

Introducción al lenguaje

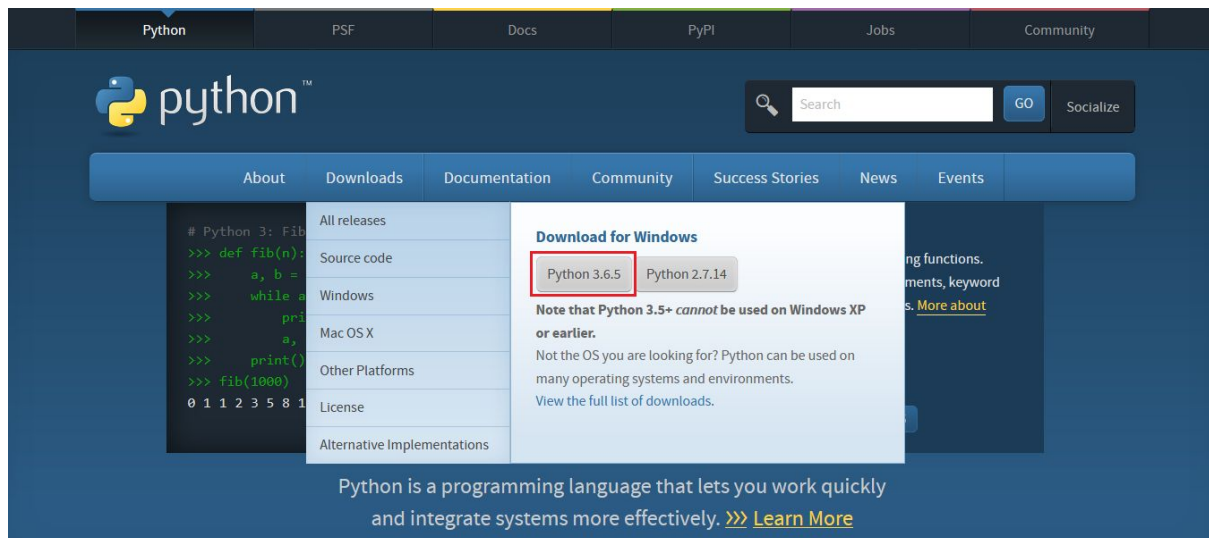
Aspectos básicos

Python es un lenguaje de programación creado en 1991 por el holandés Guido van Rossum. Es multiplataforma, ya que un mismo programa escrito en el lenguaje puede correr sin modificaciones en diversos sistemas operativos: Microsoft Windows, distribuciones de Linux y Mac OS. Es de código abierto, es decir, totalmente gratuito y cualquiera puede acceder a su código de fuente, modificarlo y distribuirlo libremente. Es multiparadigma pues soporta tres tipos de paradigmas: programación imperativa (típica en lenguajes tradicionales como C), orientada a objetos (el paradigma por antonomasia en lenguajes como Java) y funcional (Haskell, Elixir, entre otros). El modelo predominante es el de orientación a objetos.

Se trata de un lenguaje interpretado, de modo que un programa no es compilado y ejecutado directamente por el procesador, sino por el denominado *intérprete* (que no es más que un programa escrito en C). Por esta razón, a los programas de Python se los conoce en la jerga más bien como *scripts*.

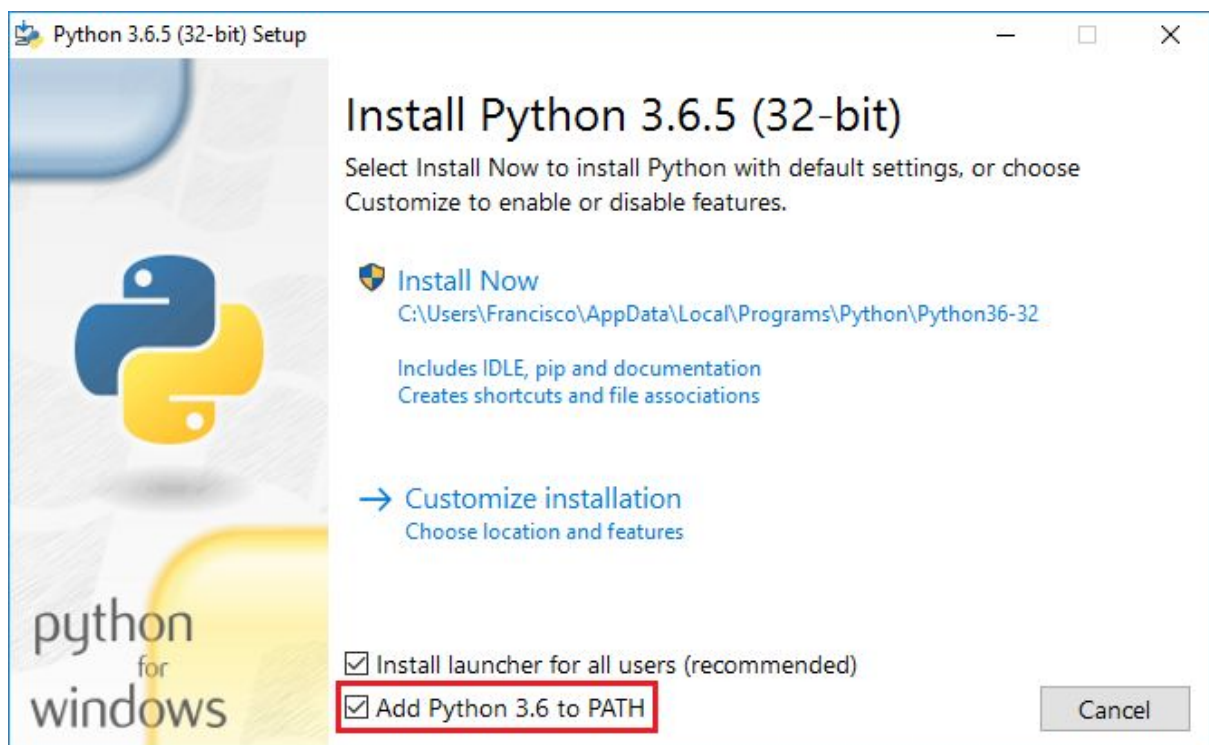
Python es ampliamente utilizado en la comunidad de programadores y el lenguaje con mayor proyección para los próximos años ya que, entre otras cosas, protagoniza el auge de la inteligencia artificial y el *machine learning*. Además es ideal para el desarrollo de aplicaciones web y de escritorio multiplataforma. Algunas de las grandes empresas que desarrollan con Python desde hace tiempo incluyen Google, YouTube, Netflix, Amazon, NASA, Instagram, entre otras.

Para descargar Python, nos dirigimos a la [página oficial \(python.org\)](https://python.org) y en la sección de descargas ("Downloads") seleccionamos "Python 3.x.x" (actualmente la última versión de Python 3 es la 3.7.3).



(Las versiones 2 y 3 de Python son incompatibles, es decir, existe cierta posibilidad de que un programa escrito para Python 2 no corra en Python 3 y viceversa. Python 2 dejará de recibir actualizaciones de seguridad en 2020, por lo tanto, se recomienda siempre utilizar la última versión estable de Python 3).

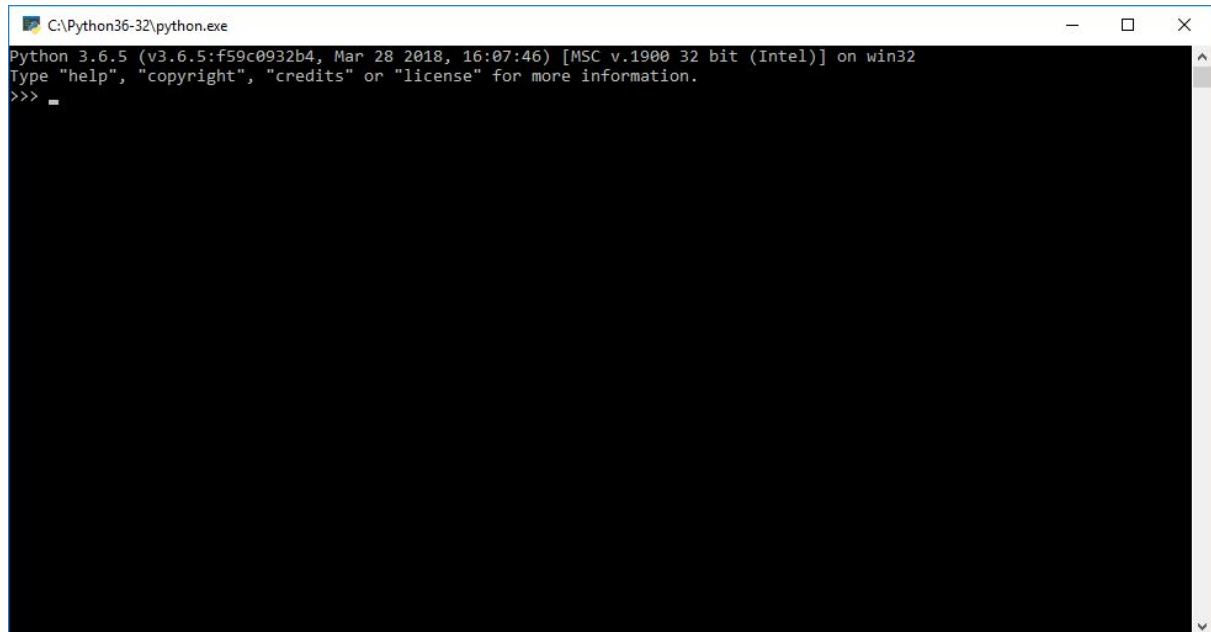
Una vez descargado el instalador para Windows, procedemos a ejecutarlo. La configuración por defecto del instalador es pertinente, a excepción de la opción “Add Python 3.x to PATH”, que inicialmente se encuentra desactivada, recomendamos activarla tal como se muestra en la siguiente imagen.



Luego proseguimos con la instalación hasta el final del asistente.

En Mac OS y distribuciones de Linux Python ya está instalado, pues muchas herramientas del sistema están escritas en el lenguaje.

Para verificar que Python se haya instalado correctamente abrimos la terminal (Inicio + R en Windows) y escribimos “python” (sin comillas).



Lo que vemos en pantalla se conoce con el nombre de consola interactiva. Allí podemos escribir código Python que se ejecutará instantáneamente al presionar la tecla Enter. Resulta una herramienta sumamente útil para realizar pruebas y desarrollar prototipos rápidos. Volveremos sobre esto más adelante.

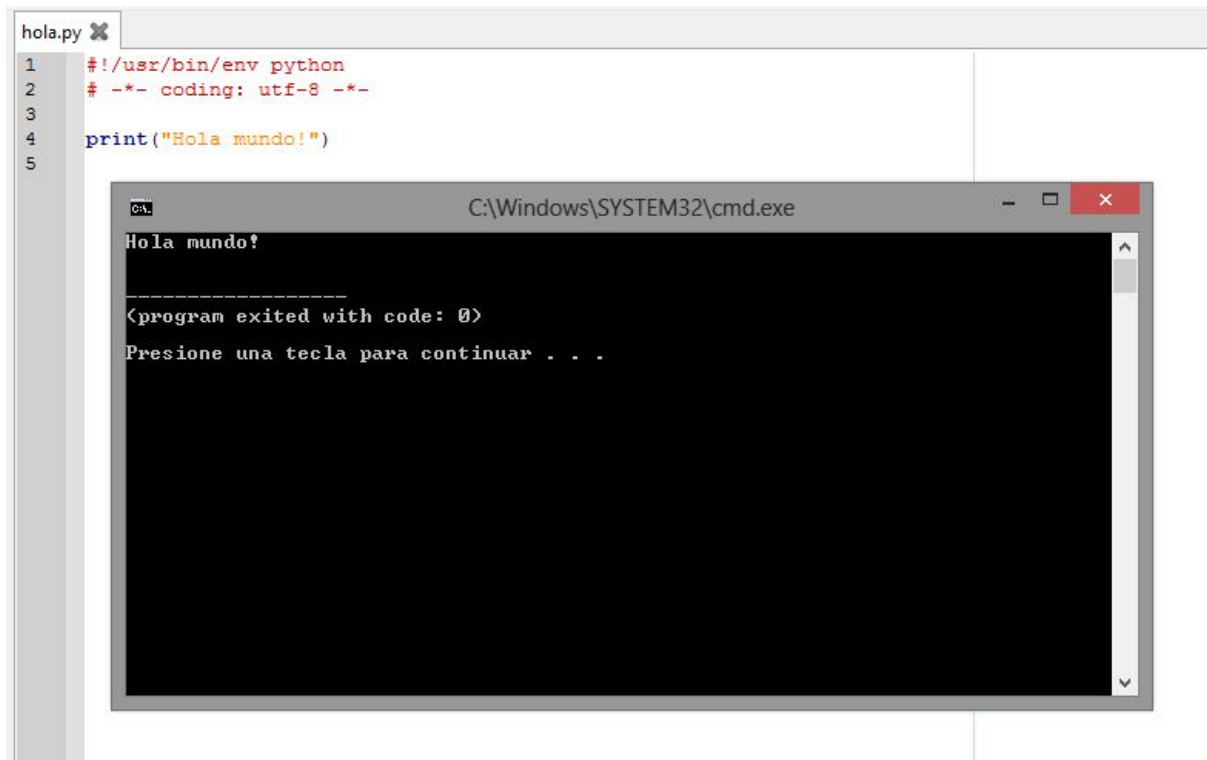
Python se distribuye con un editor de código llamado IDLE, bastante primitivo, que podrás encontrar en la carpeta de instalación. Además de un editor, incluye una consola interactiva como la anterior con resaltado de sintaxis y completado automático. Existen infinidad de editores de código para Python. En este curso haremos uso de uno de ellos llamado Geany, con funciones básicas pero suficiente para los propósitos del curso ya que es rápido, liviano y multiplataforma. Podés descargarlo desde [esta sección de la página oficial \(geany.org\)](http://geany.org). Otros IDEs más completos para el lenguaje incluyen Visual Studio, Visual Studio Code, Atom, PyCharm, Wing IDE.

Una vez instalado Python y el editor de código, ya tenemos el escenario listo para crear nuestro primer programa o *script*, el clásico “Hola mundo”. En Geany, creamos un nuevo archivo y dentro de él colocamos lo siguiente:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

print("Hola mundo")
```

Lo guardamos en cualquier ubicación con el nombre de “hola.py” y luego presionamos “Ejecutar” (o bien la tecla F5). Deberíamos ver algo como lo siguiente:



```
hola.py X
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  print("Hola mundo!")
5

C:\Windows\SYSTEM32\cmd.exe
Hola mundo?

-----
<program exited with code: 0>
Presione una tecla para continuar . . .
```

Nuestro programa simplemente imprimió en pantalla el mensaje “Hola mundo” y luego finalizó (un *script* termina cuando no hay nada más por ejecutar). `print()` es una función incorporada del lenguaje, que toma cualquier objeto y lo envía a la salida estándar; lo veremos con mayor detalle más adelante. Por el momento ignora las dos primeras líneas que, dicho sea de paso, son comentarios por empezar con el carácter # (numeral).

Tipos de datos

Tipos básicos

Python es un lenguaje de “tipado” dinámico. Esto quiere decir que las variables, como se las conoce en otros lenguajes, no son una suerte de cajas que aceptan únicamente un tipo de datos, sino más bien etiquetas que se le asignan a un objeto de cualquier tipo. No es necesario ahondar en cómo el intérprete maneja internamente las variables, simplemente tener en cuenta que no están ligadas a un tipo de dato en específico, y que acostumbramos a llamarlas “objetos” (ya veremos por qué).

Comencemos por abrir la consola interactiva de Python y escribir lo siguiente.

```
>>> a = 1
```

(A partir de ahora, siempre que veas los caracteres ">>>" da por sentado que estamos trabajando sobre la consola interactiva).

Hemos creado un objeto que contiene el número 1. Escribiendo el nombre de un objeto en la consola interactiva obtenemos su valor:

```
>>> a
1
```

Como los objetos no están ligados a un tipo específico de datos, ahora podemos reemplazar el contenido de `a` por una cadena:

```
>>> a = "Hola mundo"
```

(En Python las cadenas se construyen con comillas dobles o simples. Yo acostumbro a usar comillas dobles).

Ya hemos visto dos tipos de datos: un número entero (`int`) y una cadena (`str`). Continuemos creando un número de coma flotante y un booleano.

```
>>> b = True
>>> pi = 3.14
```

Un objeto booleano puede contener los valores `True` o `False`.

Para determinar si dos objetos tienen el mismo valor, utilizamos doble signo de igual.

```
>>> a = 1
>>> a == 1
True
>>> b = 2
>>> a == b
False
```

Y para determinar si su valor es desigual:

```
>>> a != b
True
>>> a != 1
False
```

Para valores numéricos, otros signos de comparación incluyen `>` (mayor), `<` (menor), `>=` (mayor o igual) y `<=` (menor o igual).

En Python la comparación siempre se realiza según el valor de un objeto independientemente del tipo. Por ejemplo, `a = 3` y `b = 3.0` son objetos diferentes (`a` es un entero y `b` es un número de coma flotante), no obstante tienen el mismo valor:

```
>>> a = 3
>>> b = 3.0
>>> a == b
True
```

Nótese que tanto los números enteros como los de coma flotante pueden también ser menores a cero:

```
>>> a = -5
>>> b = -10.35
```

Podemos conocer el tipo de un objeto usando la función `type()`:

```
>>> type(a)
<type 'int'>
>>> type(b)
<type 'float'>
```

Ahora bien, para objetos de tipos numéricos, las operaciones aritméticas se realizan vía `+`, `-`, `*` y `/`.

```
>>> 7 + 5
12
>>> 8 - 2
6
>>> 7 * 3
21
>>> 10 / 2
5
```

La operación conocida como módulo (no confundir con valor absoluto), que retorna el resto de una división, se realiza con el operador `%`.

```
>>> 3 % 2
1
>>> 10 % 5
0
```

Colecciones

Las colecciones son objetos que pueden almacenar otros objetos. Por ejemplo, utilizando corchetes creamos una lista, con números del 1 al 4.

```
>>> a = [1, 2, 3, 4]
```

Las listas son objetos mutables (es decir, su contenido puede variar) y ordenados (en otros lenguajes se las conoce por vectores o *arrays*). Por esta razón podemos acceder a sus elementos a partir de su posición, indicando un índice entre corchetes, comenzando desde el 0.

```
>>> a[0]
1
>>> a[1]
2
>>> a[2]
3
>>> a[3]
4
```

Los índices pueden ser negativos, para indicar que se debe empezar a contar desde el último elemento:

```
>>> a[-1]
4
>>> a[-2]
3
```

Los elementos dentro de una lista no necesariamente tienen que ser del mismo tipo. Incluso pueden contener otras listas.

```
>>> b = [3.14, True, ["Hola mundo", False]]
>>> b[0]
3.14
>>> b[1]
True
>>> b[2]
['Hola mundo', False]
>>> b[2][0]
'Hola mundo'
>>> b[2][1]
False
```

Como las listas son objetos mutables, podemos cambiar el objeto en una posición determinada con una simple asignación.

```
>>> a = [1, 2, 3, 4]
```

```
>>> a[2] = "Hola mundo"
>>> a
[1, 2, 'Hola mundo', 4]
```

Para añadir un elemento al final de una lista, utilizamos la función `append()`.

```
>>> a = [1, 2, 3, 4]
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
```

Y para insertar un elemento en una posición específica, la función `insert()` toma como primer argumento un índice de base 0 seguido del objeto a insertar.

```
>>> a.insert(0, -1)
>>> a
[-1, 1, 2, 3, 4, 5]
```

Por último, vía la palabra reservada `del` procedemos a remover un elemento.

```
>>> a = [1, 2, 3, 4]
>>> del a[2]
>>> a
[1, 2, 4]
```

Las tuplas son similares a las listas, pero son objetos inmutables. Una vez creada una tupla, no pueden añadirse ni removerse elementos.

```
>>> a = (1, 2, 3, 4)
```

Se crean indicando sus elementos entre paréntesis (aunque también pueden omitirse) y, al igual que las listas, pueden contener objetos de distintos tipos, incluso otras tuplas. Se accede a sus elementos indicando su posición entre corchetes.

Es importante aclarar que para crear una tupla con un único elemento, se debe usar (nótese la coma):

```
>>> a = (1,)
```

Ya que los paréntesis son también utilizados para agrupar expresiones y, de lo contrario, Python no podría distinguir si se trata de una expresión o de una tupla con un único elemento.

Siempre que requieras de un conjunto ordenado de objetos, pensá si luego su contenido va a ser modificado. En caso afirmativo, creá una lista. De lo contrario, una tupla.

Dos listas o dos tuplas pueden concatenarse utilizando el operador de suma.

```
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
```

El último tipo de colección que veremos se llama diccionario. Es lo que en otros lenguajes se conoce como vector o lista asociativa. Está constituido por pares de una clave y un valor.

```
>>> d = {"a": 1, "b": 2}
```

En este caso, las claves son las cadenas "a" y "b", que están asociadas a los números 1 y 2, respectivamente. Así, accedemos a los valores a partir de su respectiva clave.

```
>>> d["a"]
1
>>> d["b"]
2
```

Los valores en un diccionario pueden ser de cualquier tipo, también otros diccionarios. Las claves solo pueden ser objetos inmutables (por ejemplo, cadenas, enteros, números de coma flotante, tuplas; no así listas) y no pueden repetirse dentro de un mismo diccionario.

```
>>> d = {123: "Hola, mundo", True: 3.14, (1, 2): False}
>>> d[123]
'Hola, mundo'
>>> d[True]
3.14
>>> d[(1, 2)]
False
```

Los pares clave-valor de un diccionario no pueden ser accedidos a partir de un índice porque no es una colección ordenada.

Para cambiar el valor de una clave mantenemos la misma sintaxis que en las listas:

```
>>> d = {"a": 1, "b": 2}
>>> d["b"] = 3.14
>>> d
{'a': 1, 'b': 3.14}
```

Lo cual también sirve para añadir elementos:

```
>>> d["c"] = "Hola mundo"
>>> d
```

```
{'a': 1, 'c': 'Hola mundo', 'b': 3.14}
```

Y para remover un par clave-valor:

```
>>> del d["b"]
>>> d
{'a': 1, 'c': 'Hola mundo'}
```

Para conocer la cantidad de elementos de una colección utilizamos la función `len()`.

```
>>> len([1, 2, 3])
3
>>> len((True, False))
2
>>> len({"a": 1})
1
>>> len("Hola mundo")
10
```

Otros tipos

Cuando queremos crear un objeto pero por el momento no asignarle ningún valor, generalmente se utiliza `None` (en inglés, literalmente, “nada”).

```
>>> a = None
```

Para determinar si un objeto equivale a `None`, utilizamos la palabra reservada `is` en lugar del signo de comparación (`==`).

```
>>> a is None
>>> True
```

Control de flujo

Python reserva las palabras `and` y `or` para las operaciones lógicas de conjunción y disyunción, respectivamente. Ambas operaciones siempre retornan `True` o `False`.

```
>>> True and True
True
>>> False and False
False
>>> True and False
False
>>> True or True
```

```
True
>>> False or False
False
>>> True or False
True
```

Una conjunción equivale a `True` cuando los dos objetos a comparar equivalen a `True`; caso contrario, el resultado es `False`. Una disyunción equivale a `True` cuando uno de los dos objetos equivale a `True`; caso contrario, el resultado es `False`. Dicho así suena un tanto abstracto pero se te irá aclarando a medida que avances con el lenguaje. De hecho no se trata de operaciones estrictamente de la programación sino más bien de la lógica. Para más información véase [Conjunción lógica](#) y [Disyunción lógica](#).

De forma similar, la palabra `not` equivale a la [Negación lógica](#).

```
>>> not True
False
>>> not False
True
```

Condicionales

Al igual que en cualquier lenguaje imperativo, en Python puede controlarse el flujo de un programa de acuerdo a una o varias condiciones vía `if`. (A partir de ahora, considera crear un archivo de Python en Geany para probar las porciones de código).

```
edad = 30
if edad > 18:
    print("Sos mayor de edad.")
else:
    print("No sos mayor de edad.")
```

Para añadir más condiciones, utilizamos `elif`:

```
edad = 70
if edad > 18 and edad < 65:
    print("Sos mayor de edad.")
elif edad >= 65:
    print("Sos un adulto mayor.")
else:
    print("No sos mayor de edad.")
```

Cuando ninguna de las condiciones se cumple (es decir, todas retornan `False`), se ejecuta el bloque de código luego de `else`.

Nótese que en Python se utilizan cuatro espacios para delimitar bloques de código, en lugar de llaves (`{ }`), como se acostumbra en otros lenguajes.

Los números en Python equivalen a `True` siempre que sean distintos a cero. Las colecciones (listas, tuplas, diccionarios e incluso las cadenas) equivalen a `True` cuando no están vacías (es decir, cuando `len(coleccion) > 0`).

```
a = [1, 2, 3, 4]
if a:
    print("La lista no está vacía.")
else:
    print("La lista está vacía.")
```

Bucle “while”

El bucle “while” ejecuta una porción de código mientras (*while*, en inglés) se cumpla una determinada condición.

```
a = 1
while a < 10:
    print(a)
    a += 1
```

El código imprime en pantalla los números del 1 al 9 ya que, según la condición, el bucle debe ejecutarse siempre que `a` sea menor a 10. En cada ejecución utiliza la función incorporada `print()` para enviar un mensaje a la pantalla y la sentencia `a += 1` para incrementar el valor de `a` en 1 (es equivalente a `a = a + 1`). Si hubiésemos omitido esta última línea, el código imprimiría 1 en pantalla infinitamente.

Bucle “for”

A diferencia de otros lenguajes, en Python el bucle “for” se utiliza para recorrer una colección de objetos (una lista, tupla, diccionario, etc.). Se dice que un objeto es “iterable” cuando puede ser “recorrido” (estrictamente hablando el concepto de objeto iterable es un tanto más complejo, pero para nuestro propósito vamos a obviar los detalles).

```
a = [1, 2, 3, 4]
for i in a:
    print(i)
```

Este código imprime en pantalla cada uno de los elementos en la lista `a`. Le estamos indicando a Python que “para cada elemento de la lista `a`, ejecute el siguiente código (`print(i)`), luego de haber colocado el valor de dicho elemento en `i`”.

Ahora bien, si queremos imprimir números del 1 al 100, lógicamente no es muy cómodo escribir manualmente `a = [1, 2, 3, 4, ..., 100]`. Para ello existe la función `range()`.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

(La llamada a `list()` es necesaria únicamente en la consola interactiva para ver todos los elementos del rango).

En la primera llamada indicamos que queremos una lista de 10 números (que por defecto, como todo en Python, empieza desde el 0). En la segunda especificamos que la lista debe empezar desde el 1 y terminar cuando se haya alcanzado el 11, lo que resulta en una colección del 1 al 10.

Para recorrer simultáneamente los índices de una colección y su respectivo valor podría hacerse:

```
nombres = ["Pablo", "Juan", "Pedro"]
for i in range(len(nombres)):
    nombre = nombres[i]
    print(i, nombre)
```

Lo cual imprime (nótese que `print()` puede imprimir varios objetos seguidos por una coma entre los cuales colocará un espacio):

```
0 Pablo
1 Juan
2 Pedro
```

No obstante la forma óptima de hacerlo en Python es usando la función `enumerate()`.

```
nombres = ["Pablo", "Juan", "Pedro"]
for i, nombre in enumerate(nombres):
    print(i, nombre)
```

En este código se hace uso de una propiedad llamada *unpacking* de la cual hablaremos más adelante, ya que la función `enumerate(nombres)` genera la siguiente lista¹:

```
[(0, 'Pablo'), (1, 'Juan'), (2, 'Pedro')]
```

¹ Estrictamente hablando genera un iterador.

En ambos bucles, las palabras reservadas `break` y `continue` lo fuerzan a finalizar su ejecución o bien continuar con la siguiente iteración, respectivamente.