaje de máquina. Este es el conjunto de símbolos más simple y primario que podemos usar para dar instrucciones a un ordenador. Es la lengua materna de la computadora, pero es ininteligible para los humanos.

Necesitamos un lenguaje común para poder comunicarnos entre dos mundos tan diferentes. Es lo que llamamos **lenguajes de programación de alto nivel**. De alguna manera, se trata de lenguajes que usan símbolos, palabras y convenciones legibles para comandos que serán ejecutados por los ordenadores.

Los programas escritos en lenguajes de alto nivel se denominan "código fuente" y están contenidos en "**archivos fuente**". Estos archivos pueden ser ejecutados directamente si están escritos en lenguajes **interpretados**, o bien necesitan un proceso de conversión (**compilación**) en el caso de lenguajes **compilados**, como vimos en el capítulo anterior.

El **entorno de desarrollo** de un lenguaje de programación (por ejemplo, el entorno Spyder para Python3) requiere al menos tres elementos:

- Un **editor** que permite escribir el código, frecuentemente con asistencia contextual (por ejemplo, es frecuente contar con ayuda al pasar el ratón o al empezar a escribir comandos, así como acceder fácilmente a los manuales y librerías).
- Una **consola** que permita ejecutar el código recién escrito, pararlo y controlar su ejecución.
- Un **depurador** para lanzar el código paso a paso y poder inspeccionarlo en cada momento de la ejecución, así como ver el valor de las variables en directo.



3 Comenzando a programar en Python. La función "print()"

El lenguaje Python cuenta con palabras reservadas y funciones que se pueden importar desde diferentes librerías. Es importante tener en cuenta que en este lenguaje solo se puede escribir **una instrucción por línea**.

Como en todos los manuales de programación, comenzaremos con el típico programa "Hola, mundo", que tendrá solo una línea de código:

```
print("Hello, World!!")
```

En realidad, "print" es una **función** propia de Python. Como todas las funciones, tiene:

- Un **resultado** (lo que devuelve)
- Un **efecto** (lo que hace)
- **Argumentos** (lo que recibe entre paréntesis)

Sobrecarga de la función "print"

En Programación se conoce como "sobrecarga" a la posibilidad de que una función reciba un número variable de argumentos, comportándose de manera diferente en cada caso. En el caso de la función "print", tal como la hemos empleado, recibiría un solo argumento entre comillas ("Hello, World!!"), pero existe la posibilidad de utilizar otras implementaciones de la función cuando se utilizan argumentos especiales, como "end" (carácter de fin de línea) o "sep" (carácter separador). Ejemplos:

```
>>> print("Hello","World")
Hello World
>>> print("Hello","World",end="*")
Hello World*>>> print ("bye")
bye
>>> print("Hello","World",sep="- ")
Hello-World
>>> print("Hello","World",sep="- ",end="*")
Hello-World*>>>
>>> c='Esto es\nun string'
>>> print(c)
Esto es
un string
```

Las palabras reservadas (cadenas de texto que no pueden usarse como variables) son:

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

4 Elementos de un programa informático

4.1 Literales y tipos de datos

Los literales son las expresiones numéricas o alfanuméricas directas que se pueden utilizar dentro del código, usadas generalmente para realizar operaciones matemáticas o como parámetros de entrada de funciones. Hay diferentes formas de expresar estos valores, que están directamente relacionados con el tipo de datos que usa el programa:

Números

Los números enteros se pueden expresar directamente, sin comillas, con el carácter "-" en caso de valores negativos. También se puede usar la notación científica (por ejemplo, 3E4 sería 3×10^4), o indicar otras bases numéricas diferentes de la decimal, como octal (por ejemplo, 0o123) o hexadecimal (por ejemplo, 0x123).

```
>>> 0o10+1
9
>>> 0x10-1
15
```

Los números decimales pueden expresarse con un carácter "." separando la parte entera de la decimal.

```
>>> 4.
4.0
>>> 3E4
30000.0
>>> 6.62E-31
6.62e-31
```

Los principales tipos de datos numéricos son:

Clase	Tipo	Notas	Ejemplo
int	Números	Número entero con precisión fija.	42
long	Números	Número entero en caso de overflow.	42L ó 456966786151987643L
float	Números	Coma flotante de doble precisión.	3.1415927
complex	Números	Parte real y parte imaginaria j.	(4.5 + 3j)

Strings (cadenas alfanuméricas)

Las cadenas de texto se representan entre comillas. Se pueden usar comillas simples o dobles. Se pueden usar comillas simples dentro de dobles o viceversa. Para representar los caracteres especiales que podrían ser interpretados erróneamente, se usa el carácter escape (\). Existen secuencias especiales como tabulador (\t) o retorno de carro (\n).

También resulta muy interesante el uso de operadores matemáticos con cadenas de texto. Por ejemplo, el símbolo "+" significa "concatenación" y el símbolo "*" significa "repetición".

Vemos diferentes posibilidades en los siguientes ejemplos:

```
>>> print ("Hola")
Hola
>>> print ('Hola')
Hola
>>> print ("Digo: 'hola'")
Digo: 'hola'
>>> print ('Digo: "hola"')
Digo: "hola"
>>> print ("Digo: \"hola\"")
Digo: "hola"
>>> print ("pa"*2)
papa
>>> print ("pa"+"pa")
papa
```

Más ejemplos de uso del escape y cómo especificar a la función "print" que use el texto literalmente:

```
>>> print("C:\nombre\fichero")
C:
ombre
    ichero
>>> print("C:\\nombre\fichero")
C:\nombre
    ichero
>>> print("C:\\nombre\\fichero")
C:\nombre\fichero
>>> print(r"C:\nombre\fichero")
C:\nombre\fichero
```

Listas

Python permite utilizar unas variables especiales que contienen listas de elementos. Estos elementos pueden ser del mismo tipo o de tipos diferentes. Solo están identificados por la posición que ocupan, teniendo en cuenta que el primer elemento corresponde a la posición 0. Se pueden utilizar operadores de concatenación para añadir elementos:

```
>>> numeros=[1,2,3,4]
>>> numeros[3]
4
>>> numeros+[5,6]
[1, 2, 3, 4, 5, 6]
>>> numeros
[1, 2, 3, 4]
>>> numeros+=[5,6]
>>> numeros
[1, 2, 3, 4, 5, 6]
```


También se puede usar la función "append()":

```
numeros.append(7)
>>> numeros
[1, 2, 3, 4, 5, 6, 7]
```

Y acceder a rangos de valores:

```
>>> numeros[:2]
[1, 2]
>>> numeros[4:7]
[5, 6, 7]
>>> numeros[4:5]
[5]
```

Ejemplo de lista con datos de tipos diferentes:

```
>>> datos=[1,"hola",3.5]
>>> datos[2]
3.5
```

Si se utiliza un índice negativo, el elemento "-1" sería el último, "-2" sería el penúltimo y así sucesivamente:

```
>>> datos[-1]
3.5
>>> datos[-2]
'hola'
```

La función "len()" nos permite saber cuántos elementos hay en una lista:

```
>>> len(numeros)
7
```

Listas multidimensionales (matrices)

Python es un lenguaje muy potente para manejar conjuntos de datos. Las listas pueden agruparse a su vez en listas, formando conjuntos de datos que se pueden manipular fácilmente:

```
>>> a=[1,2,3]
>>> b=[4,5,6]
>>> c=[7,8,9]
>>> lista=[a,b,c]
>>> lista
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
>>> matriz=[
...     [1,1,1,10],
...     [2,2,2,2],
...     [3,3,3,3],
...     [4,4,4,4]
... ]
```



```
>>> matriz
[[1, 1, 1, 10], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]]
```

Veamos un ejemplo de manipulación para modificar el último elemento de cada lista (la última columna de la matriz) de manera que contenga un sumatorio del resto de elementos:

```
>>> matriz[0][-1]=sum(matriz[0][:-1])
>>> matriz[1][-1]=sum(matriz[1][:-1])
>>> matriz[2][-1]=sum(matriz[2][:-1])
>>> matriz[3][-1]=sum(matriz[3][:-1])
>>> matriz
[[1, 1, 1, 3], [2, 2, 2, 3], [3, 3, 3, 3], [4, 4, 4, 3]]
```

Strings como listas

Una variable de tipo String es considerada en Python una **lista** de caracteres, es decir, una variable de tipo Lista cuyo tamaño es el número de caracteres, en la que cada uno de ellos puede ser identificado por la posición que ocupa.

```
>>> palabra="Python"
>>> palabra[3]
'h'
```

Los strings son listas **inmutables**, es decir, no podemos modificar el valor de sus elementos. Pero sí podemos manipularlos como conjuntos de datos:

```
>>> palabra[1:3]
'yt'
>>> palabra[0]='N'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> palabra='N'+palabra[1:]
>>> palabra
'Nython'
```

Veamos un ejemplo de cómo voltear una secuencia de texto (teniendo en cuenta que se trata de una lista de caracteres):

```
>>> cadena="zeréP nauJ,01"
>>> cadena_volteada=cadena[::-1]
>>> cadena_volteada
'10,Juan Pérez'
```

Booleanos

Los valores booleanos son "False" o "True". Son el resultado de la evaluación de alguna operación de comparación. Hay que tener en cuenta que Python es "case sensitive", es decir, se diferencian mayúsculas y minúsculas. Los valores "false" o "true" no existen.

```
>>> 3>2
True
>>> false
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'false' is not defined
```

```
>>> condicion1=True
>>> condicion2=False
>>> condicion1 and condicion2
False
>>> condicion1 or condicion2
True
```

4.2 Operadores aritméticos

Las operaciones aritméticas pueden realizar conversiones de tipos. Si uno de los resultados es "float", el resultado de la operación será "float".

+	Suma
-	Resta
*	Multiplicación
/	División
//	División entera (realiza el redondeo truncando el resultado)
**	Potencia
%	Resto entero

Operadores reducidos (aplican una operación sencilla a una variable):

```
-=
+=
*=
/=
%=
**=
```

Ejemplos:

```
>>> 2**3
8
>>> 2.**3
8.0

>>> 8//3
2
>>> 14%4
2
>>> 12/4.5
2.6666666666666665
>>> 13%4
1
```

4.3 Operadores de comparación

Los operadores de comparación son:

==	Igual que
!=	Distinto
<	Menor
>	Mayor
<=	Menor o igual
>=	Mayor o igual

Ejemplos:

```
>>> print(5>4)
True
>>> print('x'=='y')
False
>>> x=3
>>> y=3
>>> print(x==y)
True
>>> print(x!=0)
True
>>> print (10=='10')
False
```



4.4 Operadores lógicos y de control

Los operadores **lógicos** "and" y "or" evalúan operaciones de izquierda a derecha, y retornan uno de los valores evaluados, NO un booleano.

El operador "not" devuelve el valor "NO".

and

- Si la expresión evaluada es verdadera, devuelve el último valor evaluado
- Si alguno de los valores es falso en contexto booleano, devuelve el primer valor falso

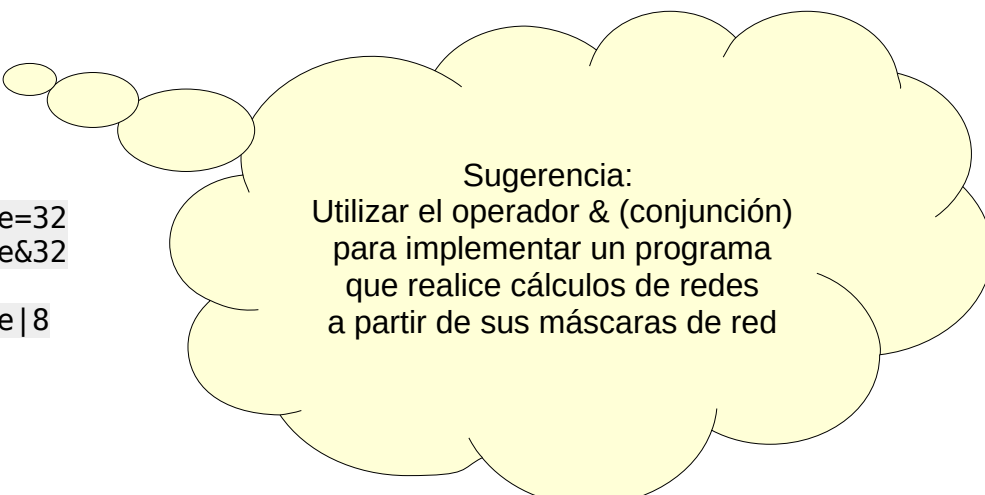
or

- Si la expresión evaluada es verdadera, devuelve el primer valor evaluado.
- Si alguno de los valores es verdadero en contexto booleano, devuelve ese valor inmediatamente y omite el resto (no lo evalúa).
- Si todos los valores son falsos, devuelve el último

Operadores de bit:

&	Conjunción
	Disyunción
~	Negación
^	xor
>>	Desplazamiento de bits a la derecha
<<	Desplazamiento de bits a la izquierda

```
>>> 1&1
1
>>> 1&2
0
>>> 32&8
0
>>> variable=32
>>> variable&32
32
>>> variable|8
40
>>> 1|1
1
>>> 2|2
2
>>> 1|2
3
```



Sugerencia:
Utilizar el operador & (conjunción)
para implementar un programa
que realice cálculos de redes
a partir de sus máscaras de red

4.5 Resumen de operadores (del curso oficial de Cisco NetAcad)

1. Python supports the following logical operators:

- **and** → if both operands are true, the condition is true, e.g., (True and True) is True,
- **or** → if any of the operands are true, the condition is true: (True or False) is True,
- **not** → returns false if the result is true, and returns true if the result is false.

2. You can use bitwise operators to manipulate single bits of data. Given the sample data:

- `x = 15` (000 1111 in binary)
- `y = 16`, (0001 0000 in binary)

will be used to illustrate the meaning of bitwise operators in Python. Analyze the examples:

- `&` does a bitwise and, e.g., `x & y = 0` (0000 0000 in binary)
- `|` does a bitwise or, e.g., `x | y = 31` (0001 1111 in binary)
- `~` does a bitwise not, e.g., `~ x = 240` (1111 0000 in binary)
- `^` does a bitwise xor, e.g., `x ^ y = 31` (0001 1111 in binary)
- `>>` bitwise right shift, e.g., `y >> 1 = 8` (0000 1000 in binary)
- `<<` bitwise left shift, e.g., `y << 3 = 128` (1000 0000 in binary)

Exercise 1: What is the output of the following snippet?

```
x = 1
y = 0
z = ((x == y) and (x == y)) or not(x == y)
print(not(z))
```

Resultado:

False

Exercise 2: What is the output of the following snippet?

```
x = 4
y = 1
a = x & y
b = x | y
c = ~x
d = x ^ 5
e = x >> 2
f = x << 2
print(a, b, c, d, e, f)
```

Resultado:

0 5 -5 1 1 16

4.6 Variables

Los nombres de variables:

- Pueden contener mayúsculas, minúsculas, dígitos y “_”
- Deben comenzar con una letra (se incluye “_”)
- Son sensibles a mayúsculas
- No pueden ser una palabra reservada
- Se permiten varios alfabetos
- Se crean en realidad cuando se les da un valor.
- No se deben usar antes de crearlas

```
>>> var=1
>>> print(var)
1
>>> print(Var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Var' is not defined
>>> Var=var+1
>>> print(Var)
2
>>> A=3
>>> B=4
>>> C=(A**2+B**2)**.5
>>> print("C=",C)
C= 5.0
```

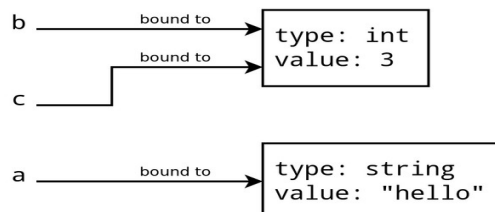
Para saber qué variables tenemos en uso en nuestro entorno podemos recurrir a la función "vars()":

```
>>> variable1=2
>>> variable2=3
>>> suma=variable1+variable2
>>> suma
5
>>> vars()
(...)
{'variable1': 2, 'variable2': 3, 'suma': 5}
```

Executed Code: Variable Assignment

```
a = 3
b = a
c = a
a = "hello"
```

Variables



Values in Memory

4.7 Entrada de datos

Para preguntar un dato al usuario se usa la función "`input()`", que solo puede tener un argumento y devuelve un dato de tipo "string":

```
>>> edad=input("Dime tu edad: ")
Dime tu edad: 38
>>> edad
'38'
>>> type(edad)
<class 'str'>
```

```
>>> edad=int(input("Dímela otra vez: "))
Dímela otra vez: 38
>>> edad
38
>>> type(edad)
<class 'int'>
```



4.8 Constantes

Una constante es un tipo de variable cuyo valor no puede modificarse. En realidad en Python no existen constantes, pero hay variables que se consideran constantes por convenio. Se declaran y asignan en módulos dedicados (archivos separados que se importan en nuestra aplicación). Por convenio, las constantes se nombran con letras mayúsculas; si su nombre contiene varias palabras, se separan con el guión bajo, "_".

Hay un pequeño número de constantes en el espacio de nombres por defecto: (que no cumplen la regla de denominarse con mayúsculas)

```
None
NotImplemented
Ellipsis
False
True
__debug__
```

Como ejemplo de uso de constantes, crear un archivo denominado "constantes.py" con este contenido:

```
IP_DB_SERVER = "127.0.0.1"
PORT_DB_SERVER = 3307
USER_DB_SERVER = "root"
PASSWORD_DB_SERVER = "123456"
DB_NAME = "nomina"
```

Y a continuación creamos el siguiente programa:

```
import constantes
print("scp -v -P {0} {1}@{2}:{3}/{4}/{4}.sql /srv/backup"
      .format(str(constantes.PORT_DB_SERVER),
              constantes.USER_DB_SERVER,
              constantes.IP_DB_SERVER,
              constantes.USER_DB_SERVER,
              constantes.DB_NAME))
```

Salida del programa:

```
scp -v -P 3307 root@127.0.0.1:/root/nomina/nomina.sql /srv/backup
```

4.9 Palabras reservadas

Las palabras reservadas no pueden asignarse como nombres de variables.

Para obtener una lista de todas las palabras reservadas

```
>>> 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']
```

4.10 Reglas y convenciones de nombres

Algunas reglas y convenciones de nombres para las variables y constantes:

- No usar símbolos especiales como !, @, #, \$, %, etc.
- El primer carácter no puede ser un número o dígito.
- Las constantes son colocadas dentro de módulos Python y significa que sus valores no pueden ser cambiados.
- Los nombres de constante y variable debería tener una combinación de letras en minúsculas (de a a la z) o MAYÚSCULAS (de la A a la Z) o dígitos (del 0 al 9) o un underscore (_). Por ejemplo: `snake_case`, `MACRO_CASE`, `camelCase`, `CapWords`
- Los nombres que comienzan con guión bajo (simple _ o doble __) se reservan para variables con significado especial
- No pueden usarse como identificadores las palabras reservadas .

4.11 Sentencia del

La sentencia `del` se define recursivamente de manera muy similar a la forma en que se define la asignación. A continuación unos ejemplos donde se inicializan variables:

```
>>> cadena, numero, lista = "Hola Plone", 123456, [7,8,9,0]
>>> tupla = (11, "Chao Plone", True, None)
>>> diccionario = {"nombre": "Leonardo", "apellido": "Caballero"}
```

Tras inicializar las variables del código anterior, se puede usar la función `"vars()"` para obtener un diccionario conteniendo ámbito actual de las variables, ejecutando:

```
>>> vars()
{'tupla': (11, 'Chao Plone', True, None),
 '__builtins__': <module '__builtin__' (built-in)>,
 'numero': 123456, '__package__': None, 'cadena': 'Hola Plone',
 'diccionario': {'apellido': 'Caballero', 'nombre': 'Leonardo'},
 '__name__': '__main__', 'lista': [7, 8, 9, 0], '__doc__': None}
```

Para eliminar la referencia a la variable `cadena`, ejecutamos:

```
>>> del cadena
```

```
>>> vars()
{'tupla': (11, 'Chao Plone', True, None),
 '__builtins__': <module '__builtin__' (built-in)>,
 'numero': 123456, '__package__': None,
 'diccionario': {'apellido': 'Caballero', 'nombre': 'Leonardo'},
 '__name__': '__main__', 'lista': [7, 8, 9, 0], '__doc__': None}
```

Al eliminar la referencia, la variable ya no está disponible.

Se puede eliminar una lista de valores recursivamente, de izquierda a derecha:

```
>>> del numero, lista, tupla, diccionario
```

```
>>> vars()
{'__builtins__': <module '__builtin__' (built-in)>,
 '__package__': None, '__name__': '__main__', '__doc__': None}
```

El eliminar las referencias se borra el enlace del nombre en el espacio de nombres locales o globales, dependiendo de si el nombre aparece en una sentencia “global” en el mismo bloque de código. Si el nombre no está vinculado, se generará una excepción “NameError”.

4.12 Conversiones de tipo

En Python no existe la necesidad de declarar las variables con un tipo de dato en específico, ya que las variables se procesan según el tipo de datos con el que se encuentren. Sin embargo, la conversión de tipo de datos se utiliza frecuentemente con la entrada de datos, es decir, cuando utilizamos la función 'input()'.

Python cuenta con la conversión de tipo de datos (comúnmente llamada cast o casting) para especificar explícitamente el tipo de dato con el que deseamos trabajar.

Para hacer castings en la entrada de datos, debemos colocar el tipo de dato que deseamos manejar (int, str, float, etc) y entre paréntesis la variable que va a ser convertida.

```
>>> var1 = 52
>>> float(var1)
52.0
>>> var2 = 152.02
>>> var3 = str(var2)
>>> var3
'152.02'
print var3
152.02
```

En el ejemplo anterior se declara la variable 'var1' y se le asigna el valor entero 52. En la línea dos el entero que contiene 'var1' se convierte a flotante con la instrucción 'float(var1)' y se muestra el resultado de la conversión. En la línea 4 se declara 'var2' con un valor de 152.02 (un flotante) para posteriormente ser convertido a una cadena con 'str(var2)' y esa cadena asignarla a 'var3'. El contenido de 'var3' es mostrado en la línea 7 entre comillas simples, esto se debe a que cuando omitimos la palabra print, se muestra el contenido de las variables pero especificando si son de tipo numérico o de cadenas de caracteres, en éste caso 'var3' es un string, por esta razón la salida es entre comillas simples. Sin embargo, cuando colocamos la palabra print, se muestran los datos de forma normal (sin comillas simples).

La función 'input()' espera que el usuario introduzca una expresión evaluable. Veamos algunas conversiones:

```
>>> var1=input('Numero: ')
Numero: 3*2+2
>>> var2=3
>>> var3=38.47
>>> var4=38.47/3

>>> print(var1,var2,var3,var4)
(8, 3, 38.47, 12.823333333333332)
```

4.13 Comentarios

Son cadenas de caracteres que se incluyen en el código como ayuda y son esenciales, tanto para quien está desarrollando el programa como para otras personas que lean el código.

Los comentarios en el código tienen una vital importancia en el desarrollo de todo programa, algunas de las funciones más importantes que pueden cumplir los comentarios en un programa, son:

- Brindar información general sobre el programa.
- Explicar qué hace cada una de sus partes.
- Aclarar y/o fundamentar el funcionamiento de un bloque específico de código, que no sea evidente de su propia lectura.
- Indicar cosas pendientes para agregar o mejorar.

El signo para indicar el comienzo de un comentario en Python es el carácter #, a partir del cual y hasta el fin de la línea, todo se considera un comentario, siendo ignorado por el intérprete Python.

El carácter # puede estar al comienzo de línea (en cuyo caso toda la línea será ignorada), o después de finalizar una instrucción válida de código.

```
>>> # Programa que calcula la sucesión
... # de números Fibonacci
...
>>> # se definen las variables
... a, b = 0, 1
>>> while b < 100: # mientras b sea menor a 100 itere
...     print b,
...     a, b = b, a + b # se calcula la sucesión Fibonacci
...
1 1 2 3 5 8 13 21 34 55 89
```

Comentarios multilínea

Python no dispone de un método para delimitar bloques de comentarios de varias líneas, pero se puede utilizar un truco, mediante "triples comillas". Esto tiene el inconveniente que, aunque Python no genera código ejecutable, el bloque delimitado no es ignorado por el intérprete Python, que crea el correspondiente objeto de tipo cadena de caracteres.

```
>>> """comentarios en varias líneas"""  
'comentarios en varias líneas'  
>>> '''comentarios en varias líneas'''  
'comentarios en varias líneas'
```

A continuación, una comparación entre comentarios multilínea y comentarios en solo una línea:

```
>>> # Calcula la sucesión  
... # de números Fibonacci  
...  
>>> """Calcula la sucesión  
... de números Fibonacci"""  
'Calcula la sucesión\nde números Fibonacci'
```

