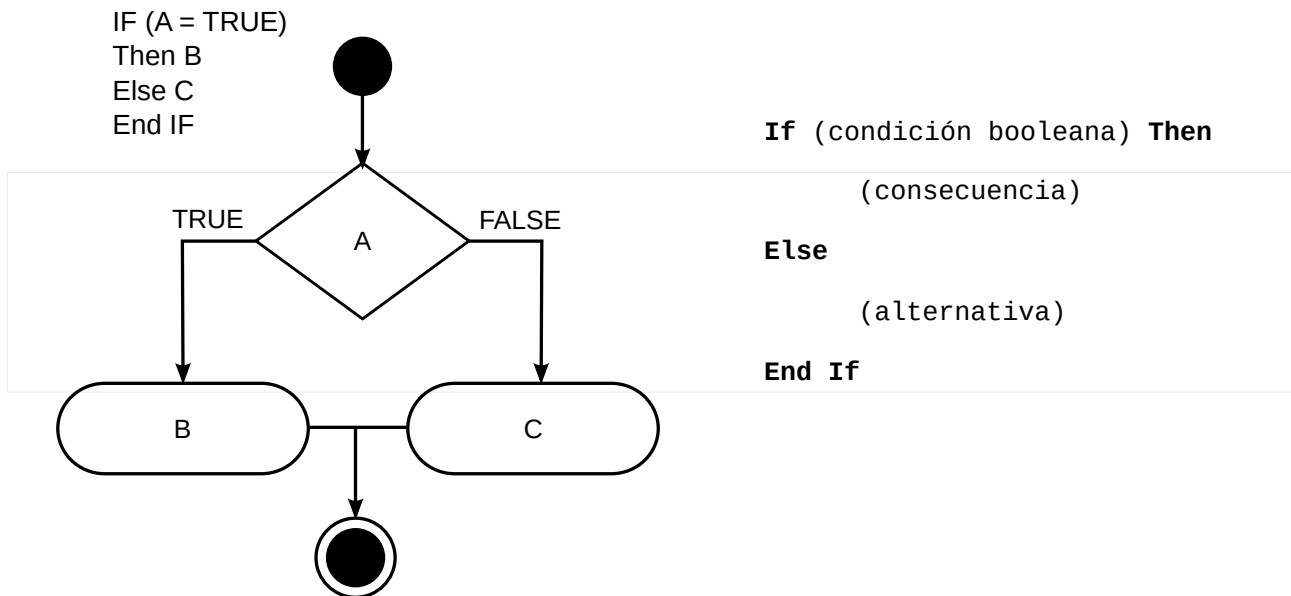


## 1.- Los condicionales

En los lenguajes de programación, los condicionales son estructuras que permiten elegir entre la ejecución de una acción u otra.

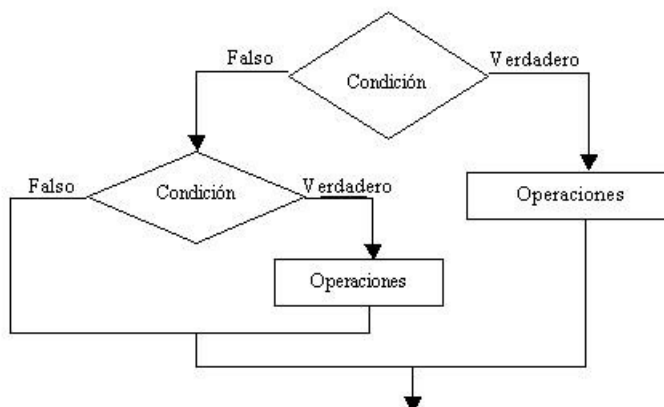
Los condicionales realizan diferentes cálculos o acciones en función de si se cumple o no una condición booleana, que se evalúa a "True" (verdadero) o "False" (falso).



Si la condición booleana es verdadera (True), se ejecuta el bloque de instrucciones contenido en B; si es falsa (False), se ejecuta el bloque de instrucciones C. En ambos casos el control de la ejecución del programa vuelve al punto situado tras el "End If".

La parte del bloque "else" es opcional, es decir, hay veces que se ejecuta un bloque de instrucciones si la condición booleana se evalúa a "False" y otras veces no se hace nada.

También podemos tener estructuras condicionales anidadas:



## Sentencia if

Si la condición que sigue a la palabra clave "if" se evalúa como verdadera, el bloque de código se ejecutará.

A continuación vamos a ver un par de ejemplos en python.

Ejemplo 1:

```
if True:
    print('¡el bloque If se ejecutará!')
```

Ejemplo 2:

```
x = 5
```

```
if x > 4:
    print("¡La condición era verdadera!") #Esta sentencia se ejecuta
```

Se usa la indentación para indicar que líneas de código forman parte del bloque del "if"

## Sentencia else

Opcionalmente, puedes agregar una respuesta "else" que se ejecutará si la condición es "false"

Veamos el siguiente ejemplo en python:

```
y = 3
```

```
if y > 4:
    print("¡No voy a imprimir!") #esta sentencia no se ejecuta
else:
    print("¡La condición no era verdadera!") #esta sentencia se ejecuta
```

## Else if

Usando "else if" es posible combinar varias condiciones. Solo se ejecutará el bloque de instrucciones que sigue a la primera condición que sea verdadera. El resto de los bloques de instrucciones se saltarán.

Por ejemplo, una tienda que ofrece hasta un 30% de descuento en los artículos que vende:

```
if descuento < 11% then
  print (tienes que pagar 30€)
elseif descuento < 21% then
  print (tienes que pagar 20€)
elseif descuento < 31% then
  print (tienes que pagar 10€)
end if;
```

En este ejemplo, si el descuento es 10%, entonces la primera condición se evalúa como verdadera (True), se imprimirá "tienes que pagar 30€". El resto de las condiciones que hay debajo se saltarán y el control del programa vuelve al punto después del "end if".

Otro ejemplo, esta vez en python:

```
z = 7

if z > 8:
  print("¡No voy a imprimir!") #esta sentencia no se ejecuta
elif z > 5:
  print("¡Yo lo haré!") #esta sentencia se ejecuta
elif z > 6:
  print("¡Tampoco voy a imprimir!") #esta sentencia no se ejecuta
else:
  print("¡Yo tampoco!") #esta sentencia no se ejecuta
```

*Solo se ejecutará la primera condición que se evalúe como "true".*

Aunque "z > 6" es "true", el bloque "if/elif/else" termina después de la primera condición verdadera. Esto significa que un "else" solo se ejecutará si ninguna de las condiciones es "true".

## **Sentencias if anidadas**

También podemos crear if anidados para la toma de decisiones.

Tomemos un ejemplo de cómo encontrar un número que sea par y también mayor que 10

```
x = 34
if x % 2 == 0: # así es como creas un comentario y ahora comprueba número par.
  if x > 10:
    print("Este número es par y es mayor que 10")
  else:
    print("Este número es par, pero no mayor 10")
else:
  print ("El número no es par. Así que punto de verificación más.")
```

## Operador ternario

Los operadores ternarios son más conocidos en Python como expresiones condicionales. Estos operadores evalúan si una expresión es verdadera o no. Se añadieron a Python en la versión 2.4.

**Forma:**

```
condition_if_true if condition else condition_if_false
```

**Ejemplo:**

```
es_bonito = True  
estado = "Es bonito" if es_bonito else "No es bonito"
```

Como se puede ver, permiten verificar de manera rápida una condición, y lo mejor de todo es que se puede hacer en una sola línea de código. Por lo general hacen que el código sea más compacto y fácil de leer.

## 2.- ¿Cuáles son los diferentes tipos de bucles en python? ¿Por qué son útiles?

Los bucles son una herramienta muy importante en la programación, ya que nos permiten ejecutar un bloque de código varias veces seguidas. Hay dos tipos principales de bucles en python: los bucles "for" y los bucles "while".

En python, los bucles "for" se utilizan cuando sabemos de antemano cuantas veces hay que repetir la iteración. Se itera sobre colecciones de datos (listas, tuplas, diccionarios, etc).

En cambio, los bucles "while" se utilizan cuando no sabemos cuantas veces hay que repetir la iteración. El bucle se repetirá mientras se cumpla una cierta condición. Cuando deje de cumplirse, el bucle finaliza. Hay que tener cuidado a la hora de plantear la condición, ya que si nos equivocamos podemos crear un bucle infinito y hacer que el sistema de un error.

### Bucles "for"

Todos los bucles tienen en común que un bloque de código se ejecuta repetidamente. Sin embargo, el mecanismo de funcionamiento del bucle "for" en Python difiere significativamente del de otros lenguajes.

La mayoría de los lenguajes de programación utilizan una variable de bucle que se incrementa o disminuye cuando se ejecuta el bucle.

Empecemos primero con un ejemplo en JavaScript: se emiten los nombres contenidos en la lista "names" uno tras otro. Aquí utilizamos la variable de bucle 'i' como índice continuo de los elementos individuales:

```
names = ['Jack', 'Jim', 'John']
for (let i = 0; i < names.length; i++) {
    console.log("Here comes " + names[i]);
}
```

La indexación directa de elementos sucesivos de la lista debe tratarse con precaución. Esto se debe a que un intento de acceso fuera de los límites permitidos conduce a un error en el momento de la ejecución. Por lo general, se trata del famoso "Off-by-one Error"

Python, en cambio, demuestra que también se puede hacer de otra manera. Aquí tenemos el mismo ejemplo con un bucle "for" en Python: iteramos directamente sobre los elementos de la lista con una variable de bucle sin indexarlos:

```
names = ['Jack', 'Jim', 'John']
for name in names:
    print(f"Here comes {name}")
```

La convención en python es usar un nombre en plural (*names*) para la variable que contiene la colección de elementos y el mismo nombre en singular (*name*) para la variable que se utiliza para iterar sobre dichos elementos.

El bucle "for" de Python es mucho más fácil de usar. Aunque se utiliza la misma palabra clave "for", se trata de un enfoque radicalmente diferente. En lugar de incrementar una variable de bucle y así indexar sucesivamente los elementos, iteramos directamente sobre los elementos de una colección. El bucle "for" en Python es, por tanto, comparable a la estructura "forEach" de algunos lenguajes como Ruby y JavaScript.

Vamos a emitir las letras individuales de una palabra a modo de ilustración. Dentro del bucle "for," una letra de la palabra se pone a disposición de la variable 'letter' por cada pasada del bucle. Esto **funciona sin una variable de bucle** y, por lo tanto, sin el riesgo de producir un Off-by-one Error. El código es preciso y fácil de leer:

```
word = "Python"

for letter in word:
```

```
print(letter)
```

## Bucles "while"

La sintaxis de un bucle "while" es similar a la de una instrucción "if". Proporciona una condición y el código que quiere ejecutar mientras la condición sea True.

### Sintaxis:

```
while <condition>:  
    # code here
```

### Ejemplo:

Veamos cómo se puede crear código para pedir a los usuarios que introduzcan valores y, después, permitirles usar *done* cuando terminen de introducirlos. En nuestro ejemplo, la entrada de usuario es la condición que se comprueba al principio del bucle "while".

```
user_input = ''  
  
while user_input.lower() != 'done':  
    user_input = input('Enter a new value, or done when done')
```

Hay que tener en cuenta que usa "input" para preguntar a los usuarios. Cada vez que los usuarios escriben un nuevo valor, cambian la condición, lo que significa que el bucle "while" se cerrará una vez que se haya escrito *done*.

### 3.- ¿Qué es una comprensión de listas en python?

La comprensión de listas, del inglés list comprehension, es una funcionalidad que nos permite crear listas avanzadas en una misma línea de código.

#### Síntaxis:

[expresion for variable in colección if condición]

A menudo la "expresión" (es decir, aquello que terminará inserto en la lista resultante) es igual a la "variable", y la "condición" es opcional. La "colección" puede ser una lista o cualquier otro objeto iterable (esto es, cualquier cosa sobre lo que podamos aplicar un bucle «for»).

Veamos varios ejemplos.

#### Ejemplo 1: Crear una lista con las letras de una palabra

```
# Método tradicional
lista = []
for letra in 'casa':
    lista.append(letra)
print(lista)                # ['c', 'a', 's', 'a']
```

```
# Con comprensión de listas
lista = [letra for letra in 'casa']
print(lista)                # ['c', 'a', 's', 'a']
```

Gracias a la comprensión de listas podemos indicar directamente cada elemento que va a formar la lista, en este caso la letra, a la vez que definimos el "for" para iterar sobre cada uno de los elementos de la cadena 'casa'.

#### Ejemplo 2: Crear una lista con todos los múltiplos de 2 entre 0 y 10.

```
# Método tradicional
lista = []
for numero in range(0, 11):
    if numero % 2 == 0:
        lista.append(numero)
print(lista)                # [2, 4, 6, 8, 10]
```

```
# Con comprensión de listas
lista = [numero for numero in range(0,11) if numero % 2 == 0 ]
print(lista)                # [2, 4, 6, 8, 10]
```

En este caso podemos observar que incluso podemos marcar una condición justo al final para añadir o no el elemento en la lista.

A través de la comprensión de listas también podemos expresar de forma compacta un conjunto de bucles anidados.

Por ejemplo, el siguiente código crea una lista "points" que contiene (en forma de tuplas de dos elementos) la posición de todos los puntos bidimensionales entre las coordenadas "(0, 0)" y "(5, 10)".

```
#Metodo tradicional
points = []
for x in range(0, 5 + 1):
    for y in range(0, 10 + 1):
        points.append((x, y))
print(points)

# Con comprensión de listas
points = [(x, y) for x in range(0, 5 + 1) for y in range(0, 10 + 1)]
print(points)
```

En ambos casos el resultado de la impresión sería:

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

Podríamos incluso agregar una condición usando ambas variables, por ejemplo, para mostrar solo los puntos en los que "x" es igual a "y".

```
points = [(x, y) for x in range(0, 5 + 1) for y in range(0, 10 + 1) if x == y]
```

## Otras colecciones

Esto que acabamos de decir se aplica por extensión a otras colecciones. Por ejemplo, podemos crear un diccionario de la misma forma, pero en este caso utilizamos llaves en lugar de corchetes.



```
doubles = {n: n * 2 for n in range(1, 11)}  
print(doubles)          # {1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9: 18, 10: 20}
```

Este fragmento genera un diccionario (`doubles`) donde las claves son números enteros del 1 al 10 y los valores, el doble de cada una de esas claves.

## 4.- ¿Qué es un argumento en Python?

En la definición de una función los valores que se reciben se denominan **parámetros**, pero durante la llamada los valores que se envían se denominan **argumentos**.

```
def greeting(first_name, last_name):  
    return f'Hi, {first_name} {last_name}'
```

```
print(greeting("Ana", "Garcia"))
```

En el ejemplo anterior tenemos:

parámetros: `first_name`, `last_name`

argumentos: `"Ana"`, `"Garcia"`

En python tenemos varios tipos de argumentos, que detallamos a continuación.

### Argumentos por posición

Cuando enviamos argumentos a una función, estos se reciben por orden en los parámetros definidos. Se dice por tanto que son argumentos por posición:

```
def resta(a, b):  
    return a - b  
  
resta(30, 10)  # argumento 30 => posición 0 => parámetro a  
               # argumento 10 => posición 1 => parámetro b
```

## Argumentos por nombre

Sin embargo es posible evadir el orden de los parámetros si indicamos durante la llamada que valor tiene cada parámetro a partir de su nombre:

```
def resta(a, b):  
    return a - b  
  
print(resta(b=30, a=10))      # -20
```

## Llamada sin argumentos

Al llamar una función que tiene definidos unos parámetros, si no pasamos los argumentos correctamente provocará un error:

```
def resta(a, b):  
    return a - b  
  
resta()  
  
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-4-78c8f433960e> in <module>()  
----> 1 resta()  
  
TypeError: resta() missing 2 required positional arguments: 'a' and 'b'
```

## Parámetros por defecto

Para solucionarlo podemos asignar unos valores por defecto nulos a los parámetros, de esa forma podríamos hacer una comprobación antes de ejecutar el código de la función:

```
def resta(a=None, b=None):  
    if a == None or b == None:  
        print("Error, debes enviar dos números a la función")  
        return # indicamos el final de la función aunque no devuelva nada  
    return a-b  
  
resta()      # Error, debes enviar dos números a la función
```

## Uso de \*args y \*\*kwargs

La mayoría de los programadores nuevos en Python tienen dificultades para entender el uso de \*args y \*\*kwargs. ¿Para qué se usan? Lo primero de todo es que en realidad no tienes porque usar los nombres args o kwargs, ya que se

trata de una mera convención entre programadores. Sin embargo lo que si que tienes que usar es el asterisco simple "\*" o doble "\*\*". Es decir, podrías escribir `*variable` y `**variables`. Empecemos viendo el uso de `*args`.

El principal uso de `*args` y `**kwargs` es en la definición de funciones. Ambos permiten pasar un número variable de argumentos a una función, por lo que si quieres definir una función cuyo número de parámetros de entrada puede ser variable, considera el uso de `*args` o `**kwargs` como una opción. De hecho, el nombre de `args` viene de argumentos, que es como se denominan en programación a los parámetros de entrada de una función.

### 1.1. Uso de `*args`

En Python, el parámetro especial `*args` en una función se usa para pasar, de forma opcional, un número variable de argumentos posicionales.

Lo que realmente indica que el parámetro es de este tipo es el símbolo '\*', el nombre `args` se usa por convención.

- El parámetro recibe los argumentos como una tupla.
- Es un parámetro opcional. Se puede invocar a la función haciendo uso del mismo, o no.
- El número de argumentos al invocar a la función es variable.
- Son parámetros posicionales, por lo que, a diferencia de los parámetros con nombre, su valor depende de la posición en la que se pasen a la función.

Veamos algunos ejemplos:

**Ejemplo 1:** Función para sumar un número de elementos variables.

```
def sum(*args):
    value = 0
    for n in args:
        value += n
    return value

print(sum())           # 0
print(sum(1, 2))       # 3
print(sum(1, 3, 5))    # 9
```

### Ejemplo 2: Función que recibe un argumento posicional y \*args

```
def test_var_args(f_arg, *args):  
    print("primer argumento normal:", f_arg)  
    for arg in args:  
        print("argumentos de *args:", arg)  
  
test_var_args('python', 'foo', 'bar')
```

Y la salida que produce el código anterior al llamarlo con 3 parámetros es la siguiente:

```
primer argumento normal: python  
argumentos de *args: foo  
argumentos de *args: bar
```

## 1.2. Uso de \*\*kwargs

\*\*kwargs permite pasar argumentos de longitud variable asociados con un nombre o **"key"** a una función. Deberías usar \*\*kwargs si quieres manejar argumentos con nombre como entrada a una función. Aquí tienes un ejemplo de su uso.

```
def saludame(**kwargs):  
    for key, value in kwargs.items():  
        print(f"{key} = {value}")  
  
print(saludame(nombre="Covadonga"))          # nombre = Covadonga
```

Es decir, dentro de la función no solo tenemos acceso a la variable como con \*args, sino que también tenemos acceso a un nombre o key asociado.

Por último, si quieres usar los tres tipos de argumentos de entrada a una función: normales, \*args y \*\*kwargs, deberás hacerlo en el siguiente orden.

```
funcion(fargs, *args, **kwargs)
```

## Ejemplo:

```
def greeting(time_of_day, *args, **kwargs):
```

```
    print("Hi {' '.join(args)}, I hope you're having a good {time_of_day}.")
```

```
    if kwargs:
```

```
        print("Your tasks for the day are:")
```

```
        for key, val in kwargs.items():
```

```
            print(f'{key} → {val}')
```

```
greeting("morning",
```

```
        "Raquel", "Boada",
```

```
        first = "Empty dishwasher",
```

```
        second = "Take dog for a walk",
```

```
        third = "Math homework")
```

## La impresión en consola sería:

```
Hi Raquel Boada, I hope you're having a good morning.
```

```
Your tasks for the day are:
```

```
first → Empty diswasher
```

```
second → take dog for a walk
```

```
third → math homework
```

## 5.- ¿Qué es una función de Python Lambda?

Las expresiones lambda se usan idealmente cuando necesitamos hacer algo simple y estamos más interesados en hacer el trabajo rápidamente en lugar de nombrar formalmente la función.

Son una forma corta de declarar funciones pequeñas y anónimas (no es necesario proporcionarles un nombre).

Las funciones Lambda se comportan como funciones normales declaradas con la palabra clave "def". Resultan útiles cuando se desea definir una función pequeña de forma concisa. Pueden contener solo una expresión, por lo que no son las más adecuadas para funciones con instrucciones de flujo de control.

### Sintaxis

lambda argumentos: expresión

Las funciones Lambda pueden tener cualquier número de argumentos, pero solo una expresión. Se ejecuta la expresión y se devuelve el resultado.

### Vamos a ver una "función normal" y un ejemplo de "Lambda":

```
#Aquí tenemos una función creada para sumar.  
def suma(x,y):  
    return(x + y)
```

```
#Aquí tenemos una función Lambda que también suma.  
lambda x,y : x + y
```

```
#Para poder utilizarla necesitamos guardarla en una variable.  
suma_dos = lambda x,y : x + y
```

Al igual que ocurre en las "*list comprehensions*" o *listas de comprensión*, lo que hemos hecho es escribir el código en una sola línea y limpiar la sintaxis innecesaria.

En lugar de usar "**def**" para definir nuestra función, hemos utilizado la palabra clave "**lambda**"; a continuación escribimos "**x, y**" como argumentos de la función, y "**x + y**" como expresión. Además, se omite la palabra clave "**return**", condensando aún más la sintaxis.

Por último, y aunque la definición es anónima, la almacenamos en la variable "**suma\_dos**" para poder llamarla desde cualquier parte del código, de no ser así tan solo podríamos hacer uso de ella en la línea donde la definamos.

Vamos a ver otros ejemplos

### **Ejemplo 1:** Calcular el cuadrado de un número

```
# Función Lambda para calcular el cuadrado de un número
square = lambda x: x ** 2
print(square(3)) # Resultado: 9

# Funcion tradicional para calcular el cuadrado de un numero
def square1(num):
    return num ** 2
print(square(5)) # Resultado: 25
```

En el ejemplo de lambda anterior, "lambda x: x \*\* 2" produce un objeto de función anónimo que se puede asociar con cualquier nombre. Entonces, asociamos el objeto de función con "square". De ahora en adelante, podemos llamar al objeto "square" como cualquier función tradicional, por ejemplo, "square(10)"

### **Ejemplo 2:** Añade 10 al argumento "a" y devuelve el resultado

```
X = lambda a: a + 10
print(x(5))      # 15
print(x(12))     # 22
```

### **Ejemplo 3:** Suma los argumentos a, b y c y devuelve el resultado

```
x = lambda a, b, c: a + b + c
print(x(5, 6, 2)) # 13
```

## **6.-¿Qué es un paquete pip?**

**Pip** en python es un sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software escritos en python y descargarlos a nuestro ordenador con la finalidad de integrarlos a nuestros desarrollos. **Pip** a su vez está realizado en python.

Muchos de estos paquetes pueden ser encontrados en el Python Package Index (PyPI).

Python 2.7.9 y posteriores (en la serie Python2), Python 3.4 y posteriores incluyen pip (pip3 para Python3) por defecto; lo cual no es necesario instalarlo

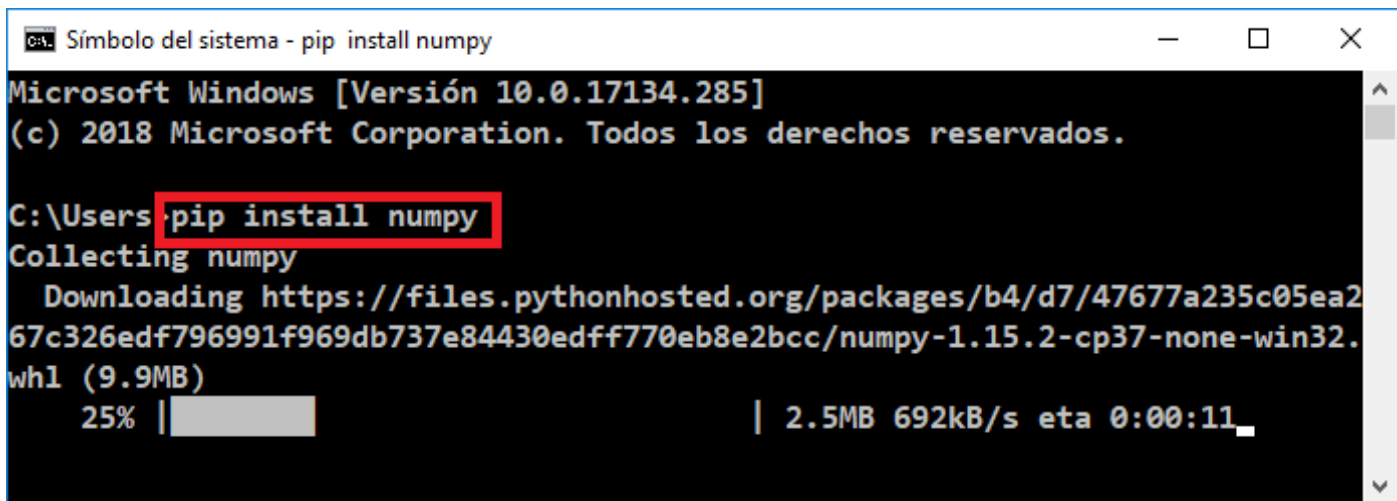
en nuestro ordenador, ya que al instalar python en la version 3.4 o superior en automático se instala el gestor de paquetes.

**Pip** es un acrónimo recursivo que se puede interpretar como *Pip Instalador de Paquetes* o *Pip instalador de Python*.

Una ventaja importante de **pip** es la facilidad de su interfaz de línea de comando, el cual permite instalar paquetes de software de Python fácilmente desde solo una orden:

```
pip install nombre-paquete
```

Para el ejemplo se instaló la librería Numpy:



```
Símbolo del sistema - pip install numpy
Microsoft Windows [Versión 10.0.17134.285]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users> pip install numpy
Collecting numpy
  Downloading https://files.pythonhosted.org/packages/b4/d7/47677a235c05ea267c326edf796991f969db737e84430edff770eb8e2bcc/numpy-1.15.2-cp37-none-win32.whl (9.9MB)
    25% | ██████████ | 2.5MB 692kB/s eta 0:00:11_
```

Los usuarios también pueden fácilmente desinstalar algún paquete:

```
pip uninstall nombre-paquete
```

Otra característica particular de **pip** es que permite gestionar listas de paquetes y sus números de versión correspondientes a través de un archivo de requisitos. Esto nos permite una recreación eficaz de un conjunto de paquetes en un entorno separado (p. ej. otro ordenador) o entorno virtual. Esto se puede conseguir con un archivo correctamente formateado llamado `requisitos.txt` y la siguiente orden:

```
pip install -r requisitos.txt
```

Con **pip** es posible instalar un paquete para una versión concreta de Python, sólo es necesario reemplazar “`${versión}`” por la versión de Python que queramos: 2, 3, 3.4, etc:



```
pip${versión} install nombre-paquete
```

## Cómo actualizar pip para Python

Si tuviéramos una versión anterior o muy viejita instalada de pip en nuestro ordenador y quisiéramos tener la última versión con la finalidad de descargar también las últimas versiones de librerías que se encuentran en los repositorios de python se puede hacer si ningún problema.

PIP no es una aplicación que se actualice muy a menudo. Sin embargo, no por ello es menos importante hacer uso de las nuevas versiones cuando están disponibles, ya que además de corregir errores y problemas de compatibilidad, también pueden corregir posibles agujeros de seguridad. Actualizar **pip** en *python* es muy sencillo:

Para actualizar **pip** en Windows utiliza el siguiente comando:

```
python -m pip install -U pip
```