Python.industria(4.0)

PY005

Fundamentos de programación con Python – Parte III

Comenzamos en 5’

Python.industria(4.0)

PY005

Fundamentos de programación con Python – Parte III

Indice:

Funciones_com

01(

Control de flujo

~~Condicionales~~
~~Bucles~~
~~La función Range~~
While, Break y Continue
Comprensión de listas

)

While:

En Python, La función while se utiliza para crear bucles o ciclos que se ejecutan repetidamente mientras una determinada condición sea verdadera.

Al comienzo de cada iteración, se evalúa la condición y, si es verdadera, se ejecuta el bloque de código. Una vez finalizada la ejecución, se vuelve a evaluar la condición y el ciclo continúa mientras la condición siga siendo verdadera.

```
i=0

while i<10:
 i += 1
 print("Repeticion:", i)
```

```
Repeticion: 1
Repeticion: 2
Repeticion: 3
...
Repeticion: 9
Repeticion: 10
```

While:

Mucho cuidado:

Es importante tener precaución al utilizar un bucle while en Python para evitar caer en una condición que siempre sea verdadera.

Si la condición del bucle se configura incorrectamente o no se actualiza adecuadamente dentro del bloque de código, podría resultar en un ciclo infinito. Esto significa que el bucle se ejecutaría continuamente sin cesar, consumiendo recursos del sistema y potencialmente bloqueando el programa.

```
i=0

while i >= 0:
 i += 1
 print("Repeticion:", i)
```

```
Repeticion: 1
Repeticion: 2
Repeticion: 3
...
Repeticion: 866584
Repeticion: 866585
```

break:

Para éstos casos, tenemos una función muy interesante, la función break. Ésta función nos permite interrumpir un bucle de forma prematura.

Combinada con un condicional, podemos utilizarla de forma efectiva para salir del bucle cuando se cumple determinada condición

```
i=0

while i >= 0:
 i += 1
 print("Dentro del bucle...")
 print("Repeticion:", i)
 if i >= 5:
  break
print("----------------")
print("Fuera del bucle!")
```

```
Dentro del bucle…
Repeticion: 1
Dentro del bucle...
Repeticion: 2
Dentro del bucle...
Repeticion: 3
Dentro del bucle...
Repeticion: 4
Dentro del bucle...
Repeticion: 5
----------------
Fuera del bucle!
```

continue:

¿Y si queremos solamente “saltarnos” una iteración pero continuar con las restantes?

Para éstos casos, tenemos una función que nos resulta muy útil, la función continue. Esto permite omitir ciertas acciones o cálculos en situaciones específicas y continuar con las iteraciones posteriores.

```
i=0

while i >= 0:
 i += 1
 if (i % 2 == 1):
  continue
 print("Dentro del bucle...")
 print("Repeticion:", i)
 if i >= 10:
  break
print("----------------")
print("Fuera del bucle!")
```

```
Dentro del bucle…
Repeticion: 2
Dentro del bucle...
Repeticion: 4
Dentro del bucle...
Repeticion: 6
Dentro del bucle...
Repeticion: 8
Dentro del bucle...
Repeticion: 10
----------------
Fuera del bucle!
```

Comprensión.listas[

La comprensión de listas es una característica única de Python, y es ciertamente una de las más interesantes.

Vamos a ver algunos ejemplos para ver de qué se trata:

```
pares = []
for n in range(10):
    if n%2 == 0:
        pares.append(n)

pares
```

```
[0, 2, 4, 6, 8]
```

```
pares = [n for n in range(10) if(n%2 == 0)]
pares
```

```
[0, 2, 4, 6, 8]
```

]

Comprensión.listas[

La comprensión de listas en Python es una forma concisa y poderosa de crear listas utilizando una estructura compacta y legible.

Permite combinar bucles "for" y condiciones en una sola línea de código para generar y filtrar elementos de una lista de manera eficiente.

```
nueva_lista = [<expresión> for <elemento> in <iterable> if <condicion>]
```

Al recorrer el iterable, la expresión se evalúa y se agrega ala nueva lista si la condición es verdadera. Esto se realiza automáticamente en cada iteración del bucle, lo que resulta en una lista finalizada una vez que se han recorrido todos los elementos.

]

Comprensión.listas[

Por ejemplo, si tenemos una lista de números, y queremos mostrar solamente los negativos, podemos escribir algo así

```
numeros = [1, 6, -6, 45, -23, 12, -3, 5, 9]
negativos = [n for n in numeros if n<0]
negativos
```

“agrega n para cada n en números sólo si n es menor a cero”

De ésta manera, podemos escribir soluciones de filtrado muy complejas en una sola línea de código.

→→→

]

Comprensión.listas[

Vamos a ver cuáles ubicaciones tienen temperatura menor a 22 grados para encender la calefacción en esas zonas.

```
sensores_temperatura = {
    "Sala de producción": 25.5,
    "Oficinas administrativas": 22.1,
    "Almacén principal": 18.7,
    "Área de carga y descarga": 27.8,
    "Laboratorio de control de calidad": 24.3,
    "Zona de empaque": 21.6,
    "Área de refrigeración": 15.9,
    "Sala de máquinas": 28.2,
    "Comedor de empleados": 20.5,
    "Pasillos de distribución": 23.8
}
```

]

Comprensión.listas[

Vamos a ver cuáles ubicaciones tienen temperatura menor a 22 grados para encender la calefacción en esas zonas.

```
sensores_temperatura = {
    "Sala de producción": 25.5,
    "Oficinas administrativas": 22.1,
    "Almacén principal": 18.7,
    "Área de carga y descarga": 27.8,
    "Laboratorio de control de calidad": 24.3,
    "Zona de empaque": 21.6,
    "Área de refrigeración": 15.9,
    "Sala de máquinas": 28.2,
    "Comedor de empleados": 20.5,
    "Pasillos de distribución": 23.8
}
```

```
calefaccionar = [[sensor, sensores_temperatura[sensor]]  for sensor in sensores_temperatura if sensores_temperatura[sensor] < 22.0 ]
calefaccionar
```

]

02(

Bibliotecas externas

)

import bibliotecas

Hasta ahora, solamente hemos trabajado con las funciones estándares de Python, pero en ocasiones necesitaremos añadir algunas funcionalidades adicionales.

Podemos agregar algunas de éstas funcionalidades en cualquier lugar en donde instalemos Python, ya que se instalan por default con todas las herramientas de programación.

En el caso de otras herramientas, debemos instalarlas primero antes de comenzar a trabajar con ellas. Incluso podemos tener nuestra propia biblioteca de herramientas y cargarla aparte.

En cualquiera de éstos 3 casos, añadiremos funcionalidades con la palabra clave import

import bibliotecas

```
import math
```

Para acceder a los elementos de cada biblioteca, basta con usar el . para apuntar a cada uno de los elementos internos:

```
print(math.pi)
```

```
3.141592653589793
```

Una biblioteca puede tener objetos, como el caso de math.pi, como funciones. Podemos ver una lista de todas las funciones y elementos disponibles con la función help, o en la documentación online de cada biblioteca.

```
cateto1 = 4.0
cateto2 = 5.0
hipotenusa = math.hypot(cateto1, cateto2)
#otra manera interesante de escribir el valor de una variable dentro de un print:
print(f"Hipotenusa: {hipotenusa}")
```

import bibliotecas as libreria

Otra manera de importar una biblioteca es asignarle un alias

```
import math as mt
```

De ésta manera, podemos reemplazar todos los “math” en nuestro código por la forma más corta “mt”.

Aún así, debemos utilizar la sintaxis de punto para acceder a cada elemento:

```
print(mt.pi)
```

```
3.141592653589793
```

Y qué pasa si en realidad lo que queremos es acceder a cada uno de los elementos, sin tener que referirnos con . a la biblioteca de donde proceden? (por ejemplo, quiero usar pi directamente en lugar de math.pi o mt.pi)

from bibliotecas import *

La forma correcta de hacerlo es usar `from x import y`

```
from math import pi
pi
```

Podemos reemplazar `y` por el comodín `*` para importar **todas** las funciones y objetos de una biblioteca.

```
from math import *
pi
```

```
3.141592653589793
```

Cuidado! Si bien es cómodo importar con `*`, esto puede llevar a conflictos entre bibliotecas, especialmente si ésas bibliotecas tienen funciones con el mismo nombre (por ejemplo, la función *log* existe en `math` y `numpy`). Una buena práctica es sólo importar con `from` las funciones que necesitamos.

PIP

X

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

```
03(
    PIP
)
```

pip install externas [

Y... ¿Qué sucede cuando las bibliotecas que tenemos instaladas no nos alcanzan?

Ahí es donde entra en juego otra herramienta muy útil para instalar módulos externos: pip

pip es el sistema de gestión de paquetes utilizado en Python para instalar y administrar bibliotecas de terceros. Es una herramienta de línea de comandos que facilita la instalación, actualización y eliminación de paquetes Python.

```
# El nombre "pip" es un acrónimo de "Pip Installs Packages" (Pip Instala Paquetes). Es la forma principal de distribución de paquetes y módulos de Python desarrollados por la comunidad, y es ampliamente utilizado en el ecosistema de Python.
```

]

pip install externas [

Con pip, puedes instalar paquetes desde el Python Package Index (PyPI) y otros repositorios, lo que te permite acceder a una amplia gama de bibliotecas y herramientas desarrolladas por la comunidad. PyPI almacena miles de paquetes de Python disponibles para su instalación.

]

pip install externas [

Para utilizar pip, basta abrir cualquier consola (o escribir dentro de la consola disponible en nuestro IDE) e ingresamos la palabra clave pip seguido del comando que necesitemos para trabajar.

A continuación, una lista de comandos útiles:

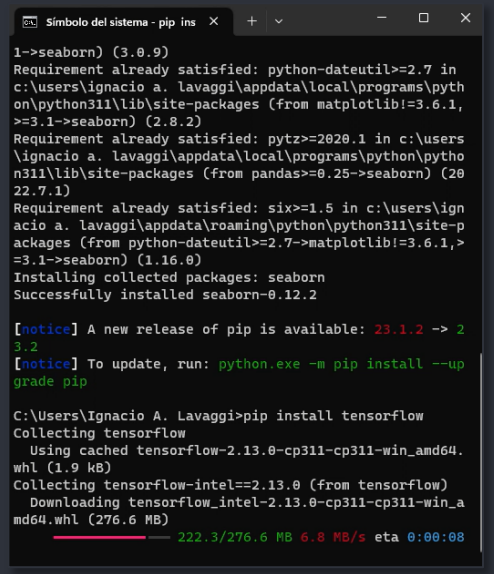
`pip install nombre_paquete`
Instala un paquete específico.

`pip uninstall nombre_paquete`
Elimina un paquete instalado.

`pip list`
Muestra una lista de los paquetes instalados en el entorno actual.

`pip show nombre_paquete`
Muestra información detallada sobre un paquete instalado, como la versión, la ruta de instalación, los requisitos y más.

`pip install --upgrade nombre_paquete`
Actualiza un paquete instalado a la última versión disponible.]



04(

Funciones

)

def funciones:

Al igual que en otros lenguajes de programación, las funciones son excelentes herramientas para encapsular y reutilizar fragmentos de código.

Todas las funciones que definamos en Python comienzan con el encablezado def, seguido del nombre de la función, sus argumentos entre paréntesis (si los hubiera) y dos puntos (:).

```
def mifuncion():
    print("tarea1")
    print("tarea2")
    print("tarea3")
```

Como vimos anteriormente, todo el bloque de código que pertenezca a esa función deberá estar indentado o tabulado en un nivel.

def funciones:

Para llamar a mi función, lo único que debo hacer es escribir su nombre y sus argumentos entre paréntesis.

```
mifuncion()
```

```
tarea1
tarea2
tarea3
```

def funciones:

Si quiero pasarle información a mi función para que ejecute una tarea en particular con esos datos, lo haremos a través de los mencionados argumentos, los cuales se definen al momento de definir la función. Voy a ubicar todos mis argumentos separados por coma.

```
def mifuncion(argumento):
    print("tarea1")
    print("tarea2")
    print("tarea3")
    print(argumento)
```

```
mifuncion("hola")
```

```
tarea1
tarea2
Tarea3
hola
```

```
mifuncion(2)
```

```
tarea1
tarea2
Tarea3
2
```

def funciones:

¡También puedo darle valores iniciales a mis argumentos! Estos reemplazarán sus valores si paso el argumento correspondiente, pero mantendrán su valor si no paso ninguno.

```
def mifuncion(argumento1="Default", argumento2 = "Default"):
    print("tarea1")
    print("tarea2")
    print("tarea3")
    print(argumento1)
    print(argumento2)
```

```
mifuncion("este es el arg1", "este es el arg2")
```

```
tarea1
tarea2
tarea3
este es el arg1
este es el arg2
```

```
mifuncion("este es el arg1")
```

```
tarea1
tarea2
tarea3
este es el arg1
Default
```

def funciones:

Como ven, todos los argumentos se escriben implícitamente en orden, por más que escribamos uno solo

```
mifuncion("este es el arg1", argumento2)
```

```
tarea1
tarea2
tarea3
este es el arg1
Default
```

Si queremos escribir un solo argumento, también podemos hacer uso de su nombre

```
mifuncion(argumento2 = “Solo el 2”)
```

```
tarea1
tarea2
tarea3
Default
Solo el 2
```

return:

La palabra clave return es una palabra clave asociada a las funciones. Cada vez que Python encuentra un return, sale de la función de inmediato y devuelve el valor a la derecha del return. Este valor puede ser un valor fijo, una variable, o un cálculo completo.

```
def sumar(a,b):
    return (a+b)
```

```
sumar(2,3):
```

```
5
```

Con todas las herramientas que hemos visto hasta ahora, podemos evaluar condiciones complejas, y hacer funciones que se adapten a nuestras necesidades:

return:

Por ejemplo, podemos evaluar con condicionales y devolver dos valores distintos :

```
def dividir(a,b):
    if b == 0:
        return("No se puede dividir por cero")
    else:
        return (a/b)
```

```
dividir(2,3):
```

```
0.6666666666666666
```

```
dividir(2,0):
```

```
No se puede dividir por cero
```

Documentación:

Si necesitamos documentar qué hace nuestra variable, podemos proveerle una descripción llamada “docstring”, escribiéndola entre **tres comillas** inmediatamente después de la definición principal

```
def dividir(a,b):
    """
    Uso:
    >>> dividir (dividendo,divisor)
    Retorna la división entre dos números, siempre y cuando el divisor sea distinto de cero
    """

    if b == 0:
        return("No se puede dividir por cero")
    else:
        return (a/b)
```

```
help(dividir)
```

```
Help on function dividir in module __main__:

dividir(a, b)
Uso: >>> dividir (dividendo,divisor)
Retorna la división entre dos números, siempre y cuando el divisor sea distinto de cero
```

Python.A_programar!

Ejercicios:

Fácil – Escalado de analógicas:

Escribe una función llamada "escalar_valores" que reciba un valor analógico como argumento y lo escale linealmente según un rango de entrada y un rango de salida específicos. El objetivo es transformar el valor analógico desde el rango de entrada al rango de salida.

valor_analogico: El valor analógico a escalar.
minimo_ing: El valor mínimo del rango de entrada. Default = 0
maximo_ing: El valor máximo del rango de entrada. Default = 32767
minimo: El valor mínimo del rango de salida. Default = 0.0
maximo: El valor máximo del rango de salida. Default = 100.0

Python.A_programar!

Ejercicios:

Intermedio – Verificador de temperaturas:

Hacer una función que tome, de una lista de valores, las temperaturas registadas en un día. De esa lista, debe retornar:

Promedio de temperaturas
La temperatura más baja
La temperatura más alta
Cantidad de temperaturas menores a 22°c

Hacer otra función que imprima los valores en un texto legible para el operador

```
Temperaturas1=[27.1, 22.3, 26.8, 23.5, 22.7, 15.3, 26.6, 16.9, 18.1, 24.7, 23.8, 18.4, 26.1, 27.5, 27.3, 21.9, 25.4, 25.1, 20.4, 16.2, 27.5, 22.7, 25.9, 21.2]
Temperaturas2=[25.4, 21.5, 27.3, 25.5, 20.2, 26.6, 16.1, 27.7, 26.4, 24.0, 22.6, 19.4, 27.0, 18.3, 25.0, 24.3, 25.6, 27.1, 15.6, 27.1, 26.6, 22.7, 20.4, 23.3]
Temperaturas3=[16.4, 20.5, 23.5, 17.3, 26.2, 26.2, 22.9, 21.2, 24.2, 26.0, 18.7, 27.5, 25.0, 22.7, 21.7, 22.7, 23.3, 25.0, 26.7, 18.7, 19.6, 23.9, 20.0, 17.2]
```