1. ¿Qué es una condicional?

Condicionales en python: qué son y sintáxis

Una condicional es una estructura de programación que nos permite ejecutar uno o distintos bloques de código en base a unas condiciones.

La sintáxis básica consiste en la palabra clave **if** seguida de la condición a cumplir, y de dos puntos. La siguiente línea nos indica el código a ejecutar si la condición se cumple, y deberá estar indentada (este término hace referencia a dejar espacio al comienzo de la línea).

En este ejemplo a continuación, tenemos una variable llamada temperatura, y una condicional que comprueba si la variable es inferior a 25. Si la condición se cumple, nos imprimirá un texto advirtiéndonos de coger una chaqueta.

```
temperatura = 13

if temperatura < 25:
  print("Llévate una chaqueta que refresca")
```

```
Llévate una chaqueta que refresca
```

Dado que la condición se cumple, el programa imprimirá el texto. Si el valor de la variable temperatura fuese por ejemplo 30, el programa no imprimiría nada.

Uso de else y elif

Else

Puede ser que además de querer ejecutar un código si nuestra condición se cumple, también queramos ejecutar otro código cuando no se cumple. Para esto, utilizamos la palabra clave **else**

Siguiendo el ejemplo anterior, vamos a establecer una condición else. El programa comprobará si la primera condición se cumple, y si no es así, pasará a ejecutar el código establecido en **else**.

```
temperatura = 33

if temperatura < 25:
  print("Llévate una chaqueta que refresca")
else:
  print("Por favor llévate una botella de agua")
```

```
Por favor llévate una botella de agua
```

Elif

Podemos añadir condiciones adicionales a través de la palabra clave **elif**, diminutivo de else if.

En nuestro ejemplo, podemos añadir una condición que compruebe que el valor de la variable es superior a 40. El programa comprobará primero la primera condición, si no se cumple pasará a la segunda, y si tampoco se cumple pasará al apartado else.

```
if temperatura < 25:
  print("Llévate una chaqueta que refresca")
elif temperatura > 40:
  print("Tú no sales, que te da una insolación")
else:
  print("Por favor llévate una botella de agua")
```

Operadores de comparación

Los operadores de comparación se introducen dentro de una condición para hacer las comparaciones entre los valores de la misma. La lista es la siguiente:

```
 == Igual a
 != Diferente a
 <> Expresa inequalidad pero está obsoleto
 > Mayor que
 >= Mayor o igual que
 < Menor que
 <= Menor o igual que
```

Los últimos cuatro operadores no podrán utilizarse para strings. A continuación un ejemplo sencillo de la utilización de cada uno.

```
if beneficios == 0:
  print("Hemos llegado al punto de equilibrio")
elif beneficios != 0:
  print("Sabemos que no estamos en el punto de equilibrio")
elif beneficios >= 0:
  print("Estamos al menos en el punto de equilibrio")
elif beneficios > 0:
  print("Estamos teniendo beneficios")
elif beneficios <= 0:
  print("Estamos como mucho en el punto de equilibrio")
elif beneficios < 0:
  print("Tenemos pérdidas")
```

La realidad es que técnicamente este ejemplo no funcionará correctamente, ya que todo valor diferente a 0 caerá en la segunda condición, y no pasará a comprobar las siguientes. Por eso es importante asegurarse de que las condiciones no se solaparán. O si queremos que se solapen, procurar que el orden de la condición que queramos priorizar sea anterior.

Operadores in y not in

Los operadores **in** y **not in** ter permiten si un valor es o no parte de una colección. Estos operadores se conocen también como operadores de membresía.

Vamos a verlo con un ejemplo muy sencillo:

```
letters = ["a", "b", "c", "d"]

if "a" in letters:
  print("There is an a")
```

Esto funcionaría también si la variable letters fuese un string:

```
letters = "abcd"

if "a" in letters:
  print("There is an a")
```

Ahora vamos a verlo con otro ejemplo. Este es el club de no Homers, y si no has crecido con los Simpson deberías saber que club de no HomerS significa que solo puede haber un Homer

```
#The no Homers club

members = ['Lenny', 'Carl', 'Barney']
new_member = 'Homer'

if new_member != 'Homer':
  members.append(new_member)
  print('You are in, welcome to the club')
elif new_member == 'Homer':
  if 'Homer' in members:
    print('Get out, we already have a Homer!')
  elif 'Homer' not in members:
    members.append(new_member)
    print('Ok you are in, first Homer')

print(members)
```

Así, tenemos una primera condición con la que cualquiera que no se llame Homer podrá entrar en el club, y su nombre será incluido en la lista de miembros.

Pero si el nombre del nuevo miembro es Homer, el programa comprobará primero si ya hay un Homer en la lista de miembros. Si lo hay, este segundo Homer ya no podrá entrar en el club. Si no hay ningún Homer en la lista, significa que es el primero y sí puede pertenecer al club.

En este ejemplo podemos ver además que podemos tener condiciones anidadas dentro de otra condición.

Un aspecto importante a tener en cuenta es que en las condiciones se diferencia entre mayúsculas y minúsculas. Por lo tanto, si el nuevo miembro escribiese su nombre como 'homer' en lugar de 'Homer' podría saltarse nuestros filtros de seguridad.

Existe una solución para esto, y es que podemos transformar en minúscula o mayúscula ambos valores a comparar:

```
if new_member.lower() != 'Homer'.lower():
```

Condicionales compuestos

Los condicionales compuestos consisten en condiciones múltiples dentro de una clave if. La sintáxis consiste en unir las condiciones mediante las palabras clave **and** (ambas condiciones deben cumplirse) u **or** (basta con que se cumpla solo una de las condiciones).

En el siguiente ejemplo tenemos un programa de log-in en el que es necesario que tanto el usuario/email como la contraseña sean correctos. Envolvemos la condición or entre paréntesis para que se ejecute primero. Así, basta con que uno entre username y email sean correctos para pasar a la comprobación de la contraseña.

```
username = "Bluey"
email = "bluey@heeler.com"
password = "heelerbluey"

if (username == "Bluey" or email == "bluey@heeler.com") and password == 'heelerbluey':
  print("Access Granted")
else:
  print("Access Denied")
```

```
Access Granted
```

Operador ternario

El operador ternario o ternary operator es una declaración if-else que se define en una sola línea.

Es recomendable usarlo para condiciones cortas, y es especialmente útil para almacenar el output de la condicional en una variable.

Veamos el ejemplo de esta condicional con sintaxis clásica:

```
num = 5

if num % 2 == 0:
  print("Even")
else:
  print("Odd")

#Odd
```

Y este sería el operador ternario equivalente:

```
print("Even") if num % 2 == 0 else print("Odd")
```

Como podréis ver la estructura es la siguiente:

```
# [código si se cumple] if [condición] else [código si no se cumple]
```

A continuación podemos ver cómo guardar el resultado en una variable en lugar de imprimirlo, para poder utilizarlo en otro punto del programa:

```
result = "Even" if num % 2 == 0 else "Odd"

print(result)
```

##¿Cuando usar el operador ternario

Si tu operador ternario es tan largo que comienza a requerir múltiples líneas, lo más recomendable será que utilices la sintaxis tradicional. Asimismo, es importante comprobar que el operador ternario sea legible.

Fuentes

Agradecimiento a:

Guías Devcamp

https://ellibrodepython.com/if-python

https://www.w3schools.com/python/python_conditions.asp

https://realpython.com/python-in-operator/

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

Bucles o loops en Python: qué son

Un bucle es una instrucción que se repite varias veces mientras se cumpla una condición. Cuando se deje de cumplir, se saldrá del bucle y se continuará la ejecución normal.

Existen dos tipos de bucles, los que tienen un número de iteraciones no definidas, y los que tienen un número de iteraciones definidas. El while estaría dentro del primer tipo y los for se engloban en el segundo.

Bucles for o for loops

En los bucles for el número de iteraciones está definido de antemano, ya que existe un iterable que define las veces que se ejecutará el código. Así, el bucle for se usa para iterar sobre una secuencia (puede ser una lista, una tupla, un diccionario, un set o un string.

La sintaxis sería la siguiente, la palabra clave **for** seguida del nombre de la variable iterable, **in** y el nombre de la colección sobre la que queramos iterar. Al final de la línea debemos poner dos puntos. La siguiente línea deberá estar indentada, y contiene el bloque de código que queremos ejecutar para cada elemento de la colección.

```
meals = ["spam", "egg", "sausage", "spam"]
for x in meals:
  print(x)
```

```
spam
egg
sausage
spam
```

El nombre de la variable iterable (x en el ejemplo superior) puede ser el que nosotros decidamos. Sin embargo, es una convención común que cuando iteramos sobre una colección, el nombre de la variable iterable sea el singular del nombre de la variable de colección (si es plural). Así, el siguiente código se ajustaría mejor a la convención general:

```
meals = ["spam", "egg", "sausage", "spam"]
for meal in meals:
  print(meal)
```

Con el ejemplo anterior hemos visto como se itera sobre listas, vamos a ver ahora ejemplos de otros iterables.

Iterar sobre diccionarios

Para iterar sobre diccionarios, en lugar de dar un nombre de variable iterable, proporcionamos dos nombres (de nuestra elección). El primero se asociará a las key, y

el segundo a los valores. También debemos aplicar la función items() al diccionario. Esta función nos devuelve un objeto view con tuplas dentro, y podemos iterar sobre él.

```
lancelot = {
  "Name": "Sir Lancelot of Camelot",
  "Quest": "To seek the Holy Grail",
  "Favourite color": "Blue"
}

for question, answer in lancelot.items():
  print("Question: ", question)
  print("Answer: ", answer)
```

```
Question:  Name
Answer:  Sir Lancelot of Camelot
Question:  Quest
Answer:  To seek the Holy Grail
Question:  Favourite color
Answer:  Blue
```

Iterar sobre caracteres de un string

El bucle for se puede usar también con strings, accediendo a cada caracter como si fuesen items de una colección. Ejemplo:

```
word = "SPAM"

for letter in word:
  print(letter)
```

```
S
P
A
M
```

Iterar sobre un rango

Cuando tenemos un determinado número de veces que queremos iterar, in querer estar limitados por una colección, podemos construir un rango. Los rangos toman desde 2 a tres argumentos. El primero determina donde comienza el rango, el segundo donde termina (el número establecido no estará incluido), y el tercero, que es opcional, el salto que queramos realizar.

```
for num in range(1, 11):
  print(num)
```

```
1
2
3
4
5
6
```

```
7
8
9
10
```

```
for num in range(1, 11, 2):
  print(num)
```

```
1
3
5
7
9
```

Funciones adicionales

Else

La palabra clave else especifica un bloque de código a ejecutar cuando el loop finalice.

```
for num in range(0, 3):
  print(num)
else:
  print("It's over!")
```

```
0
1
2
It's over!
```

Añadir elementos a listas

Podemos almacenar una lista vacía e una variable fuera de nuestro bucle, e ir añadiendo elementos en cada iteración mediante la función append.

```
list = []

for num in range(0, 3):
  list.append(num)

print(list)
```

```
[0, 1, 2]
```

Continue y break

En ocasiones, podemos buscar detener o alterar el comportamiento en algún lugar del bucle según una condición. La palabra clave continue indica al programa que debe seguir iterando.

En el siguiente ejemplo hemos construido un bucle que añadirá los nombres de los usuarios a la lista de miembros, pero si encuentra un bot simplemente lo ignorará:

```
members = []

users = ["Mike", "Sally", "bot", "Alex", "Melissa"]

for user in users:
  if user == "bot":
    continue
  members.append(user)

print(members)
# ['Mike', 'Sally', 'Alex', 'Melissa']
```

El concepto opuesto sería la palabra clave break, que interrumpe el bucle si encuentra la condición. En el siguiente ejemplo, si el bucle encuentra un bot entre la lista de usuarios nos envía un aviso y deja de añadir miembros.

```
members = []

users = ["Mike", "Sally", "bot", "Alex", "Melissa"]

for user in users:
  if user == "bot":
    print("Bot found, please check the veracity of the user list")
    break

  members.append(user)

print(members)

# Bot found, please check the veracity of the user list
# ['Mike', 'Sally']
```

Bucles while o while loops

Los bucles while nos permiten ejecutar una sección de código mientras una condición determinada se cumpla. Cuando se deje de cumplir, se saldrá del bucle.

Veamos un ejemplo sencillo:

```
num = 5
while num > 0:
    num -=1
    print(num)
```

```
4
3
2
1
0
```

La condición nos dice que el código seguirá ejecutándose mientras num sea mayor que 0. En este caso se itera 5 veces hasta que en la sexta num es igual a 0 y el bucle se

para. Asimismo, debemos ir decrementando el valor de num para que en algún momento alcance el 0, de ahí el código num -= 1 que en cada iteración restará 1 a la variable.

Veamos otro ejemplo, en el que establecemos un valor centinela fuera del bucle, y lo incrementamos en cada iteración para que el bucle sepa cuando terminar:

```
paw_patrol = ["Ryder", "Marshall", "Zuma", "Rocky", "Skye", "Rubble"]
counter = 0

while counter < len(paw_patrol):
  print(paw_patrol[counter])
  counter += 1
```

Errores

Debemos tener cuidado ya que un mal uso del while puede dar lugar a bucles infinitos. Si no establecemos un valor centinela para que el bucle se detenga, seguirá ejecutándose eternamente o al menos hasta hacer que tu máquina o el servidor se cuelguen.

```
# No ejecutes esto, en serio
while True:
    print("Bucle infinito")
```

Otro error común es el conocido como off-by-one error, por el que o bien damos instrucciones al programa para iterar una vez más de lo que deseamos, o una vez menos. Por ejemplo, ejecutar el siguiente código nos daría un error porque cuando el contador alcance el valor 6 el bucle intentará ejecutar el código, pero el índice 6 no existe en la lista.

```
paw_patrol = ["Ryder", "Marshall", "Zuma", "Rocky", "Skye", "Rubble"]
counter = 0
while counter <= len(paw_patrol):
  print(paw_patrol[counter])
  counter += 1

# IndexError: list index out of range
```

Fuentes

Agradecimiento a:

Guías Devcamp

https://www.simplilearn.com/tutorials/python-tutorial/python-loops

https://www.w3schools.com/python/python_for_loops.asp

https://www.w3schools.com/python/python_while_loops.asp

https://ellibrodepython.com/while-python

3. ¿Qué es una lista por comprensión en Python?

Comprensión de listas en Python: qué es

La compresión de listas nos permite crear listas de elementos en una sola línea de código.

Veámoslo con un ejemplo sencillo. Mediante este ejemplo crearemos una lista con los valores al cuadrado de la primera lista.

```
nums = [2, 3, 5, 10]

squares = [num**2 for num in nums]

print(squares)

# [4, 9, 25, 100]
```

La estructura, en la segunda línea, consiste en primero la variable que queremos asignar. Abrimos corchetes [] para convertir nuestro código en una lista, a continuación establecemos primero qué operación deberá contener cada elemento de nuestra lista. La siguiente parte son las palabras clave clásicas de un bucle for.

Equivalencia en múltiples líneas de código

El ejemplo del apartado anterior puede escribirse también en múltiples líneas de código:

```
nums = [2, 3, 5, 10]
squares = []

for num in nums:
  squares.append(num ** 2)

# [4, 9, 25, 100]
```

Añadiendo condicionales a la comprensión de listas

Podemos construir una lista en la que se realice la operación sobre el elemento solo si una determinada condición se cumple, añadiendo un condicional if. La expresión genérica sería la siguiente:

```
lista = [expresión for elemento in iterable if condición]
```

Continuemos con el ejemplo anterior, pero en esta ocasión solo querremos añadir los cuadrados a nuestra lista si el valor es impar:

```
nums = [2, 3, 5, 10]
odd_squares = [num**2 for num in nums if (num ** 2) % 2 != 0]

print(odd_squares)

# [9, 25]
```

Errores

Si intentamos ejecutar la comprensión de listas y se nos olvida envolver el código entre corchetes [], Python nos indicará un error de sintaxis.

Por otro lado, debemos destacar que hay que tener cuidado con su uso, ya que las compresiones muy largas pueden ser muy dificiles de leer.

Fuentes

Agradecimiento a:

Guías Devcamp

https://ellibrodepython.com/list-comprehension-python

https://www.w3schools.com/python/python_lists_comprehension.asp

https://ellibrodepython.com/list-comprehension-python

4. ¿Qué es un argumento en Python?

Argumentos de funciones: qué son

Antes de entrar a explicar el tema, hagamos un repaso de lo que son las funciones en Python. Una función es un bloque de líneas de código o un conjunto de instrucciones cuya finalidad es realizar una tarea específica. Puede reutilizarse a voluntad para repetir dicha tarea. Una función tiene tres componentes, un input, los procesos, y el output.

El input consiste en los argumentos o parámetros de la función, en a que podemos proporcionarle datos para que los procesos se realicen en base a ellos.

Existen funciones que no tienen argumentos, y por lo tanto no requieren un input para ser ejecutadas. Por ejemplo:

```
def greeting():
  print("Hello World!")

greeting()
```

Como podemos ver en el ejemplo, la función deberá ser llamada con paréntesis al final, aunque no tenga argumentos.

La función del siguiente ejemplo requiere dos argumentos. Los nombres de los argumentos podemos definirlos como queramos, pero es recomendable que guarden relación con los valores esperados.

```
def substraction(num1, num2):
  return num1 - num2


print(substraction(10, 8 ))
# 2
```

¿Argumentos o parámetros?

Los términos parámetro y argumento son utilizados para el mismo concepto: información que se pasa a una función.

Desde la perspectiva de una función:

- Un parámetro es la variable que aparece entre paréntesis en la definición de la función.
- Un argumento es el valor que se envía a la función cuando ésta es llamada.

Error en número de argumentos

Imaginemos que tenemos la función anterior, en la que se esperan dos argumentos. Si llamamos a la función pasando más o menos valores de los esperados, obtendremos un error que nos indicará claramente que el número de argumentos no es correcto:

```
def substraction(num1, num2):
  return num1 - num2


print(substraction(10, 8, 9))
#TypeError: substraction() takes 2 positional arguments but 3 were given
```

Argumentos por posición vs argumentos por nombre

En el ejemplo anterior hemos pasado los argumentos de la función por posición, en el que el primer valor que pasamos al llamar a la función se interpreta como primer argumento y así sucesivamente.

Tenemos también la opción de llamar a una función usando argumentos por nombre. La sintaxis es la siguiente:

```
def substraction(num1, num2):
  return num1 - num2


print(substraction(num2=3, num1=5))
# 2
```

Así, no es necesario que respetemos el orden de los argumentos.

Este método es recomendable cuando nuestra función tiene muchos argumentos. Asimismo, si indicamos un nombre de argumento que no se encuentra en la definición de la función obtendremos un error. Y sí, es sensible a mayúsculas.

```
def substraction(num1, num2):
  return num1 - num2


print(substraction(num2=3, Num1=5))
# TypeError: substraction() got an unexpected keyword argument 'Num1'
```

Argumentos por defecto

Como hemos visto antes, llamar una función sin los argumentos requeridos nos dará un error.

Sin embargo, puede ser que deseemos que una función se ejecute incluso si no pasamos el argumento. Para esto podemos tener un valor por defecto para nuestros argumentos. La sintáxis es la siguiente, en la que el valor por defecto será Brian.

```
def praise(name = 'Brian'):
  print(f"Bless {name}")

praise("the cheesemakers")
```

```
# Bless the cheesemakers
praise()
# Bless Brian
```

Así, si pasamos un valor al llamar a la función, se tomará este nuevo valor. Si no pasamos ninguno, tomará el argumento por defecto.

Mala praxis: listas como argumentos por defecto

Un posible problema a tener en cuenta surge cuando establecemos un argumento por defecto mutable, como por ejemplo una lista.

Si el código de nuestra función modifica la composición de la lista, debemos tener en cuenta que la siguiente vez que llamemos a la función no empezaremos de 0, sino que partiremos de la lista que se ha modificado. Por ejemplo:

```
def append_function(collection = []):
  collection.append(1)
  return collection

print(append_function())
# [1]
print(append_function())
# [1, 1]
```

Args y kwars

Uso de *args

Con los *args en Python, podemos definir funciones cuyo número de argumentos es variable. A esto también se le conoce como unpacking de argumentos.

La sintaxis consiste en añadir *args como argumento al definir la función. El uso de este término es una convención general, pero el código se ejecutará si en lugar de args se usa otra palabra.

En el ejemplo a continuación usamos la interpolación de strings junto con la función join() para imprimir todos los posibles argumentos, separados por una coma y un espacio:

```
def praise(*args):
  print("Bless " + ", ".join(args))

praise("the cheesemakers", "Brian", "manufacturers of dairy products")
# Bless the cheesemakers, Brian, manufacturers of dairy products
```

Si dentro de la función incluyésemos la instrucción de imprimir args, obtendríamos una tupla

Uso de kwargs

A diferencia de *args, los* *kwargs nos permiten dar un nombre a cada argumento de entrada, pudiendo acceder a ellos dentro de la función a través de un diccionario.

Como en el caso de args, la palabra kwargs es tan solo una convención.

Veamos en el siguiente ejemplo como en la definición de la función estamos proporcionando una key con sintaxis de diccionario, y al llamar la función especificamos los valores de estas keys.

```
def office(**kwargs):
  print(f'Our best employees are {kwargs["accountant"]}, {kwargs["salesperson"]}, and
{kwargs["receptionist"]}.')

office(accountant='Angela', salesperson='Dwight', receptionist='Pam')

# Our best employees are Angela, Dwight, and Pam.
```

Si dentro de la función incluyésemos la instrucción de imprimir kwargs, obtendríamos un diccionario.

Combinación de args y kwargs

Podemos combinar los args y kwargs entre sí, y también con argumentos clásicos. Debemos seguir el orden de primero los argumentos normales, después los args y por último los kwargs.

A continuación un ejemplo:

```
def office(company, *args, **kwargs):
  print(f'Welcome to {company}, we have branches in: {", ".join(args)}')

  print(f'Our best employees are {kwargs["accountant"]}, {kwargs["salesperson"]}, and
{kwargs["receptionist"]}.')


office('Dunder Mifflin',
       'Scranton', 'Utica',
      accountant='Angela', salesperson='Dwight', receptionist='Pam'
      )

# Welcome to Dunder Mifflin, we have branches in: Scranton, Utica
# Our best employees are Angela, Dwight, and Pam.
```

Fuentes

Agradecimientos a:

- Guías Devcamp
- https://www.w3schools.com/python/python_functions.asp
- https://ellibrodepython.com/funciones-en-python#pasando-argumentos-de-entrada
- https://docs.python-guide.org/writing/gotchas/

5. ¿Qué es una función Lambda en Python?

Funciones Lambda: qué son y sintaxis

Las funciones lambda o anónimas son un tipo de función pequeña que tipicamente se define en una línea e incluye solo una expresión.

La sintaxis básica es la siguiente

```
lambda argumentos: expresión
```

Veámoslo con un ejemplo muy sencillo. La siguiente función lambda:

```
lambda num1, num2: num1 + num2
```

sería la equivalente a la siguiente función tradicional:

```
def suma(num1, num2):
  return num1 + num2
```

Como podréis ver la función lambda no tiene nombre, de ahí que sea una función anónima.

¿Cómo usarlas?

Almacenarlas en una variable

Una función básica de las funciones lambda es poder almacenar su resultado en una variable, para luego poder usarla en otros puntos del programa.

De hecho, al guardarla en una variable podemos posteriormente tratarla como una función normal y llamarla con los arguments:

```
suma = lambda num1, num2: num1 + num2

print(suma(3,5))
# 8
```

Usarla dentro de otra función

Podemos usar una función lambda dentro de la definición de otra función. En el siguiente ejemplo usamos la función reduce de functools. Esta función toma dos argumentos, una función lambda y una colección, ejecuta la suma definida en la función lambda con los dos primeros elementos, y hace la misma operación con el resultado y el tercer elemento:

```
import functools

my_list = [1,2,3]

def suma_acumulada(list):
```

```
  return functools.reduce((lambda num1, num2: num1 + num2), list)


print(suma_acumulada(my_list))
#6
```

Sí, es totalmente cierto que la función sum() nos haría el apaño de forma más sencilla, pero esto es solo un ejemplo para ver las posibilidades de las funciones lambda.

Fuentes

Agradecimientos a:

- Guías Devcamp
- https://ellibrodepython.com/lambda-python
- https://www.w3schools.com/python/python_lambda.asp

6. ¿Qué es un paquete pip?

PIP: qué es

PIP es un gestor de paquetes para paquetes o módulos Python. Los módulos son bibliotecas de código Python que puedes incluir en tu proyecto.

Instalar PIP

Para instalar PIP, debemos seguir las instrucciones del siguiente enlace:

https://pip.pypa.io/en/stable/installation/#get-pip-py

Tal y como se indica, PIP estará automáticamente instalado si estas trabajando en un entorno virtual, utilizando Python descargado desde python.org o utilizando Python que no haya sido modificado por un redistribuidor para eliminar ensurepip.

Si no tienes PIP instalado, existen dos mecanismos apoyados directamente por los mantenedores de PIP, hacerlo mediante el módulo ensurepip o mediante el script get-pip.py.

Lo recomendable será comprobar primero que tengamos PIP instalado, y si no, elegir el método que prefiramos y seguir las indicaciones del link arriba mencionado.

Comprobar que PIP esté instalado

Puedes ejecutar el siguiente código en la línea de comando para comprobar que tengas PIP instalado:

```
pip –version
```

Si te indica la versión disponible, es que está correctamente instalado.

Python Package Index (PyPI)

PyPI o the Python Package Index es una herramienta de Python que funciona como un centro de paquetes de dependencias y otras herramientas para Python.

CheeseShop es un nombre clave secreto para PyPI (es secreto, pero podéis contarlo), y hace referencia a un sketch de los Monty Python.

La URL por la que se accede es la siguiente:

http://pypi.org/

PyPI te perminte también tener una cuenta de usuario para poder agregar nuevos paquetes, buscarlos o guardarlos para utilizarlos en tus programas.

Instalar un paquete

Para instalar un paquete podemos acceder a http://pypi.org/ e introducir en el buscador el nombre del paquete.

Por ejemplo, si buscamos numpy, podemos acceder al primer resultado (numpy 1.26.4). En la parte superior izquierda PyPI nos da el código que debemos introducir en la línea de comando, e incluso la opción de copiarlo directamente al portapapeles.

Para instalarlo debemos simplemente pegar la línea de código en la línea de comando:

```
pip install numpy
```

Si se instala correctamente, obtendremos un mensaje de confirmación. Asimismo, si ya está instalado en nuestro sistema, obtendremos un mensaje informándonos.

Usar un paquete

Para usar un paquete en tu programa, deberás primero importarlo en la parte superior de tu código.

Puedes importar el paquete completo:

```
import numpy
```

Recuerda que si lo importas así, cuando en tu código quieras usar las funciones del paquete tendrás que escribir primero su nombre:

```
numpy.arange()
```

E incluso puedes darle un alias, para que cuando tengas que llamar las funciones, no tengas que escribir el nombre completo

```
import numpy as np
```

También puedes importar solo una función en concreto del paquete. En este caso no tendrás que usar ni el nombre ni el alias para llamarla.

```
from numpy import arange

arange()
```

Desinstalar un paquete

Para desinstalar un paquete, simplemente podemos teclar pip uninstall seguido del nombre del paquete en la línea de comando, y responder a las preguntas Y/N que nos hace. Para desinstalar numpy deberíamos introducir la siguiente línea:

```
pip uninstall numpy
```

Algunos paquetes interesantes

Aquí están algunos de los nombres de paquetes que pueden resultar interesantes, para que puedas buscarlos en PyPI:

- Dash: Nos permite construir apps de visualización de datos en Python
- Colorama: te permite añadir color a tu terminal
- Requests: permite enviar peticiones HTTP utilizando Python
- Flask: permite crear un servicio web rápido o un sitio web sencillo

Fuentes

Agradecimientos a:

- Guías Devcamp
- https://pip.pypa.io/en/stable/
- https://keepcoding.io/blog/que-es-pypi/
- https://wiki.python.org/moin/CheeseShop
- https://pypistats.org/top
- https://python.land/top-15-python-packages