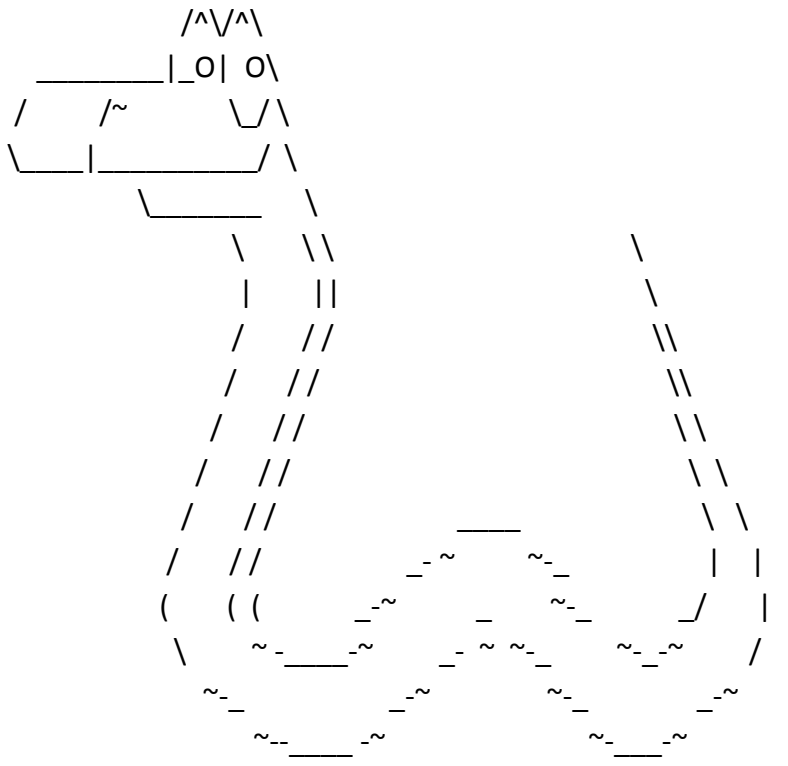
Bases de Python



El objetivo de este archivo es recopilar información de distintas bibliografías con el fin de formar un resumen el cual recopila las principales funciones y aplicaciones con las que python cuenta y cómo se pueden utilizar estas.

Cualquiera que desee aprender sobre Python será capaz de descargar este archivo. De igual manera la bibliografía utilizada para este este resumen estará destacada por capítulo y se recomienda leerla a medida que se desee. El objetivo es que este resumen cumpla con el mismo fin de esta.

Parte I: Bases - Sección I: Variables y tipos de datos simples

Variables

Toda **variable** está conectada a un valor, que es la información asociada con ese valor. En este caso se agrega la variable message la cual contiene el valor de texto “Hello world!”.

En Python, una variable es un nombre que se usa para almacenar un valor que puede cambiar o reutilizarse a lo largo del programa. Se define escribiendo el nombre de la variable, un signo igual (=) y el valor que quieres asignarle.

```
message = “Hello world!”
print(message)
```

```
>>> Hello world!
```

Añadir variables hace un poco más de trabajo para el interpretador de Python. Cuando procesa la última línea asocia la variable message con el primer valor otorgado. Al continuar con la ejecución del texto este utiliza el último valor que se le dio a la variable.

```
message = “Hello world!”
print(message)

message = “Hello Python!”
print(message)
```

```
>>> Hello world!
>>> Hello Python!
```

Nombrar y utilizar variables

Cuando se utilizan variables se suele seguir algunas reglas que, al romperlas, puede causar errores o dificultades a la hora de leer y entender el código escrito:

a) El nombre de la variable solo debe contener letras, números y guiones bajos. Puede comenzar con un guión bajo pero no con un número (puede ser message_1, pero no 1_message);
b) No se permiten espacios;
c) No se utilizan palabras clave de Python o nombre de funciones;
d) Deben ser nombres cortos y descriptivos del valor que almacenan.
e) De ser posible, las variables se escriben en minúsculas.

Strings

Un **string** es una serie de caracteres, Todo aquello que se encuentre dentro de unas comillas, dobles o simples, es considerado un string en Python.

---

texto_1 = “Esto es un string”
texto_2 = ‘Esto también’

Esta flexibilidad permite que se utilizan comillas y apóstrofos dentro de un string:

---

texto_1 = ‘Y le dije “Messi es mejor que Cristiano” ’
texto_2 = ‘...Cristiano es “Mejor” que Messi …’

print(texto_1)

>>> Y le dije “Messi es mejor que Cristiano”

Métodos de strings

Una de las tareas más simples es cambiar la mayúscula al principio del texto en la variable. Python puede manipular texto (representado por el tipo str, conocido como «cadenas de caracteres») al igual que números. Esto incluye caracteres «!», palabras «conejo», nombres «París», oraciones «¡Te tengo a la vista!», etc. «Yay! :)». Se pueden encerrar en comillas simples ('...') o comillas dobles ("...") con el mismo resultado

Para citar una cita, debemos «escapar» la cita procediéndose con \. Si no quieres que los caracteres precedidos por \ se interpreten como caracteres especiales, puedes usar cadenas sin formato agregando una **r** antes de la primera comilla. Alternativamente, podemos usar el otro tipo de comillas:

---

nombre_1 = ‘“ joaquin lopez” ’
print(r'C:\some\name')
print(nombre_1.title())

>>> “Joaquin lopez”

El método **title()** aparece luego de la variable en el print. Un método es una acción que Python puede ejecutar en una pieza de información. El punto (.) luego del nombre_1 le dice a Python que aplique el método title() en nombre_1. Los métodos generalmente llevan paréntesis luego de ser utilizados porque necesitan información adicional la cual va dentro del paréntesis. En el caso de title() la función no necesita información adicional.

También se puede cambiar un sting para que sea todo mayuscula y todo minuscula:

```
print(nombre_1.upper())
print(nombre_1.lower())
```

```
>>> JOAQUIN LOPEZ
>>> joaquin lopez
```

El método **lower()** es particularmente útil para almacenar data ya que quita las mayúsculas del texto

Utilizar variables en strings

En muchas situaciones, se utilizaran valores de variables dentro de un string, para lograr esto se coloca una f antes de abrir comillas y se colocan llaves **{}** donde se incluirá el nombre de la variable. Estos strings se llaman f-strings donde la f significa formato

---

```
nombre = “joaquín”
apellido = “lopez”
nombre_completo = f”{nombre} {apellido}”
print(f“Buen dia {nombre_completo .title()}”)
```

```
>>> Joaquin Lopez
```

También se pueden almacenar f-strings en variables

```
saludos = f“Buen dia {nombre_completo .title()}”
```

Añadir espacios en blanco con tabulaciones

En programación, espacios en blanco se refiere a caracteres no imprimibles, como espacios, tabulaciones, etc. Permite generar salidas que sean fáciles de leer, para añadir una tabulación a tu texto, utiliza \t:

```
print(“\tPython”)
```

```
>>> Python
```

Para añadir una nueva línea utiliza \n:

```
print(“Lenguajes:\n1 -Python\n2 - Francia\n3- C++”)
```

```
>>> Lenguajes
>>> 1 - Python
>>> 2 - Francia
>>> 3 - C++
```

Esto también se puede combinar:

```
print(“Lenguajes:\n1 -Python\n\t2 - Francia\n3- C++”)
```

```
>>> Lenguajes
>>> 1 - Python
>>> 2 - Francia
>>> 3 - C++
```

Las cadenas se pueden concatenar (pegar juntas) con el operador + y se pueden repetir con * y dos o más cadenas literales (es decir, las encerradas entre comillas) una al lado de la otra se concatenan automáticamente, también se puede concatenar variables o una variable y un literal, usa +:

```
3 * 'a' + 'las'
'a' 'las'

variable_de_texto = “a”
variable_de_texto + 'las'
```

```
>>> aaalas
>>> alas
>>> alas
```

Eliminar espaciado

Los espaciados pueden generar problemas a la hora de leer strings en Python. “python” y “python “ son considerados string diferentes. Python puede buscar por espacios extra tanto a la izquierda como a la derecha de un string utilizando el método **rstrip()**:

```
espacio_derecho = “python “
print(espacio_derecho.rstrip())
```

```
>>> python
```

Tambien se pued reutilizar lstrip para eliminar espaciado del lado izquierdo o strip si se quiere eliminar espaciado de ambos lados:

```
espacios = “ python “
print(espacios.lstrip())
print(espacios.lstrip())
```

```
>>>’python ‘
>>>’python‘
```

Quitar prefijos

Otra tarea común es remover prefijos. Cuando se trabaja con URL, un prefijo común es https://. Si se quiere remover esto para que se pueda utilizar la URL:

```
google_url= “https://google.com“
print(google_url.removeprefix("https://"))
```

```
>>> google.com
```

Se ingresa el nombre de la variable seguido de un punto seguido por el método removeprefix(). Dentro del método se ingresa el prefijo que se quiere remover del string original. Al igual que cuando se quitan los espacios extras, este método deja a la variable intacta por si se quiere utilizar su forma inicial

Métodos de strings

Método	Descripción	Ejemplo
lower()	Convierte el texto a minúsculas	"Hola".lower() ➡️ "hola"
upper()	Convierte el texto a mayúsculas	"hola".upper() ➡️ "HOLA"
capitalize()	Convierte la primera letra a mayúscula	"python".capitalize() ➡️ "Python"
title()	Convierte la primera letra de cada palabra a mayúscula	"hola mundo".title() ➡️ "Hola Mundo"
strip()	Elimina espacios en blanco al inicio y final	" hola ".strip() ➡️ "hola"
replace(a, b)	Reemplaza todas las apariciones de a por b	"hola mundo".replace("mundo", "Python") ➡️ "hola Python"
split(sep)	Divide el string en una lista usando el separador sep	"a,b,c".split(",") ➡️ ['a', 'b', 'c']
join(lista)	Une una lista de strings con un separador	",".join(['a', 'b', 'c']) ➡️ "a,b,c"
find(sub)	Devuelve la posición de la primera aparición de sub	"hola mundo".find("m") ➡️ 5
count(sub)	Cuenta cuántas veces aparece sub	"banana".count("a") ➡️ 3
startswith(sub)	Devuelve True si empieza con sub	"hola mundo".startswith("hola") ➡️ True
endswith(sub)	Devuelve True si termina con sub	"archivo.txt".endswith(".txt") ➡️ True
isalpha()	Devuelve True si solo contiene letras	"hola".isalpha() ➡️ True
isdigit()	Devuelve True si solo contiene dígitos	"123".isdigit() ➡️ True
islower()	True si todas las letras están en minúsculas	"hola".islower() ➡️ True
isupper()	True si todas las letras están en mayúsculas	"HOLA".isupper() ➡️ True
len()	(No es método, es función) Devuelve la longitud del string	len("hola") ➡️ 4

Números

Python trata los números de maneras distintas, dependiendo de cómo estén siendo utilizados. En primer lugar se miran a los integers ya que son los más simples para trabajar.

Integers

Los **integers** son valores donde se puede sumar (+), restar (-), multiplicar (*) y dividir (/). Cuando se escribe en la terminal python simplemente devuelve el resultado. A su vez python utiliza dos símbolos de multiplicación para representar exponentes y acompaña la orden de la operacion asi que se puede modificar el orden de la operación con paréntesis:

```
2 + 2
2 - 2
2 * 2
2 / 2
```

```
>>> 4
>>> 0
>>> 4
>>> 1
```

```
3 ** 2
2 + 3 * 4
(2 + 3) * 4
```

```
>>> 9
>>> 14
>>> 20
```

El espaciado no afecta en como Python evalúa las expresiones, solamente ayudan para la lectura de la misma.

Los números enteros (ej. 2, 4, 20) tienen tipo int, los que tienen una parte fraccionaria (por ejemplo 5.0, 1.6) tiene el tipo float. Vamos a ver más acerca de los tipos numéricos más adelante en el tutorial.

En el modo interactivo, la última expresión impresa se asigna a la variable _. Esto significa que cuando se está utilizando Python como calculadora, es más fácil seguir calculando.

```
tax = 12.5 / 100
price = 100.50
price * tax
price + _
round(_, 2)
```

```
>>> 12.5625
>>> 113.0625
>>> 113.06
```

Esta variable debe ser tratada como de sólo lectura por el usuario. No le asignes explícitamente un valor; crearás una variable local independiente con el mismo nombre enmascarando la variable con el comportamiento mágico.

Floats

Python llama a cualquier decimal **float**. Se llama float en casi todos los lenguajes para referirse a los valores en donde puede aparecer un punto decimal en cualquier posición del número. Hay que tener en cuenta que a veces se puede obtener un número arbitrario de decimales:

```
0.2 + 0.2
0.2 - 0.2
2 * 0.1
0.2 + 0.1
```

```
>>> 0.4
>>> 0.0
>>> 0.2
>>> 0.3000000000004
```

Python intenta representar los valores con la mayor exactitud posible lo que a veces puede ser difícil dado como los computadores tienen que representar los valores de forma interna. Además de int y float, Python admite otros tipos de números, como Decimal y Fracción. Python también tiene soporte incorporado para complejos números, y usa el sufijo j o J para indicar la parte imaginaria (por ejemplo, 3+5j).

Integers a Floats

Cuando se dividen 2 números, aunque estos sean integers, el resultado ser aún float:

```
4 / 2
```

```
>>> 2.0
```

Si estos se mezclan, el resultado también será un float:

```
1 + 2.0
```

```
>>> 3.0
```

Guiones bajos en números

Cuando se escriben números largos, se puede utilizar guinness bajos en lugar de puntos para hacer esto más legible. Cuando este se imprime, Python imprime solamente los dígitos:

```
numero_largo = 10_000_000_000_000_000
print(numero_largo)
```

```
>>> 10000000000000000
```

Python ignora los guiones bajos cuando se almacena este tipo de información, aunque este no se guarde en grupos de 3, el valor no se verá afectado.

Asignaciones múltiples

Se puede asignar valores a variables múltiples en una sola línea de código, ayudando a acortar la cantidad de código escrito y haciéndolo fácil de leer:

```
x, y, z = -1, 0, 1
```

Estos necesitan estar separados por comas, tanto las variables como los valores que se le otorgan a estas

Métodos de integers y floats

Función / Método	Descripción	Ejemplo
int(x)	Convierte un valor a entero (si es posible)	int(5.9) ➡️ 5
float(x)	Convierte un valor a decimal	float(5) ➡️ 5.0
abs(x)	Valor absoluto de un número	abs(-4) ➡️ 4
pow(x, y) o x**y	Eleva un número a la potencia de otro	pow(2, 3) ➡️ 8 / 2**3 ➡️ 8
round(x, n)	Redondea el número x a n decimales	round(3.14159, 2) ➡️ 3.14
divmod(a, b)	Devuelve una tupla (cociente, resto) de la división entera	divmod(10, 3) ➡️ (3, 1)
max(a, b, ...)	Devuelve el mayor valor	max(1, 4, 2) ➡️ 4
min(a, b, ...)	Devuelve el menor valor	min(1, 4, 2) ➡️ 1
sum(lista)	Suma todos los elementos de una lista	sum([1, 2, 3]) ➡️ 6
math.floor(x)	Redondea hacia abajo (requiere import math)	math.floor(3.7) ➡️ 3
math.ceil(x)	Redondea hacia arriba (requiere import math)	math.ceil(3.1) ➡️ 4
math.sqrt(x)	Raíz cuadrada (requiere import math)	math.sqrt(16) ➡️ 4.0
is_integer() (método)	Para float, devuelve True si el valor es un número entero	(3.0).is_integer() ➡️ True

Comentarios

Los comentarios son una parte fundamental de cualquier lenguaje de programación. A medida que los códigos se van tornando más largos y complicados, los comentarios facilitan la comprensión del mismo. Los comentarios son notas que describen lo que está sucediendo.

¿Cómo se escriben comentarios?

En Python,el numeral (#indica que es un comentario. Todo lo que se encuentre luego de un numeral es ignorado por el interpretado de Python:

```
# Tu nombre almacenado en una variable más arriba
print(nombre_1.title() )
```

```
>>> Joaquin lopez
```

¿Qué tipo de comentarios deberías escribir?

La principal razón para escribir comentarios es explicar qué hace tu código y cómo funciona. Cuando estás trabajando en un proyecto, entiendes cada parte, pero con el tiempo puedes olvidar detalles. Los comentarios te ayudan a recordar tu enfoque sin tener que volver a analizar todo.

Si quieres ser un programador profesional o colaborar con otros, es fundamental escribir comentarios claros y útiles, ya que la mayoría del software se desarrolla en equipo. Los buenos programadores esperan ver comentarios que expliquen el código.

Cuando dudes si deberías comentar algo, pregúntate si te llevó pensar varias soluciones antes de llegar a la correcta. Si fue así, deja un comentario explicándolo. Siempre es más fácil eliminar comentarios innecesarios después que tener que agregarlos más tarde.

A partir de ahora, el autor usará comentarios en sus ejemplos para explicar mejor el código.

"The Zen of Python"

Es un conjunto de principios y filosofías que guían el diseño y la escritura de código en Python. Fue escrito por Tim Peters y es una especie de “manifiesto” que describe cómo debería ser el código Python: claro, simple y elegante. Puedes verlo directamente escribiendo en la consola de Python:

```
import this
```

Algunos de los principios más destacados son:

- Lo bello es mejor que lo feo;
- Lo explícito es mejor que lo implícito;
- La simplicidad es mejor que la complejidad;
- La legibilidad cuenta;
- En caso de ambigüedad, rechaza la tentación de adivinar.

Parte I: Bases - Sección II: Introducción a las listas

¿Que es una lista?

Una lista es una colección de items en un orden particular. Se puede hacer una lista que incluya las letras del alfabeto o de números del 0 al 9. Una lista está indicada por corchetes [] y los elementos individuales están separados por comas (es una buena idea nombrar a la lista segun la informacion que contiene):

```
autos = [“nissan”,”chevrolet”,”ford”]
print(autos)
```

```
>>> [“nissan”,”chevrolet”,”ford”]
```

Al imprimir, Python devuelve la representación de la lista incluyendo los corchetes.

Acceder a los elementos de la lista

Las listas tienen un orden, así se puede acceder a los elementos que están en ellas diciéndole a Python la posición, o índice, del dato que necesitamos. Para acceder a los elementos de las listas hay que incluir la posición del dato entre corchetes comenzando desde el 0:

```
autos = [“nissan”,”chevrolet”,”ford”]
print(autos[0])
```

```
>>> [“nissan”]
```

Al preguntar por un solo dato, Python devolverá ese dato sin los corchetes. A esto se le puede aplicar los métodos de la sección I:

```
print(autos[0].title())
```

```
>>> Nissan
```

Posiciones de índice

Las posiciones comienzan desde el 0 en adelante. Esta es una característica de la mayoría de los lenguajes de programación y se debe a como las listas están indexadas basadas en cero (zero-based indexing)

Python tiene una sintaxis especial para acceder al último elementos de una lista, que es utilizando el -1

```
print(autos[-1].upper())
```

```
>>> FORD
```

Usar valores individuales de una lista

Se pueden utilizar valores individuales de una lista justo como cualquier otra variable. Por ejemplo en un f-string para crear un mensaje basado en una lista.

```
message = f”Mi primer auto fue un {autos[1].title()}.”
print(message)
```

```
>>> Mi primer auto fue un Chevrolet
```

Modificar, añadir y quitar datos

La mayoría de las listas que se crean son dinámicas, esto significa que luego de crearlas podrás añadir y quitar datos de la lista a medida que tu código se vaya ejecutando.

Modificar elementos de una lista

La sintaxis para modificar datos es similar a la sintaxis para acceder a ellos. Para esto se utiliza el nombre de la lista seguido por el índice del dato que se quiere modificar, y luego proveer un nuevo valor para ese dato:

```
autos_modificados = autos
autos_modificados [0] = “renault”
print(autos)
```

```
>>> [“renault”,”chevrolet”,”ford”]
```

¿Por qué los índices empiezan en 0?
Esto se debe a cómo los lenguajes como C (en el que Python está escrito) manejan la memoria. En términos de punteros, la dirección base de un array es la del primer elemento. Si el índice empezará en 1, habría que hacer un desplazamiento adicional en cada acceso, lo cual sería menos eficiente.

Por eso, la indexación basada en cero simplifica la aritmética de punteros y hace que los cálculos de posición sean más rápidos y consistentes.

Anteriormente este dato era “nissan” pero ahora es “renault” demostrando que la posición 0 de la lista fue modificada.

Añadir elementos a la lista

Existen muchos motivos por los cuales se querrán añadir métodos a una lista. Para esto Python posee varias maneras para lograr esto.

Añadir un elemento al final de la lista

La manera más simple de añadir un elemento es con el método append() el cual agrega un nuevo elemento al final de la lista.

```
agregar_auto = autos
agregar_auto.append(“ferrari”)
print(autos)
```

```
>>> [“renault”,”chevrolet”,”ford”,“ferrari”]
```

El método append hace fácil añadir elementos a las listas haciendo de estas listas más dinámicas. Por ejemplo, se puede iniciar con una lista vacía e ir añadiendo elementos a medida que el código se va ejecutando:

```
lista_de_autos_vacia = []

lista_de_autos_vacia.append(“ford”)
lista_de_autos_vacia .append(“ferrari”)

print(lista_de_autos_vacia )
```

```
>>> [”ford”,“ferrari”]
```

Esta manera de escribir listas es muy común ya que generalmente no se sabe la información que se cargará en el programa hasta después de que el programa esté corriendo, para eso se crea una lista vacía que irá reteniendo los datos.

Insertar datos en una lista

Se pueden añadir elementos en cualquier posición con el método insert(). Con este método se puede especificar en donde se busca incluir un dato en la lista:

```
autos.insert(0, “toyota”)
print(autos)
```

```
>>> [“toyota”,“renault”,”chevrolet”,”ford”]
```

Este método desplaza a los demás elementos una posición a la derecha desde el lugar en el cual se agregó.

Quitar datos de una lista

De igual manera que se buscan agregar datos, se busca quitar datos ya sea porque no se necesitan o quedan obsoletos. Python provee varias maneras de hacerlo:

Quitar un elemento con del

Se se sabe la posición del dato que se quiere quitar, se puede utilizar del statement:

```
del autos[0]
print(autos)
```

```
>>> [“renault”,”chevrolet”,”ford”,”ferrari”]
```

Ya que anteriormente añadimos “toyota” a la lista, ahora lo podemos quitar utilizando este método. Cambiando la posición que se le dio a del se quitara otro elemento de la lista, por ejemplo:

```
del autos[-1]
print(autos)
```

```
>>> [“renault”,”chevrolet”,”ford”]
```

Quitar un elemento utilizando el método pop()

En algunas ocasiones se buscará utilizar algún valor de la lista luego de quitarlo. El método pop() permite hacer esto. El término pop proviene de pesar en la lista como una pila de datos en donde se está quitando el dato de arriba.

```
popped_autos = autos.pop()
print(popped_autos)
```

```
>>> ford
```

Quitar un elemento utilizando el método pop() de cualquier lugar en la lista

Se puede utilizar el método pop() para quitar elementos en cualquier posición en la lista solo con incluir el índice del dato entre los paréntesis del metoodo:

```
mi_primer_auto = autos.pop(1)
print(f"Mi primer auto fue un {mi_primer_auto.title()}.")
```

>>> Mi primer auto fue un Chevrolet

Cabe recordar que cada vez que se utilice pop(), el elemento con el que se trabaja no se encontrará más en la lista.

Quitar un dato según el valor

Algunas veces no sabrás la posición del valor que se quiere quitar. Si solamente sabes el valor del elemento, puedes utilizar el método remove().

```
removed_autos = autos.remove("renault")
print(removed_autos)
```

>>> None

Organizar una lista

El hecho de ordenar o no una lista dependerá de para que será utilizada esta. Python provee de varias maneras distintas de organizar una lista, según cual sea el fin.

Organizar utilizando el método sort()

Es una manera sencilla de organizar una lista. Este método cambia el orden de la lista de manera permanente en orden alfabético, así que no podremos obtener en la lista original en caso de que la necesitemos.

El método remove() elimina solo la primera aparición del valor que especifiques. Si existe la posibilidad de que el valor aparezca más de una vez en la lista, necesitarás usar un bucle para asegurarte de que se eliminen todas las ocurrencias del valor.

```
autos.sort()
print(autos)
```

>>> ["chevrolet","ford","renault"]

Si se quiere ordenar alfabéticamente pero al revés se puede utilizar el argumento reverse=True:

```
autos.sort(reverse=True)
print(autos)
```

```
>>> [“renault”,”ford”,”chevrolet”]
```

Organizar una lista de forma temporal utilizando sorted()

Para mantener el orden de la lista original se utiliza el método sorted(), la cual te deja orden la lista en un orden particular soon cambiar el orden original:

```
print(sorted(autos))
print(autos)
```

```
>>> [”chevrolet”,”ford”,“renault”]
>>> [“renault”,”ford”,”chevrolet”]
```

Invertir una lista

Si se desea invertir los valores dentro de una lista se puede utilizar el método reverse():

```
autos.reverse()
print(autos)
```

```
>>> [”chevrolet”,”ford”,“renault”]
```

Este método no ordena de manera alfabética, cambiando el orden de manera permanente, aunque se puede utilizar el mismo método de nuevo y obtener el valor original.

Ordenar una lista alfabéticamente es un poco más complicado cuando no todos los valores están en minúsculas. Existen varias formas de interpretar las letras mayúsculas al determinar un orden de clasificación, y especificar el orden exacto puede ser más complejo de lo que queremos abordar en este momento. Sin embargo, la mayoría de los métodos de ordenación se basarán directamente en lo que aprendiste en esta sección.

Largo de listas

Se puede saber la longitud (por longitud hablamos de la cantidad de datos) de las listas utilizando la función len().

```
print(len(autos))
```

```
>>> 3
```

Python cuenta los datos comenzando por 1 ya que así se evitan errores de faltantes cuando se revisa los largos de la lista.

Evitar errores de indexado en las listas

Hay un tipo de error que es común ver cuando trabajas con listas por primera vez.

```
print(autos[3])
```

```
>>> Traceback (most recent call last):
      File "xxxxxxxxxx.py", line x, in
            print(motorcycles[3])
                  ~~~~~~~~~^^^
    IndexError: list index out of range
```

Python intenta darte el elemento en el índice 3. Pero cuando busca en la lista, no encuentra ningún elemento en motorcycles con un índice de 3. Debido a la naturaleza de la indexación basada en cero, este error es común. Las personas suelen pensar que el tercer elemento tiene el número 3, porque comienzan a contar desde 1. Sin embargo, en Python, el tercer elemento tiene el índice 2, ya que la indexación comienza en 0.

Un IndexError significa que Python no puede encontrar un elemento en el índice que solicitaste. Si este error ocurre en tu programa, intenta ajustar el índice restando uno y vuelve a ejecutar el programa para ver si los resultados son correctos.

Anexo Parte I: Bases - Sección II: Introducción a las listas

Python tiene varios tipos de datos compuestos, utilizados para agrupar otros valores. El más versátil es la lista, la cual puede ser escrita como una lista de valores separados por coma (ítems) entre corchetes. Las listas pueden contener ítems de diferentes tipos, pero usualmente los ítems son del mismo tipo.

```
squares = [1, 4, 9, 16, 25]
print(squares)
```

```
>>> [1, 4, 9, 16, 25]
```

Al igual que las cadenas (y todas las demás tipos integrados sequence), las listas se pueden indexar y segmentar:

```
print(squares[-3:]) # La segmentación devuelve una nueva lista
```

```
>>> [9, 16, 25]
```

Las listas también admiten operaciones como concatenación:

```
concatenado = squares + [36, 49, 64, 81, 100]
print(concatenado )
```

```
>>> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Todas las operaciones de rebanado retornan una nueva lista que contiene los elementos pedidos. Esto significa que la siguiente rebanada retorna una shallow copy de la lista:

```
squares + [36, 49, 64, 81, 100]
squares = squares [:]
```

```
>>> [9, 16, 25]
```

- Una copia superficial (shallow copy) construye un nuevo objeto compuesto y luego (en la medida de lo posible) inserta una referencias en él a los objetos encontrados en el original.
- Una copia profunda (deep copy) construye un nuevo objeto compuesto y luego, recursivamente, inserta copias en él de los objetos encontrados en el original.

Es posible anidar listas (crear listas que contengan otras listas), por ejemplo:

```
a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n]
print(x)
print(x[0])
print(x[0][1])
```

```
>>> [['a', 'b', 'c'], [1, 2, 3]]
>>> ['a', 'b', 'c']
>>> 'b'
```

Métodos de listas

Método	Descripción	Ejemplo
append(x)	Agrega un elemento al final de la lista.	lista.append(5)
extend(iterable)	Agrega múltiples elementos al final de la lista.	lista.extend([6, 7, 8])
insert(i, x)	Inserta un elemento en una posición específica.	lista.insert(1, 10)
remove(x)	Elimina la primera aparición de un elemento.	lista.remove(5)
pop([i])	Elimina y devuelve el elemento en la posición i. Si no se especifica, elimina el último.	lista.pop(2)
index(x, [start, end])	Devuelve la posición del primer elemento con el valor x.	lista.index(5)
count(x)	Devuelve la cantidad de veces que x aparece en la lista.	lista.count(5)
sort([key, reverse])	Ordena la lista en orden ascendente (o descendente si reverse=True).	lista.sort(reverse=True)
reverse()	Invierte el orden de los elementos de la lista.	lista.reverse()
copy()	Devuelve una copia superficial de la lista.	nueva_lista = lista.copy()
clear()	Elimina todos los elementos de la lista.	lista.clear()

Parte I: Bases - Sección III: Trabajar con listas

Bucle entre la lista entera

A veces es necesario realizar un bucle que realice un determinado proceso sobre todos y cada uno los datos que contiene una lista. En una lista de números, digamos los precios históricos de una acción, se buscará aplicar la misma operación estadística en cada elemento, o quizás mostrar distintos encabezados de noticias almacenados en una lista. Cuando se quiere realizar la misma acción con cada dato se utiliza el método for de Python.
Digamos que queremos hacer una cuenta regresiva:

```
números = [3,2,1]

for n in numeros:
        print(n)
```

```
>>> 3
>>> 2
>>> 1
```

Esa línea le indica a Python que tome un valor de la lista de números y lo asocie a n, luego le decimos que imprima una vez por cada elemento de la lista, ayuda leer este código: “Por cada número en la lista, imprime cada uno de los números”.

Una mirada más cercana

Hacer un bucle es la maneras más comunes en las que una computadora realiza automatiza tareas competitivas. En una lista, el bucle se repetirá por cada ítem que haya, si hay 10, 10_00, 10_000_000. También hay que tener en cuenta que Python nos permite asignar el nombre que queramos a la variable que se le asigna al valor del bucle.

```
for numero in numeros:
```

Es normal encontrar códigos que utilizan los nombres plurales y singulares de lo que se está realizando.

Evitar errores de tabulación

Python utiliza las tabulaciones para determinar la relación entre las líneas de código. Básicamente te exige utilizar tabulaciones para una estructura visual clara. En códigos de

Python más extensos, existen bloques de códigos que contienen varias tabulaciones. Estas tabulaciones ayudan a obtener un visualizaci general de la organización del programa

Hacer listas numéricas

En las visualizaciones de datos, casi siempre trabajarás con conjuntos de números, como temperaturas, distancias, tamaños de población o valores de latitud y longitud, entre otros tipos de conjuntos numéricos.

Las listas son ideales para almacenar conjuntos de números, y Python proporciona una variedad de herramientas para ayudarte a trabajar de manera eficiente con listas de números. Una vez que entiendas cómo utilizar estas herramientas de manera efectiva, tu código funcionará bien incluso cuando tus listas contienen millones de elementos.

Usar la función range()

La función range() hace fácil generar una serie de números. Por ejemplo se puede imprimir una serie de números:

```
for num in range(1,6):
```

```
>>> 1
>>> 2
>>> 3
>>> 4
>>> 5
```

Aunque parezca que el bucle imprimirá hasta el número 6, pero no es el caso. Esto se debe al comportamiento uno por uno (off-by-one) que se ve en los lenguajes de programación, lo que hace que Python cuente desde el primer valor otorgado, y frena antes del segundo valor otorgado.

Usar range() para hacer una lista

Si se quiere crear una lista de números, se puede convertir el resultado de range() en una utilizando la función list(). Cuando se coloca range() dentro de los paréntesis de list() el resultado será una lista:

```
numeros= list(range(1,6))
print(numeros)
```

```
>>> [1,2,3,4,5]
```

También se puede utilizar range() para saltarse números en un determinado rango. Dándole un tercer argumento, python usa el valar como un espaciador entre los valores:

```
numeros_impares= list(range(1,15,2))
print(numeros_impares)
```

```
>>> [1,3,5,7,9,11,13]
```

Supongamos que queremos saber los cuadrados de estos números:

```
cuadrados = []

for numero in range(1,15,2):
        cuadrado = numero ** 2
        cuadrados.append(cuadrado)

print(cuadrados)
```

```
>>> [1, 9, 25, 49, 81, 121, 169]
```

Se puede utilizar cualquiera de estos enfoques cuando crees listas más complejas. A veces, el uso de una variable temporal hace que tu código sea más fácil de leer; otras veces, lo hace innecesariamente largo.

Estadísticas simples con una lista numérica

Algunas funciones de Python resultan útiles para trabajar con una lista de números:

```
numeros = [1,5,2,8,9]
min(numeros)
max(numeros)
sum(numeros)
```

```
>>> 1
>>> 9
>>> 26
```

Comprensión de listas

Hay maneras de realizar el código de los cuadrados anteriores de manera más sencilla utilizando una sola línea de código. Una comprensión de listas te permite generar esta misma lista en una sola línea de código. Una comprensión de listas combina el bucle for y la creación de nuevos elementos en una sola línea, y agrega automáticamente cada nuevo elemento.

Las comprensiones de listas no siempre se presentan a principiantes, pero las he incluido aquí porque es muy probable que las encuentres tan pronto como empieces a ver el código de otras personas.

```
cuadrados = [numero**2 for numero in range(1,15,2)]
print(cuadrados)
```

Trabajar con partes de lista

Se puede trabajar con un grupo de datos específicos dentro de una lista, en Python este método se llama slice (rebanar).

Slicing

Para hacer un slice (rebanada) en una lista, debes especificar el índice del primer y el último elemento con los que quieres trabajar. Al igual que con la función range(), Python se detiene un elemento antes del segundo índice que especifiques. Por ejemplo, para obtener los tres primeros elementos de una lista, debes solicitar los índices del 0 al 3, lo que devolverá los elementos en las posiciones 0, 1 y 2.

```
numeros = [1,2,3,4,5]
print(numeros[0:3])
```

```
>>> [1,2,3]
```

Este código imprime una parte de la lista, donde se incluyen los primeros 3 valores de la lista. Se puede generar cualquier subconjunto:

```
print(numeros[1:3])
```

```
>>> [2,3]
```

Si se omite el primer índice o el último, Python automáticamente comienza el slice con el primer dato de la lista o termina en el último dato de la lista:

```
numeros = [1,2,3,4,5]
print(numeros[:3])
```

```
>>> [1,2,3]
```

Tamiones d puede utilizar un índice negativo devolviendo valores desde el final de la lista

```
print(numeros[-3:])
```

```
>>> [3,4,5]
```

Se puede incluir un tercer valor dentro de los corchetes al hacer un slice. Este tercer valor le indica a Python cuántos elementos debe saltar entre cada elemento dentro del rango especificado.

Copiar una lista

Para copiar una lista, puedes hacer un slice que incluya toda la lista original omitiendo el primer y el segundo índice ([:]). Esto le indica a Python que haga una rebanada que comience en el primer elemento y termine en el último, creando así una copia completa de la lista.

```
numeros = [1,2,3]
otros_numeros = numeros[:]
```

Tuplas

Las listas funcionan bien para almacenar colecciones de elementos que pueden cambiar a lo largo de la ejecución de un programa. La capacidad de modificar listas es especialmente importante cuando trabajas con una lista de usuarios en un sitio web o una lista de personajes en un juego. Sin embargo, en algunas ocasiones, querrás crear una lista de elementos que no puedan cambiar. Para esto, puedes usar tuplas. Python se refiere a los valores que no pueden cambiar como inmutables, y una lista inmutable se llama tupla.

Definir una tupla

Una tupla se ve como una lista, excepto que se utilizan paréntesis en lugar de corchetes. Una vez definida se puede acceder a los elementos individuales utilizando el índice justo como una lista.

Supongamos que tenemos un rectángulo al que no se le pueden cambiar la longitud de los lados, para ello definimos el rectángulo en una tupla:

```
dimensiones = (100,50)
print(dimensiones[0])
print(dimensiones[1])
```

```
>>> 100
>>> 50
```

Estos valores no se pueden cambiar ya que están definidos en una tupla lo que los hace inmutables.

Reescribir una tupla

Aunque no puedes modificar una tupla, sí puedes asignar un nuevo valor a una variable que representa una tupla. Por ejemplo, si quisieras cambiar las dimensiones de un rectángulo, podrías redefinir toda la tupla asignándole una nueva.

```
dimensiones = (100,50)

dimensiones = (200,50)
```

Anexo Parte I - Sección III: Trabajar con listas

for()

La sentencia for en Python difiere un poco de lo que uno puede estar acostumbrado en lenguajes como C o Pascal. En lugar de siempre iterar sobre una progresión aritmética de números (como en Pascal) o darle al usuario la posibilidad de definir tanto el paso de la iteración como la condición de fin (como en C), la sentencia for de Python itera sobre los ítems de cualquier secuencia (una lista o una cadena de texto), en el orden que aparecen en la secuencia

---

```
words = ['cat', 'window', 'defenestrate']
for w in words:
   print(w, len(w))
```

```
>>> cat 3
>>> window 6
>>> 'defenestrate' 12
```

range()

El valor final dado nunca es parte de la secuencia; range(10) genera 10 valores, los índices correspondientes para los ítems de una secuencia de longitud 10. Para iterar sobre los índices de una secuencia, puedes combinar range() y len() así:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
   print(i, a[i])
```

```
>>> 0 Mary
>>> 1 had
>>> 2 a
>>>3 little
>>> 4 lamb
```

Técnicas de iteración

Cuando iteramos sobre diccionarios, se pueden obtener al mismo tiempo la clave y su valor correspondiente usando el método items().

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
   print(k, v)
```

```
>>> gallahad the pure
>>> robin the brave
```

Cuando se itera sobre una secuencia, se puede obtener el índice de posición junto a su valor correspondiente usando la función enumerate().

```
for i, v in enumerate(['tic', 'tac', 'toe']):
   print(i, v)
```

```
>>> 0 tic
>>> 1 tac
>>> 2 toe
```

Para iterar sobre dos o más secuencias al mismo tiempo, los valores pueden emparejarse con la función **zip().**

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
   print('What is your {0}?  It is {1}.'.format(q, a))
```

```
>>> What is your name?  It is lancelot.
>>> What is your quest?  It is the holy grail.
>>> What is your favorite color?  It is blue.
```

Para iterar sobre una secuencia en orden inverso, se especifica primero la secuencia al derecho y luego se llama a la función **reversed().**

```
for i in reversed(range(1, 7, 2)):
   print(i)
```

```
>>> 5
>>> 3
>>> 1
```

Para iterar sobre una secuencia ordenada, se utiliza la función **sorted()** la cual retorna una nueva lista ordenada dejando a la original intacta.

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for i in sorted(basket):
   print(i)
```

```
>>> apple
>>> apple
>>> banana
>>> orange
>>> orange
>>> pear
```

El uso de **set()** en una secuencia elimina los elementos duplicados. El uso de **sorted()** en combinación con **set()** sobre una secuencia es una forma idiomática de recorrer elementos únicos de la secuencia ordenada.

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for f in sorted(set(basket)):
   print(f)
```

```
>>> apple
>>> banana
>>> orange
>>>pear
```

A veces uno intenta cambiar una lista mientras la está iterando; sin embargo, a menudo es más simple y seguro crear una nueva lista:

Métodos de iteraciones

Método	Descripción	Ejemplo
Iteración sobre una lista	Recorre los elementos de una lista.	for item in lista:
Iteración sobre un rango	Usa range() para generar una secuencia de números.	for i in range(5):
Iteración con índice	Usa enumerate() para obtener índices y valores.	for i, item in enumerate(lista):
Iteración sobre diccionarios	Recorre claves y valores de un diccionario.	for clave, valor in diccionario.items():
Iteración con zip()	Recorre varias listas al mismo tiempo.	for a, b in zip(lista1, lista2):
Iteración inversa	Usa reversed() para recorrer en orden inverso.	for item in reversed(lista):
Iteración sobre un conjunto (set)	Recorre elementos únicos de un set.	for item in mi_set:
Iteración sobre una cadena	Recorre cada carácter de un string.	for char in "Python":
Bucle con else	Ejecuta código si el for no se interrumpe con break.	for item in lista: ... else: ...

Parte I - Sección V: Trabajar con diccionarios

En Python, los diccionarios te permiten conectar piezas de información relacionada. Aprenderás cómo acceder a la información una vez que esté dentro de un diccionario y cómo modificarla. Dado que los diccionarios pueden almacenar una cantidad casi ilimitada de información, también verás cómo recorrer los datos con un bucle. Además, aprenderás a anidar diccionarios dentro de listas, listas dentro de diccionarios e incluso diccionarios dentro de otros diccionarios. Permiten describir objetos de la realidad de manera más precisa.

Un diccionario simple

```
plantas =  {“nombre”:”albahaca”,”altura”:”12 cm”}

print(plantas[“nombre”])
print(plantas[“altura”])
```

```
>>> albahaca
>>> 12 cm
```

El diccionario de plantas almacena el nombre de la planta y la altura que esta tiene.

Trabajar con diccionarios

Un diccionario en Python es una colección de pares clave-valor. Cada clave está asociada a un valor, y puedes usar la clave para acceder a su valor correspondiente. El valor de una clave puede ser un número, una cadena, una lista o incluso otro diccionario. De hecho, cualquier objeto que puedas crear en Python puede usarse como valor en un diccionario. En Python, un diccionario se define usando llaves {}, con una serie de pares clave-valor dentro de ellas. Un par clave-valor es un conjunto de valores asociados entre sí. Cuando proporcionas una clave, Python devuelve el valor correspondiente a esa clave. Cada clave está conectada a su valor mediante dos puntos :, y los pares clave-valor están separados por comas ,. Puedes almacenar tantos pares clave-valor como desees dentro de un diccionario.

Acceder a un valor

Para acceder a un valor asociado con un valor-clave, hay que llamar al diccionario y colocar el valor-clave dentro de los corchetes []:

```
print(plantas[“nombre”])
```

```
>>> albahaca
```

Añadir un nuevo valor clave

Para añadir un nuevo valor, definimos al mismo diccionario, y añadimos un nuevo valor clave con corchetes

---

plantas[“dias”] = 7

Modificar valores en un diccionario

Para modificar un valor en un diccionario, escribe el nombre del diccionario seguido de la clave entre corchetes [] y asigna el nuevo valor que deseas asociar a esa clave.

---

plantas[“nombre”] = “romero”

>>> albahaca

Borrar valores en un diccionario

Cuando ya no necesitas una información almacenada en un diccionario, puedes usar la instrucción del para eliminar por completo un par clave-valor.Solo necesitas especificar el nombre del diccionario y la clave que deseas eliminar.

---

del plantas[“dias”]

Usar get() para acceder a un valor

Para los diccionarios específicamente, puedes usar el método get() para establecer un valor predeterminado que se devolverá si la clave solicitada no existe. El método get() requiere una clave como primer argumento. Como segundo argumento opcional, puedes pasar el valor que se devolverá si la clave no existe.

---

altura = plantas.get(‘altura’, ‘No se tiene información de la altura.’)

En caso de no existir la altura, ésta devuelve un valor predefinido especificando que no existe valor para esa clave.

Hacer un bucle en un diccionario

En valores clave

Para escribir un bucle for para un diccionario, creas nombres para las dos variables que contendrán la clave y el valor de cada par clave-valor. Puedes elegir los nombres que quieras para estas dos variables. Este código funcionaría igual de bien si hubieras utilizado abreviaciones para los nombres de las variables.

```
student = {
   "nombre": "Alice",
   "edad": 22,
   "carrera": "Computer Science"
}

# Usando un bucle for para recorrer el diccionario
for k, vin student.items():
   print(f"{k}: {v}")
```

```
>>> "nombre": "Alice",
>>> "edad": 22,
>>> "carrera": "Computer Science"
```

La segunda parte de la declaración for incluye el nombre del diccionario seguido del método ítems(), que devuelve una secuencia de pares clave-valor. Luego, el bucle for asigna cada uno de estos pares a las dos variables proporcionadas.

Hacer un bucle entre todas las claves

El método keys() es útil cuando no necesitas trabajar con todos los valores de un diccionario, sino solo con sus claves.

```
for k student.keys():
        print(f"{k}")
```

```
>>> nombre
>>> edad
>>> carrera
```

Puedes optar por usar el método keys() explícitamente si hace que tu código sea más fácil de leer, o puedes omitir si lo prefieres.

Hacer un bucle entre todos los valores

Si estás principalmente interesado en los valores que contiene un diccionario, puedes usar el método values() para devolver una secuencia de valores sin las claves.

```
for k student.keys():
        print(f"{k}")
```

```
>>> Alice
>>> 22
>>> Computer Science
```

Anidar (Nesting)

A veces necesitarás almacenar múltiples diccionarios en una lista o una lista de elementos como valor en un diccionario. Esto se llama anidamiento (nesting).

Puedes anidar:

- Diccionarios dentro de una lista
- Listas dentro de un diccionario
- Diccionarios dentro de otros diccionarios

El anidamiento es una característica poderosa, como lo demostrarán los siguientes ejemplos.

Lista de diccionarios

Supongamos que tenemos los precios de cierre de 3 acciones distintas

---

```
accion_0 = {“ticker”:”AAPL”,”close”:100}
accion_1 = {“ticker”:”MMM”,”close”:90}
accion_2 = {“ticker”:”TTT”,”close”:80}

acciones = [accion_0, accion_1, accion_2]
```

Un diccionario dentro de un diccionario

Puedes anidar un diccionario dentro de otro diccionario, pero el código puede volverse complicado rápidamente al hacerlo.

```
users = {
        'aeinstein': {
                'first': 'albert',
                'last': 'einstein',
                'location': 'princeton',
                },

        'mcurie': {
                'first': 'marie',
                'last': 'curie',
                'location': 'paris',
                 },
        }

for username, user_info in users.items():
        print(f"\nUsername: {username}")
        full_name = f"{user_info['first']} {user_info['last']}", location = user_info['location']

        print(f"\tFull name: {full_name.title()}")
        print(f"\tLocation: {location.title()}")
```

Anexo Parte I - Sección V: Trabajar con diccionarios

Los diccionarios, también llamados matrices asociativas, deben su nombre a que son colecciones que relacionan una clave y un valor.

El primer valor se trata de la clave y el segundo del valor asociado a la clave. Como clave podemos utilizar cualquier valor inmutable: podríamos usar números, cadenas, booleanos, tuplas, … pero no listas o diccionarios, dado que son mutables. Esto es así porque los diccionarios se implementan como tablas hash, y a la hora de introducir un nuevo par clave-valor en el diccionario se calcula el hash de la clave para después poder encontrar la entrada correspondiente rápidamente. Si se modificara el objeto clave después de haber sido introducido en el diccionario, evidentemente, su hash también cambiaría y no podría ser encontrado. La diferencia principal entre los diccionarios y las listas o las tuplas es que a los valores almacenados en un diccionario se les accede no por su índice, porque de hecho no tienen orden, sino por su clave, utilizando de nuevo el operador [].

Al igual que en listas y tuplas también se puede utilizar este operador para reasignar valores.

Sin embargo en este caso no se puede utilizar slicing, entre otras cosas porque los diccionarios no son secuencias, si no mappings (mapeados, asociaciones).

En otros lenguajes, los diccionarios también se conocen como "memorias asociativas" o "arrays asociativos". A diferencia de las secuencias, que están indexadas por un rango de números, los diccionarios están indexados por claves, que pueden ser cualquier tipo inmutable (como cadenas y números).

Métodos de diccionarios

Método	Descripción	Ejemplo
dict.keys()	Devuelve una vista con todas las claves del diccionario.	d.keys() → dict_keys(['a', 'b', 'c'])
dict.values()	Devuelve una vista con todos los valores del diccionario.	d.values() → dict_values([1, 2, 3])
dict.items()	Devuelve una vista con pares (clave, valor).	d.items() → dict_items([('a',1),('b',2)])
dict.get(clave, valor_defecto)	Obtiene un valor dado su clave; si no existe, devuelve valor_defecto.	d.get('x', 0) → 0
dict.pop(clave, valor_defecto)	Elimina y devuelve el valor de una clave; si no existe, devuelve valor_defecto.	d.pop('a') → 1
dict.popitem()	Elimina y devuelve un par clave-valor aleatorio.	d.popitem() → ('b', 2)
dict.update(otro_dict)	Agrega o actualiza pares clave-valor desde otro diccionario.	d.update({'c': 5})
dict.setdefault(clave, valor_defecto)	Obtiene un valor o lo crea si la clave no existe.	d.setdefault('x', 10) → 10
dict.clear()	Elimina todos los elementos del diccionario.	d.clear() → {}
dict.copy()	Devuelve una copia superficial del diccionario.	d2 = d.copy()

Parte I - Sección IV: Condiciones If

Prueba condicionada

En el núcleo de cada declaración if hay una expresión que puede evaluarse como True o False, y se llama prueba condicional. Python usa estos valores para decidir si el código dentro de un if debe ejecutarse. Si la prueba condicional se evalúa como True, Python ejecuta el código dentro del if. Si se evalúa como False, Python ignora ese bloque de código.

Comprobar la igualdad

La mayoría de pruebas condicionales comparan el valor analizado con un valor de interés. El más simple de esto comprueba si el valor de la variable es igual al valor de interés.

---

auto = “bmw”
auto == “bmw”

>>> True

El operador de igualdad (==) devuelve True si los valores a la izquierda y a la derecha del operador coinciden, y False si no coinciden.

Ignorar mayúsculas cuando se prueba una igualdad

Dos valores con capitalizaciones diferentes se consideran distintas

---

auto == “Bmw”

>>> False

Si la diferencia entre mayúsculas y minúsculas es importante, este comportamiento es ventajoso. Pero si no importa y solo quieres comparar el valor de una variable, puedes convertir su valor a minúsculas antes de hacer la comparación.

Comprobar la desigualdad

Cuándo quieres determinar si dos valores no son iguales, puedes usar el operador de desigualdad (!=).

```
if auto != “bmw”:
        print(“No es una bemba”)
else:
        print(“Si es una bemba”)
```

>>> Si es una bemba

Si estos dos valores no coinciden, Python devuelve True y ejecuta el código que sigue al if. Si los valores coinciden, Python devuelve False y no ejecuta el código dentro del if.

Comparaciones numéricas

La comparación numérica es muy similar, por no decir igual.

```
años = 18 # Escribir con ñ no está mal pero no es lo habitual
años == 18
```

>>> True

Se pueden influir varias maneras de comprar a la prueba condicional:

```
if años < 18:
        print(“No tomes alcohol.”)
else:
        print(“Toma poquito.”)
```

>>> Toma poquito.

También puedes incluir diversas comparaciones matemáticas en tus declaraciones condicionales, como:

- Menor que (<)
- Menor o igual que (<=)
- Mayor que (>)
- Mayor o igual que (>=)

    Cada uno de estos se puede incluir como parte de las condiciones.

Comprobar condiciones múltiples

Usar and para comprobar condiciones múltiples

Para verificar si dos condiciones son verdaderas simultáneamente, usa la palabra clave and para combinar ambas pruebas condicionales.

- Si ambas pruebas son True, la expresión completa se evalúa como True.
- Si una o ambas pruebas son False, la expresión completa se evalúa como False.

```
año_0 = 18
año_1 = 22

año_0 => 18 and año_1 => 22
```

```
>>> True
```

Usar or para comprobar condiciones múltiples

La palabra clave or también permite verificar múltiples condiciones, pero la expresión será True si al menos una de las pruebas individuales es True. Una expresión con or solo será False cuando todas las pruebas individuales fallen (False).

```
año_0 => 18 or año_1 => 40
```

```
>>> True
```

Comprobar si un valor está en una lista

En Python, la palabra clave in se utiliza para verificar si un valor específico está presente en una lista antes de realizar una acción. Esto permite escribir código más eficiente y limpio, evitando operaciones innecesarias.

```
aderezos = [“mayonesa”,”mostaza”,”ketchup”]
“mayonesa” in aderezos
```

```
>>> True
```

La palabra clave in le dice a Python que busque el valor en la lista. En el caso de que queramos hacer lo contrario se utiliza not in:

```
aderezos = [“mayonesa”,”mostaza”,”ketchup”]
“bmw” not in aderezos
```

```
>>> True
```

Expresiones booleanas

A medida que aprendas más sobre programación, escucharás el término expresión booleana en algún momento. Una expresión booleana es simplemente otro nombre para una prueba condicional. Su resultado siempre es un valor booleano, que puede ser True (verdadero) o False (falso). Estas expresiones son fundamentales en la toma de decisiones dentro de un programa, ya que permiten ejecutar ciertas acciones en función de condiciones específicas. Los valores booleanos ofrecen una forma eficiente de rastrear el estado de un programa o una condición específica que sea importante en su funcionamiento.

Condición If

Cuando comprendes las pruebas condicionales, puedes comenzar a escribir sentencias if. Existen varios tipos de sentencias if, y la elección de cuál usar depende de la cantidad de condiciones que necesites evaluar. Ya has visto algunos ejemplos en la discusión sobre pruebas condicionales, pero ahora profundizaremos más en este tema.

La manera más simple de aplicar una condición if es:

```
if condición:
        acción
```

Puedes colocar cualquier prueba condicional en la primera línea de una sentencia if y casi cualquier acción dentro del bloque indentado que la sigue. Si la prueba condicional se evalúa como True, Python ejecutará el código dentro del bloque if. Si la prueba se evalúa como False, Python ignorará ese bloque y continuará con el resto del programa.

Condiciones if - else

A menudo, querrás realizar una acción cuando una prueba condicional se cumple y una acción diferente en todos los demás casos. La sintaxis if-else de Python permite hacer esto. Un bloque if-else es similar a una sentencia if simple, pero la declaración else define una acción o un conjunto de acciones que se ejecutan cuando la prueba condicional falla.

```
if años < 18:
        print(“No tomes alcohol.”)
else:
        print(“Toma poquito.”)
```

```
>>> Toma poquito.
```

Cadena if - elif - else

A menudo, necesitarás evaluar más de dos situaciones posibles, y para ello puedes usar la sintaxis if-elif-else de Python. En una cadena if-elif-else, Python ejecuta solo un bloque de código. Evalúa cada prueba condicional en orden y, cuando una de ellas se cumple, ejecuta el código correspondiente y omite el resto de las pruebas.

```
años = 19

if años < 18:
        print(“No tomes alcohol.”)
elif años == 18:
        print(“Desde ahora podes tomar”.)
else
        print(“Toma poquito.”)
```

```
>>> Toma poquito.
```

Se pueden utilizar todos los bloques de elif como sea necesario. El bloque else actúa como una declaración de respaldo, ya que captura cualquier condición que no haya sido satisfecha por una prueba if o elif. Sin embargo, esto puede incluir datos no válidos o incluso maliciosos. Si tienes una condición específica que deseas evaluar al final, es recomendable usar un bloque elif en lugar de else. De esta manera, puedes asegurarte de que tu código solo se ejecutará bajo las condiciones correctas, evitando resultados inesperados.

Anexo Parte I - Sección IV: Condiciones If

Tal vez el tipo más conocido de sentencia sea el if. Por ejemplo:

```
x = 40

if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
```

```
>>> More
```

Puede haber cero o más bloques elif, y el bloque else es opcional. La palabra reservada elif es una abreviación de “else if”, y es útil para evitar un sangrado excesivo. Una secuencia if … elif … elif … sustituye las sentencias switch o case encontradas en otros lenguajes.

Si necesitas comparar un mismo valor con muchas constantes, o comprobar que tenga un tipo o atributos específicos puede que encuentres útil la sentencia match.

Parte I - Sección V: Inputs de usuario y bucles While

Funcionamiento la función input()

La función input() pausa tu programa y espera a que el usuario ingrese algún texto. Una vez que Python recibe la entrada del usuario, asigna ese valor a una variable, lo que facilita su manipulación.

```
message = input("¿Cómo te llamas?: ")
print(message)
```

La función input() toma un argumento: el mensaje que queremos mostrar al usuario para que sepa qué tipo de información debe ingresar.

Usar int() para aceptar inputs numéricos

Cuando usas la función input(), Python interpreta todo lo que el usuario ingresa como una cadena de texto (string). Sabemos que Python ha interpretado la entrada como un string porque el número aparece entre comillas. Si solo quieres imprimir la entrada, esto no es un problema. Sin embargo, si necesitas trabajar con valores numéricos, puedes usar la función int(), que convierte la cadena en un valor numérico.

```
message = int(input("¿Cuántos años tienes?: "))
print(message)
```

El operador Modulo

Una herramienta útil para trabajar con información numérica es el operador módulo (%), que divide un número entre otro y devuelve el residuo de la división. El operador módulo (%) no te dice cuántas veces cabe un número dentro de otro, solo te da el residuo de la división. Cuando un número es divisible por otro, el residuo es 0, por lo que el operador módulo devuelve 0. Puedes usar este hecho para determinar si un número es par o impar:

- Si numero % 2 == 0, el número es par.
- Si numero % 2 != 0, el número es impar.

```
numero = int(input(“Par o Impar: ”))

if numero % 2 == 0:
        print(“Par”)
else:
        print(“Impar”)
```

Introducción a los bucles While

El bucle for toma una colección de elementos y ejecuta un bloque de código una vez por cada elemento de la colección. En contraste, el bucle while se ejecuta mientras una cierta condición sea verdadera.

Un bucle while permite ejecutar un bloque de código mientras se cumpla una condición. En el siguiente ejemplo, “número” comienza en 1 y el bucle se ejecuta mientras su valor sea menor o igual a 5. En cada iteración, se imprime el número actual y luego se incrementa en 1 con +=. Cuándo “número” supera 5, la condición deja de cumplirse y el bucle finaliza.

```
numero = 1
while numero <= 5:
        print(numero)
        numero += 1
```

```
>>> 1
>>> 2
>>> 3
>>> 4
>>> 5
```

Usar una bandera

En un programa que debe ejecutarse mientras varias condiciones sean verdaderas, se puede utilizar una variable llamada flag como señal de control. Este flag es una variable booleana que indica si el programa debe continuar (True) o detenerse (False). El bucle while verifica solo el estado del flag, y dentro del código se pueden realizar diferentes pruebas para cambiar su valor a False cuando ocurra un evento que deba detener la ejecución del programa. Esto permite organizar mejor las condiciones y mejorar la legibilidad del código.

```
prompt = “Di algo: ”

activo = True
while active:
        message = input(prompt)

        if message == ‘salir’:
                activo = False
        else:
                print(message)
```

Usar la declaración break

La declaración break permite salir de un bucle while de inmediato, sin importar las condiciones establecidas, deteniendo la ejecución del resto del código dentro del bucle. Esto otorga mayor control sobre el flujo del programa, asegurando que solo se ejecuten las instrucciones deseadas en el momento adecuado.

```
prompt = “Hola: ”

while True:
        message = input(prompt)

        if message == ‘salir’:
                brake
        else:
                print(message)
```

Usar continue en un bucle

En lugar de salir completamente de un bucle sin ejecutar el resto de su código, puedes usar la declaración continue para regresar al inicio del bucle, según el resultado de una prueba condicional.

```
numero = 0
while numero < 0:
        numero += 1
        if numero % 2 == 0:
                continue

        print(numero)
```

Usar bucles while en listas y diccionarios

Hasta ahora, hemos trabajado con una sola pieza de información del usuario a la vez, recibiendo la entrada y respondiendo a ella en cada iteración del bucle while. Sin embargo, para llevar un registro de múltiples usuarios y datos, es necesario usar listas y diccionarios junto con bucles while. Aunque un bucle for es útil para recorrer listas, no se recomienda modificar una lista dentro de uno, ya que Python podría tener dificultades para gestionar los elementos. En su lugar, los bucles while permiten modificar listas mientras se recorren, facilitando la recopilación, almacenamiento y organización de grandes cantidades de datos para analizarlos y reportarlos posteriormente.

Anexo Parte I - Sección V: Inputs de usuario y bucles While

Mientras que los condicionales nos permiten ejecutar distintos fragmentos de código dependiendo de ciertas condiciones, los bucles nos permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición.

```
edad = 0
while edad < 18:
   edad = edad + 1
   print("Felicidades, tienes " + str(edad))
```

La variable edad comienza con el valor 0. Como la condición edad < 18 es cierta (0 es menor que 18), se entra en el bucle.

Dentro del bucle, se incrementa edad en 1 y se imprime un mensaje informando que el usuario ha cumplido un año. Es importante recordar que el operador + para cadenas funciona concatenando ambas cadenas. Por esta razón, es necesario utilizar la función str() para convertir el número en una cadena, ya que no se pueden concatenar directamente números y cadenas.

A continuación, se vuelve a evaluar la condición. Como 1 < 18, el código se ejecuta nuevamente, aumentando la edad y mostrando el mensaje en pantalla. Este proceso continúa hasta que la edad sea igual a 18. En ese momento, la condición deja de cumplirse y el programa continúa con las instrucciones siguientes al bucle.

Ahora, imaginemos que olvidamos escribir la instrucción que aumenta la edad. En ese caso, la condición edad 18 siempre sería verdadera, porque la edad seguiría siendo 0. Esto provocaría que el bucle se ejecutará indefinidamente, escribiendo en pantalla "Has cumplido 0" sin detenerse.

Este fenómeno se conoce como un bucle infinito.

Sin embargo, hay situaciones en las que un bucle infinito es útil. Por ejemplo, veamos un pequeño programa que repite todo lo que el usuario diga hasta que escriba "adios":

```
while True:
    entrada = input("> ")
    if entrada == "adios":
        break
    else:
        print(entrada)
```

Para obtener lo que el usuario escribe en pantalla, utilizamos la función input(). No es necesario conocer en detalle qué es una función ni cómo funciona exactamente, pero por ahora podemos aceptar que, en cada iteración del bucle, la variable entrada contendrá el texto ingresado por el usuario hasta que pulse Enter.

Luego, comprobamos si el usuario escribió "adios". Si es así, se ejecuta la instrucción break, que termina el bucle. En caso contrario, se imprime en pantalla lo que el usuario escribió.

La palabra clave break (romper) permite salir del bucle en el que nos encontramos. Este mismo bucle también se podría haber escrito de la siguiente manera:

```
salir = False
while not salir:
    entrada = input("> ")
    if entrada == "adios":
        salir = True
    else:
        print(entrada)
```

Sin embargo, el primer enfoque nos ha servido para entender cómo funciona break.

Otra palabra clave que podemos encontrar dentro de los bucles es continue (continuar). Como su nombre indica, esta instrucción permite saltar directamente a la siguiente iteración del bucle sin ejecutar el resto del código en la iteración actual.

```
edad = 0
while edad < 18:
    edad = edad + 1
    if edad % 2 == 0:
        continue
    print("Felicidades, tienes " + str(edad))
```

Esta es una pequeña modificación de nuestro programa de felicitaciones. En esta ocasión, hemos añadido un if que comprueba si la edad es un número par. Si lo es, se ejecuta continue, lo que provoca que el bucle pase inmediatamente a la siguiente iteración sin imprimir el mensaje.

Es decir, con esta modificación, el programa solo imprimirá felicitaciones cuando la edad sea impar.

Parte I - Sección VII: Funciones

Cuando quieres realizar una tarea específica que has definido en una función, simplemente llamas a la función encargada de ejecutarla. Si necesitas realizar esa tarea varias veces a lo largo de tu programa, no es necesario escribir el mismo código repetidamente; solo llamas a la función correspondiente, y esa llamada le indica a Python que ejecute el código dentro de ella. Usar funciones hace que tus programas sean más fáciles de escribir, leer, probar y corregir.

Definir una función

La primera línea usa la palabra clave def para informar a Python que estás definiendo una función. Esta es la definición de la función, que le indica a Python el nombre de la función y, si es aplicable, qué tipo de información necesita para realizar su tarea.

```
def di_hola():
    “””
    Simplemente dice Hola.
    ”””
    print(“Hola.”)

di_hola()
```

```
>>> Hola.
```

El cuerpo de la función está compuesto por su contenido. El texto en la segunda línea es un comentario llamado docstring, que describe qué hace la función. Cuando Python genera documentación para las funciones de tus programas, busca una cadena de texto justo después de la definición de la función. Estas cadenas suelen estar encerradas entre triples comillas, lo que permite escribir varias líneas. La línea print("Hola.") es la única línea de código real dentro del cuerpo de esta función.

Darle información a la función

Se le puede agregar un variable a la función dentro del paréntesis la cual representa información que puede utilizar la función

```
def di_hola(usuario):
    “””
    Simplemente dice Hola.
    ”””
    print(f“Hola {usuario.title()}.”)

di_hola(“Toretto”)
```

```
>>> Hola toretto.
```

Argumentos y parámetros

La variable usuario es el ejemplo de un parámetro, que es un pedazo de información que la función necesita para hacer su trabajo, y el valor Toretto es el ejemplo de un argumento, que es información que se envía a la función al momento de ser llamada.

Enviar un argumento

Dado que una función puede tener múltiples parámetros en su definición, una llamada a la función puede necesitar varios argumentos. Puedes pasar argumentos de varias maneras: mediante argumentos posicionales, que deben seguir el mismo orden en que fueron definidos los parámetros; argumentos con palabras clave, donde cada argumento se especifica con un nombre de variable y un valor; y utilizando listas o diccionarios para pasar múltiples valores. Veamos cada uno de estos métodos en detalle.

Argumentos posicionales

Python debe asociar cada argumento de la llamada a la función con un parámetro en la definición de la función. La forma más sencilla de hacerlo es siguiendo el orden en el que se proporcionan los argumentos. Los valores emparejados de esta manera se conocen como argumentos posicionales.

```
def descipcion_de_plantas(tipo, altura):
    “””” Información de las plantas ”””
    print(”Características:”)
    print(f”{tipo.title()}”)
    print(f”{altura} cm”)

descipcion_de_plantas(“Albahaca”,13)
```

```
>>> Características:
>>> Albahaca
>>> 13 cm
```

Argumentos por palabras clave

Un argumento con palabra clave es un par nombre-valor que pasa a una función. Al especificar directamente el nombre y el valor dentro del argumento, no hay confusión al pasarlo a la función. Los argumentos con palabras clave evitan la preocupación de mantener un orden específico en la llamada a la función y ayudan a clarificar el propósito de cada valor.

```
descipcion_de_plantas(tipo="Albahaca", altura=13)
```

Valores por defecto

Al escribir una función, puedes definir un valor predeterminado para cada parámetro. Si en la llamada a la función se proporciona un argumento para ese parámetro, Python usará ese valor; de lo contrario, emplea el valor predeterminado. Esto permite omitir ciertos argumentos en la llamada a la función cuando no es necesario cambiarlos. El uso de valores predeterminados puede simplificar las llamadas a funciones y hacer más claro cómo se utilizan normalmente.

```
def descipcion_de_plantas(altura,tipo="Albahaca"):
        """ Información de las plantas """
        print("Características:")
        print(f"{tipo.title()}")
        print(f"{altura} cm")

descipcion_de_plantas(altura=14)
descipcion_de_plantas(14)
```

Es importante notar que el orden de los parámetros en la definición de la función tuvo que ser modificado. Dado que el valor predeterminado hace innecesario especificar el tipo de planta como argumento, el único argumento obligatorio en la llamada a la función es la altura. Python sigue interpretándolo como un argumento posicional, por lo que, si la función se llama solo con la altura de la planta, este argumento se asociará con el primer parámetro en la definición de la función. Por esta razón, el primer parámetro debe ser la altura.

Llamadas a funciones equivalentes

Dado que los argumentos posicionales, los argumentos con palabras clave y los valores predeterminados pueden combinarse, a menudo tendrás varias formas equivalentes de llamar a una función.

```
def descipcion_de_plantas(altura ,tipo="Albahaca")

def descipcion_de_plantas(tipo="Romero",altura=14)
def descipcion_de_plantas("Romero",14)
```

Valores return

Una función no siempre tiene que mostrar su salida directamente. En su lugar, puede procesar datos y luego devolver un valor o un conjunto de valores. El valor que devuelve la función se conoce como valor de retorno. La declaración return toma un valor dentro de la función y lo envía de vuelta a la línea que la llamó. Los valores de retorno permiten trasladar gran parte del trabajo pesado del programa a las funciones, lo que puede simplificar la estructura del código.

```
def nombre_del_musico(nombre, apellido):
        """Devuelve el nombre completo del artista """
        nombre_completo = f"{nombre} {apellido}"
        return nombre_completo.title()

musico = nombre_del_musico("David","Gilmour")
print(musico)
```

```
>>> David Gilmour
```

Hacer un argumento opcional

A veces tiene sentido hacer que un argumento sea opcional, permitiendo que quienes usen la función puedan proporcionar información adicional solo si lo desean. Para ello, se pueden usar valores predeterminados, lo que hace que el argumento sea opcional.

```
def nombre_del_artista(nombre, segundo_nombre, apellido):
        “””Devuelve el nombre completo del artista ”””
        if segundo_nombre:
                nombre_completo = f”{nombre} {segundo_nombre} {apellido}”
        else:
                nombre_completo = f”{nombre} {apellido}”
        return nombre_completo.title()

artista = nombre_del_artista(“jamie”,”lee”,”curtis”)
print(artista)
```

```
>>> Jamie Lee Curtis
```

Pasar una lista

A menudo es útil pasar una lista a una función, ya sea una lista de nombres, números o incluso objetos más complejos como diccionarios. Al pasar una lista a una función, esta obtiene acceso directo a su contenido. Usar funciones para trabajar con listas puede hacer el código más eficiente y organizado.

```
def saludar_usuarios(nombres):
   """Imprime un saludo simple para cada usuario en la lista."""
   for nombre in nombres:
      mensaje = f"¡Hola, {nombre.title()}!"
      print(mensaje)

usuarios = ['hannah', 'ty', 'margot']
saludar_usuarios(usuarios)
```

```
>>> ¡Hola, Hannah!
>>> ¡Hola, Ty!
>>> ¡Hola, Margot!
```

Pasar un número arbitrario de argumentos

A veces no sabrás de antemano cuántos argumentos necesitará aceptar una función. Afortunadamente, Python permite que una función recopile un número arbitrario de argumentos desde la llamada a la función. Esto se logra utilizando el operador *, que agrupa los argumentos en una tupla, permitiendo que la función maneje cualquier cantidad de valores de manera flexible.

```
def hacer_pizza(*ingredientes):
    """Imprime la lista de ingredientes solicitados."""
    for i in ingredientes:
        print(f" - {i}")

hacer_pizza('pepperoni')
hacer_pizza('champiñones', 'pimientos verdes', 'queso extra')
```

```
>>> - pepperoni
>>> - champiñones
>>> - pimientos verdes
>>> - queso extra
```

Mezclar argumentos posicionales y arbitrarios

Si deseas que una función acepte diferentes tipos de argumentos, el parámetro que recibe un número arbitrario de argumentos debe colocarse al final en la definición de la función. Python primero asigna los argumentos posicionales y argumentos con palabras clave, y luego recopila cualquier argumento restante en el último parámetro.

```
def hacer_pizza(tamano, *ingredientes):
    """Imprime la lista de ingredientes solicitados."""
    print(f"Preparando una pizza de tamaño {tamano} con los siguientes ingredientes:")
    for i in ingredientes:
        print(f" -{i}")

hacer_pizza('pepperoni')
hacer_pizza('champiñones', 'pimientos verdes', 'queso extra')
```

```
>>> Preparando una pizza de tamaño grande con los siguientes ingredientes:
>>> - pepperoni
>>> Preparando una pizza de tamaño mediana con los siguientes ingredientes:
>>> - champiñones
>>> - pimientos verdes
>>> - queso extra
```

Los dobles asteriscos ** antes de un parámetro hacen que Python cree un diccionario, que contendrá todos los pares clave-valor adicionales que reciba la función. Dentro de la función, puedes acceder a estos pares de la misma manera que lo harías con cualquier diccionario.

Almacenar las funciones en módulos

Una ventaja de las funciones es que permiten separar bloques de código del programa principal. Al usar nombres descriptivos para las funciones, los programas se vuelven mucho más fáciles de entender. Además, puedes ir un paso más allá almacenando tus funciones en un archivo separado llamado módulo y luego importándolo en el programa principal. La declaración import le indica a Python que haga disponible el código del módulo en el archivo que se está ejecutando.

Almacenar tus funciones en un archivo separado te permite ocultar los detalles del código de tu programa y centrarte en su lógica de alto nivel. También facilita la reutilización de funciones en diferentes programas. Cuando guardas tus funciones en archivos independientes, puedes compartirlos con otros programadores sin necesidad de proporcionar todo el programa. Además, saber cómo importar funciones te permite utilizar bibliotecas creadas por otros desarrolladores, lo que amplía las capacidades de tu código.

Importar un módulo entero

Para comenzar a importar funciones, primero necesitamos crear un módulo. Un módulo es simplemente un archivo que termina en .py y que contiene el código que deseas importar en tu programa. Al organizar funciones en módulos, puedes mantener tu código más limpio, reutilizable y fácil de mantener.

```
import python_modulo_I
```

También se pueden importar funciones específicas

```
from python_modulo_I import hacer_pizza
```

Usar as para darle un alias a la función o al módulo

Se le pueden dar alias a las funciones o módulos que importemos a nuestro código para hacer de este más legible.

```
from python_modulo_I import hacer_pizza as hp
```

Anexo Parte I - Sección VII: Funciones

Podemos crear una función que escriba la serie de Fibonacci hasta un límite determinado:

```
def fib(n):    # write Fibonacci series less than n
   """Print a Fibonacci series less than n."""
   a, b = 0, 1
   while a < n:
      print(a, end=' ')
      a, b = b, a+b
   print()
```

La palabra reservada def se usa para definir funciones. Debe seguirle el nombre de la función y la lista de parámetros formales entre paréntesis. Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente, y deben estar con sangría.

La primera sentencia del cuerpo de la función puede ser opcionalmente una cadena de texto literal; esta es la cadena de texto de documentación de la función, o docstring. (Puedes encontrar más acerca de docstrings en la sección Cadenas de texto de documentación.). Existen herramientas que usan las docstrings para producir documentación imprimible o disponible en línea, o para dejar que los usuarios busquen interactivamente a través del código; es una buena práctica incluir docstrings en el código que escribes, así que acostúmbrate a hacerlo.

La ejecución de una función introduce una nueva tabla de símbolos usada para las variables locales de la función. Más precisamente, todas las asignaciones de variables en la función almacenan el valor en la tabla de símbolos local; así mismo la referencia a variables primero mira la tabla de símbolos local, luego en la tabla de símbolos local de las funciones externas, luego la tabla de símbolos global, y finalmente la tabla de nombres predefinidos. Así, a variables globales y a variables de funciones que engloban a una función no se les puede asignar directamente un valor dentro de una función (a menos que se las nombre en la sentencia global, o mediante la sentencia nonlocal para variables de funciones que engloban la función local), aunque sí pueden ser referenciadas.

Los parámetros reales (argumentos) para una llamada de función se introducen en la tabla de símbolos local de la función llamada cuando ésta se llama; por lo tanto, los argumentos se pasan usando llamada por valor (donde el valor es siempre una referencia al objeto, no el valor del objeto). [1] Cuando una función llama a otra función, o se llama a sí misma de forma recursiva, se crea una nueva tabla de símbolos locales para esa llamada.

Una definición de función asocia el nombre de la función con el objeto de función en la tabla de símbolos actual. El intérprete reconoce el objeto al que apunta ese nombre como una función definida por el usuario.

La sentencia return retorna un valor en una función. return sin una expresión como argumento retorna None. Si se alcanza el final de una función, también se retorna None.

La sentencia result.append(a) llama a un método del objeto lista result. Un método es una función que “pertenece” a un objeto y se nombra obj.methodname, dónde obj es algún objeto (puede ser una expresión), y methodname es el nombre del método que está definido por el tipo del objeto. Distintos tipos definen distintos métodos. Métodos de diferentes tipos pueden tener el mismo nombre sin causar ambigüedad. (Es posible definir tus propios tipos de objetos y métodos, usando clases, ver Clases). El método append() mostrado en el ejemplo está definido para objetos lista; añade un nuevo elemento al final de la lista. En este ejemplo es equivalente a result = result + [a], pero más eficiente.

Part I - Sección VIII: Clases

La programación orientada a objetos (POO) es uno de los enfoques más efectivos para escribir software. En la programación orientada a objetos, escribes clases que representan cosas y situaciones del mundo real, y crear objetos basados en esas clases. Cuando escribes una clase, definen el comportamiento general que puede tener toda una categoría de objetos. Al crear objetos individuales a partir de la clase, cada objeto se equipa automáticamente con ese comportamiento general; luego puedes asignar a cada objeto los rasgos únicos que desees. Te sorprenderá lo bien que se pueden modelar situaciones del mundo real con programación orientada a objetos.

Crear y usar una clase

Definimos la clase perro la cual tendrá los atributos de nombre y edad, y también tendrá el comportamiento de sentarse y rodar.

```
class Perro:
    """Modelo de perro."""

    def __init__(self, name, age):
        """Comenzar con los atributos."""
        self.name = name
        self.age = age

    def sit(self):
        """Simulación del perro sentado."""
        print(f"{self.name} esta sentado.")

    def roll_over(self):
        """Simulación del perro rodar."""
        print(f"{self.name} rodo!")
```

El método __init__()

Una función que forma parte de una clase se llama método. Todo lo que aprendiste sobre funciones también se aplica a los métodos; la única diferencia práctica por ahora es cómo se llaman los métodos. El método __init__() es un método especial que Python ejecuta automáticamente cada vez que creamos una nueva instancia basada en la clase Perro. Este método tiene dos guiones bajos al principio y dos al final, lo cual es una convención que ayuda a evitar conflictos entre los nombres de métodos predeterminados de Python y los que puedas definir.

Definimos el método __init__() con tres parámetros: self, name y age. El parámetro self es obligatorio en la definición del método, y debe ir primero, antes que los demás parámetros.

Es necesario incluirlo porque cuando Python llama a este método más adelante (al crear una instancia de Perro), Python pasará automáticamente el argumento self. Cada vez que se llama a un método asociado con una instancia, self se pasa automáticamente. Este parámetro es una referencia a la instancia misma; le da acceso a los atributos y métodos definidos dentro de la clase. Cuando creamos una instancia de Perro, Python llamará al método __init__() de la clase Perro. Le pasaremos un nombre y una edad como argumentos. No es necesario pasar self, porqué Python lo maneja por su cuenta. Así que, siempre que queramos crear una instancia a partir de la clase Perro, solo necesitamos proporcionar los últimos dos parámetros: name y age.

Las variables definidas dentro del método __init__() con el prefijo self, como self.name y self.age, se conocen como atributos. Estos atributos están ligados a la instancia que se crea a partir de la clase y, por lo tanto, son accesibles desde cualquier método dentro de la clase, así como desde fuera de ella a través de la instancia. Por ejemplo, self.name = name toma el valor proporcionado al parámetro name y lo almacena como un atributo del objeto, lo que permite que ese valor esté disponible mientras exista la instancia. Esta estructura permite que cada objeto tenga su propio conjunto de datos, diferenciándose de otros objetos creados a partir de la misma clase.

Hacer una instancia para la clase

```
mi_perro = Perro(“Toby”, 4)

print(f”Mi perro se llama {mi_perro.name}.”)
print(f”Mi perro tiene {mi_perro.age} años.”)
```

Cuando Python lee esa línea, llama al método __init__() de la clase Perro con los argumentos Toby y 4. El método __init__() crea una instancia que representa a ese perro en particular y asigna los atributos name y age con los valores que proporcionamos. Luego, Python devuelve una instancia que representa a ese perro específico, ya configurada con su nombre y edad.

Acceder a los atributos

```
mi_perro.name
```

Este es el mismo atributo al que se hace referencia como self.name dentro de la clase Perro. Usamos el mismo enfoque para trabajar con el atributo age, es decir, asignándole dentro del método __init__() usando self.age, lo que permite que ambos atributos están directamente vinculados a la instancia del objeto y puedan ser accedidos desde cualquier parte del programa a través de esa instancia.

Utilizar métodos

Después de crear una instancia a partir de la clase Perro, podemos usar la notación de punto para llamar a cualquier método definido en Perro. Por ejemplo, si tenemos una instancia llamada mi_perro, podemos escribir mi_perro.sit() para llamar al método sit() o mi_perro.roll_over() para llamar a roll_over(). Esta notación permite acceder de forma clara y directa a los métodos y atributos de la instancia.

```
mi_perro.sit()
mi_perro.roll_over()
```

Trabajar con clases e instancias

Vamos a escribir una nueva clase que represente un auto. Nuestra clase almacenará información sobre el tipo de auto con el que estamos trabajando y tendrá un método que resuma esta información:

```
class Auto:
    """Un intento simple de representar un auto."""

    def __init__(self, marca, modelo, año):
        """Inicializa los atributos para describir un auto."""
        self.marca = marca
        self.modelo = modelo
        self.año = año

    def obtener_nombre_descriptivo(self):
        """Devuelve un nombre descriptivo con buen formato."""
        nombre_largo = f"{self.año} {self.marca} {self.modelo}"
        return nombre_largo.title()

# Crear una instancia y mostrar el nombre descriptivo
mi_auto_nuevo = Auto('audi', 'a4', 2024)
print(mi_auto_nuevo.obtener_nombre_descriptivo())
```

Poner un valor predeterminado a un atributo

Cuando una instancia es creada, los atributos pueden ser añadidos sin pasar un parámetro. Se añaden en __init__(). Añadimos el kilometraje con el que cuenta el auto comenzando en 0.

```
  def __init__(self, marca, modelo, año):
        """Inicializa los atributos para describir un auto."""
        self.marca = marca
        self.modelo = modelo
        self.año = año
        self.kilometraje = 0

def leer_kilometraje(self)
        """ Muestra los kilómetros que tiene el auto."""
        print(f"Este auto cuenta con {self.kilometraje} kilometros")

mi_auto_nuevo.leer_kilometraje()
```

Esta vez, cuando Python llama al método __init__() para crear una nueva instancia, almacena los valores de marca, modelo y año como atributos, tal como lo hizo en el ejemplo anterior. Estos atributos quedan vinculados al objeto que se está creando, lo que permite acceder a ellos más adelante desde cualquier parte del programa usando la instancia correspondiente.

Modificar el valor de los atributos

De manera directa

La forma más sencilla de modificar el valor de un atributo es acceder directamente a ese atributo a través de una instancia. Por ejemplo, si queremos establecer la lectura del odómetro en 23, simplemente lo hacemos así:

```
mi_auto_nuevo.kilometraje(23)
```

Esto asigna el valor 23 al atributo kilometraje, la instancia mi_auto_nuevo, incluso si no fue definido previamente en el método __init__(). Python lo crea dinámicamente en ese momento. También podés definirlo desde el inicio dentro de la clase para tener más control.

A través de un método

Puede ser útil tener métodos que actualicen ciertos atributos por vos. En lugar de acceder y modificar directamente el atributo desde fuera de la clase, se le pasa el nuevo valor a un método interno que se encarga de realizar la actualización. Esto permite tener mayor control sobre cómo y cuándo se modifican los atributos, y también podés incluir lógica adicional

```
def actualizar_kilometraje(self, kilometros):
        """Actualiza los kilómetros que tiene el auto."""
        self.kilometraje = kilometros

mi_auto_nuevo.kilometraje(56)
```

Incrementar un atributo a través de un método

A veces vas a querer incrementar el valor de un atributo en una cierta cantidad, en lugar de establecer un valor completamente nuevo. Por ejemplo, si comprás un auto usado y recorre 100 km antes de registrarlo, podrías sumar esos kilómetros al odómetro.

Acá tenés un método que permite pasar esa cantidad incremental y añadirla a la lectura actual del odómetro:

```
    def incrementar_kilometraje(self, km_extra):
        """Suma una cantidad dada al valor actual del odómetro."""
        self.kilometraje += km_extra
```

Herencia

No siempre tenés que empezar desde cero al escribir una clase. Si la clase que estás escribiendo es una versión especializada de otra clase que ya creaste, podés usar herencia. Cuando una clase hereda de otra, adopta los atributos y métodos de la primera.

La clase original se llama clase padre, y la nueva clase se llama clase hija. La clase hija puede heredar algunos o todos los atributos y métodos de su clase padre, pero también puede definir atributos y métodos propios para especializar su comportamiento.

El método __init__() para una clase hija

Por ejemplo, si tenés una clase general Auto, podés crear una clase hija llamada AutoElectrico que herede de Auto pero agregue características propias de un vehículo eléctrico:

```
class AutoElectrico(Auto):
    def __init__(self, marca, modelo, año):
        super().__init__(marca, modelo, año)  # Hereda atributos de Auto
        self.bateria = 75  # Atributo específico

    def descripcion_bateria(self):
        print(f"Este auto tiene una batería de {self.bateria} kWh.")

# Ejemplo
mi_tesla = AutoElectrico('Tesla', 'Model 3', 2024)
print(mi_tesla.obtener_descripcion())
mi_tesla.descripcion_bateria()
```

La función super() es una función especial que permite llamar a un método de la clase padre, le indica a Python que ejecute el método __init__() de la clase Car (o Auto en español), lo cual le da a la instancia de AutoElectrico todos los atributos definidos en ese método: marca, modelo y año.

El nombre super viene de la convención de llamar a la clase padre "superclase" y a la clase hija "subclase". Usar super() es fundamental cuando querés heredar comportamiento pero también agregar o modificar cosas en la subclase sin duplicar código.

Sobreescribir métodos de la clase padre

Podés sobrescribir cualquier método de la clase padre que no se ajuste a lo que estás intentando modelar con la clase hija. Para hacer esto, simplemente definís un método en la clase hija con el mismo nombre que el método que querés sobrescribir de la clase padre.

Cuando Python vea este método en la clase hija, ignorará el de la clase padre y solo usará el que definiste en la clase hija. Esto te permite personalizar el comportamiento de los objetos creados a partir de la clase hija sin tener que modificar la clase original.

```
class Auto:
    def __init__(self, marca, modelo, año):
        self.marca = marca
        self.modelo = modelo
        self.año = año

    def llenar_tanque(self):
        print("Este auto necesita llenar el tanque de combustible.")

class AutoElectrico(Auto):
    def llenar_tanque(self):  # Sobrescribimos el método
        print("Este auto no necesita un tanque de combustible.")
```

Instancia como atributo

Cuando modelas algo del mundo real en código, es común que empieces a agregar cada vez más detalles a una clase. Eso puede llevarte a tener una lista cada vez más larga de atributos y métodos, y tus archivos pueden volverse extensos y difíciles de mantener.

En estos casos, podés darte cuenta de que una parte de esa clase se puede escribir como una clase separada. Entonces, en lugar de seguir ampliando una sola clase, la dividía en clases más pequeñas que trabajan juntas. A este enfoque se le llama composición.

¿Qué es la composición?

La composición consiste en incluir una instancia de una clase dentro de otra clase. En lugar de heredar comportamiento, como en la herencia, se delegan ciertas responsabilidades a otra clase.

```
class Bateria:
    def __init__(self, capacidad=75):
        self.capacidad = capacidad

    def describir_bateria(self):
        print(f"Esta batería tiene una capacidad de {self.capacidad}-kWh.")

class AutoElectrico:
    def __init__(self, marca, modelo, año):
        self.marca = marca
        self.modelo = modelo
        self.año = año
        self.bateria = Bateria()  # Composición: incluye una instancia de Bateria

    def describir_auto(self):
        print(f"{self.marca} {self.modelo} ({self.año})")
        self.bateria.describir_bateria()
```

Importar clases

A medida que agregás más funcionalidad a tus clases, tus archivos pueden volverse largos, incluso cuando usás correctamente herencia y composición. Siguiendo la filosofía general de Python —que busca mantener el código limpio y legible—, lo ideal es mantener tus archivos lo menos recargados posible.

Para ayudarte con eso, Python te permite almacenar tus clases en módulos (archivos .py separados) y luego importar solo las clases que necesitás en tu programa principal. Esto mantiene tu código organizado, modular y fácil de mantener.

```
from auto import Auto # Importar una clase específica
from bateria import Bateria as bt # Importar una clase con un alias

import auto # Importar el módulo entero
mi_mustang = auto.Auto('ford', 'mustang', 2024)

from auto import * # Importar todas las clases
```

Anexo Part I - Sección VIII: Clases

Las clases proveen una forma de empaquetar datos y funcionalidad juntos. Al crear una nueva clase, se crea un nuevo tipo de objeto, permitiendo crear nuevas instancias de ese tipo. Cada instancia de clase puede tener atributos adjuntos para mantener su estado. Las instancias de clase también pueden tener métodos (definidos por su clase) para modificar su estado.

Unas palabras sobre nombres y objetos

Los objetos tienen individualidad, y múltiples nombres (en muchos ámbitos) pueden vincularse al mismo objeto. Esto se conoce como aliasing en otros lenguajes. Normalmente no se aprecia esto a primera vista en Python, y puede ignorarse sin problemas cuando se maneja tipos básicos inmutables (números, cadenas, tuplas). Sin embargo, el aliasing, o renombrado, tiene un efecto posiblemente sorpresivo sobre la semántica de código Python que involucra objetos mutables como listas, diccionarios, y la mayoría de otros tipos. Esto se usa normalmente para beneficio del programa, ya que los renombres funcionan como punteros en algunos aspectos. Por ejemplo, pasar un objeto es barato ya que la implementación solamente pasa el puntero; y si una función modifica el objeto que fue pasado, el que la llama verá el cambio; esto elimina la necesidad de tener dos formas diferentes de pasar argumentos, como en Pascal.

Sintaxis de definición de clases

Las definiciones de clases, al igual que las definiciones de funciones (instrucciones def) deben ejecutarse antes de que tengan efecto alguno. (Es concebible poner una definición de clase dentro de una rama de un if, o dentro de una función.)

En la práctica, las declaraciones dentro de una clase son definiciones de funciones, pero otras declaraciones son permitidas, y a veces resultan útiles; veremos esto más adelante. Las definiciones de funciones dentro de una clase normalmente tienen una lista de argumentos peculiar, dictada por las convenciones de invocación de métodos; a esto también lo veremos más adelante.

Cuando se ingresa una definición de clase, se crea un nuevo espacio de nombres, el cual se usa como ámbito local; por lo tanto, todas las asignaciones a variables locales van a este nuevo espacio de nombres. En particular, las definiciones de funciones asocian el nombre de las funciones nuevas allí.

Objetos clase

Los objetos clase soportan dos tipos de operaciones: hacer referencia a atributos e instanciación. Para hacer referencia a atributos se usa la sintaxis estándar de todas las referencias a atributos en Python: objeto.nombre. Los nombres de atributos válidos son todos los nombres que estaban en el espacio de nombres de la clase cuando ésta se creó.

Herencia

Por supuesto, una característica del lenguaje no sería digna del nombre «clase» si no soportara herencia. La ejecución de una definición de clase derivada procede de la misma forma que una clase base. Cuando el objeto clase se construye, se tiene en cuenta a la clase base. Esto se usa para resolver referencias a atributos: si un atributo solicitado no se encuentra en la clase, la búsqueda continúa por la clase base. Esta regla se aplica recursivamente si la clase base misma deriva de alguna otra clase.

Herencia múltiple

Para la mayoría de los propósitos, en los casos más simples, podés pensar en la búsqueda de los atributos heredados de una clase padre como una búsqueda en profundidad, de izquierda a derecha, sin repetir la misma clase cuando está dos veces en la jerarquía. En realidad es un poco más complejo que eso; el orden de resolución de métodos cambia dinámicamente para soportar las llamadas cooperativas a super(). Este enfoque es conocido en otros lenguajes con herencia múltiple como «llámese al siguiente método» y es más poderoso que la llamada al superior que se encuentra en lenguajes con sólo herencia simple.

Bibliografía:

Matthes, E. (2015). *Python crash course: A hands-on, project-based introduction to programming*. No Starch Press. ISBN 978-1-59327-603-4

Python Software Foundation. (2024). *The Python Tutorial*. Retrieved from https://docs.python.org/3/tutorial/

Python Cheat Sheet. (s.f.). *Lists and Tuples*. Recuperado el [fecha de acceso], de https://www.pythoncheatsheet.org/cheatsheet/lists-and-tuples

González Duque, R. (s.f.). *Python para todos*. Mundo Geek. Recuperado de http://mundogeek.net/tutorial-python/