# Cadenas de caracteres (String)

A los tipos de datos fundamentales (int, float, bool) se suma un tipo de dato muy utilizado que son las **cadenas de caracteres**.

Una cadena de caracteres está compuesta por uno o más caracteres. También podemos iniciar una cadena de caracteres por asignación o ingresarla por teclado.

**Inicialización de una cadena por asignación:**

```
#definición e inicio de una cadena de caracteres

dia="lunes"
```

Igual resultado obtenemos si utilizamos la **comilla simple**:

```
#definición e inicio de una cadena de caracteres

dia='lunes'
```

Para la carga por teclado de una cadena de caracteres utilizamos la función input que retorna una cadena de caracteres:

```
nombre=input("ingrese su nombre:")
```

Como su nombre lo indica, una cadena de caracteres está formada generalmente por varios caracteres (de todos modos podría tener sólo un caracter o ser una cadena vacía). Podemos acceder en forma individual a cada caracter del string mediante un subíndice:

```
nombre= 'juan'

print(nombre[0]) #se imprime una j

if nombre[0]=="j": #verificamos si el primer caracter del string es una j print(nombre)

print("comienza con la letra j")
```

**Los subíndices comienzan a numerarse a partir del cero.**

Si queremos conocer la longitud de un string en Python disponemos de una función llamada **len** que retorna la cantidad de caracteres que contiene:

```
nombre= 'juan'
```

```
print(len(nombre))
```

El código anterior imprime un 4 ya que la cadena nombre almacena 'juan' que tiene cuatro caracteres.

**Métodos propios de las cadenas de caracteres.**

Los string tienen una serie de métodos (funciones aplicables solo a los string) que nos facilitan la creación de nuestros programas.

Algunos métodos que veremos se llaman: lower, upper y capitalize.

**upper()** : devuelve una cadena de caracteres convertida todos sus caracteres a mayúsculas.

**lower()** : devuelve una cadena de caracteres convertida todos sus caracteres a minúsculas.

**capitalize()** : devuelve una cadena de caracteres convertida a mayúscula solo su primer caracter y todos los demás a minúsculas.

**Concatenación**

var1 = 'Hola' var2 = 'Python'

var3 = var1 + ' ' + var2

Para más información sobre cadenas de caracteres dirigirse a:
https://www.w3schools.com/js/js_strings.asp y
https://www.w3schools.com/js/js_string_methods.asp

**Fuente:**

http://www.tutorialesprogramacionya.com

https://j2logo.com

# Tipos de datos compuestos en Python

¿Qué tipos de datos trae consigo Python?

En Python podemos encontrar distintos tipos de datos con diferentes características y clasificaciones. Los tipos de datos básicos son los booleanos, numéricos (enteros, punto flotante, etc.) y las cadenas de caracteres.

Python también define otros tipos de datos, entre los que se encuentran:

Secuencias: list, tuple y range

Mapas: dict

Conjuntos: set

Clases

etc.

A su vez, los tipos anteriores se pueden agrupar de diferente manera. Por ejemplo, el tipo cadena de caracteres es una **secuencia inmutable** al igual que las tuplas. Las listas o diccionarios, entre otros, son contenedores y colecciones, etc. y ambos son mutables, es decir, pueden ser modificadas (se puede agregar, eliminar o modificar sus elementos en tiempo de ejecución).

En definitiva, los datos compuestos son el tipo opuesto a los tipos de datos atómicos, aquellos empleados para agrupar uno o más elementos. En Python veremos los siguientes:

Listas

Diccionarios

Tuplas

Conjuntos

En algunas ocasiones los datos compuestos se conocen también como datos o tipos agregados. Los tipos agregados son tipos de datos cuyos valores constan de colecciones de elementos de datos. Un tipo agregado se compone de tipos de datos previamente definitivos.

En resumen, además de los tipos básicos, otros tipos fundamentales de Python son las secuencias (list y tuple), los conjuntos (set) y los mapas (dict).

Todos ellos son tipos compuestos y se utilizan para agrupar juntos varios valores.

Las **listas** son secuencias mutables de valores.

Las **tuplas** son secuencias inmutables de valores.

Los **conjuntos** se utilizan para representar conjuntos únicos de elementos, es decir, en un conjunto no pueden existir dos objetos iguales.

Los **diccionarios** son tipos especiales de contenedores en los que se puede acceder a sus elementos a partir de una clave única (key).

**Fuente:** https://j2logo.com/python/tutorial/tipos-de-datos-basicos-de-python/

# Listas

En Python existe un tipo de variable que permite almacenar una colección de datos y luego acceder por medio de un subíndice (similar a los string).

**Creación de la lista por asignación**

Para crear una lista por asignación debemos indicar sus elementos encerrados entre corchetes y separados por coma.

lista1= [10, 5, 3] # lista de enteros

lista2= [1.78, 2.66, 1.55, 89,4] # lista de valores float

lista3= ["lunes", "martes", "miercoles"] # lista de string

lista4=["juan", 45, 1.92] # lista con elementos de distinto tipo (Python lo permite)

Si queremos conocer la cantidad de elementos de una lista podemos llamar a la función len:

lista1=[10, 5, 3]  # lista de enteros

print(len(lista1)) # imprime un 3

Una lista en Python es una **estructura mutable** (es decir puede ir cambiando durante la ejecución del programa).

Hemos visto que podemos definir una lista por asignación indicando entre corchetes los valores a almacenar:

**lista= [10, 20, 40]**

A una lista, luego de ser definida, podemos agregarle nuevos elementos a la colección. La primera forma que veremos para que nuestra lista crezca es utilizar el método **append** que tiene la lista y pasar como parámetro el nuevo elemento:

**lista= [10, 20, 30]**

**print(len(lista)) # imprime un 3**

**lista.append(100)**

**print(len(lista)) # imprime un 4**

**print(lista[0]) # imprime un 10**

**print(lista[3]) # imprime un 100**

Definimos una lista con tres elementos:

**lista=[10, 20, 30]**

Imprimimos la cantidad de elementos que tiene la lista:

**print(len(lista)) # imprime un 3**

Agregamos un nuevo elemento al final de la lista llamando al método append:

**lista.append(100)**

Si llamamos nuevamente a la función len y le pasamos el nombre de nuestra lista ahora retorna un 4:

**print(len(lista)) # imprime un 4**

Imprimimos ahora el primer y cuarto elemento de la lista (recordar que se numeran a partir de cero):

print(lista[0]) # imprime un 10

print(lista[3]) # imprime un 100

Lo que hace tan flexible a esta estructura de datos es que podemos almacenar componentes de tipo LISTA (listas de listas, también conocidas como matrices).

**notas= [[4,5], [6,9], [7,3]]**

En la línea anterior hemos definido una lista de tres elementos de tipo lista, el primer elemento de la lista es otra lista de dos elementos de tipo entero. De forma similar los otros dos elementos de la lista notas son listas de dos elementos de tipo entero.

**Listas: eliminación de elementos**

Hemos visto que una lista la podemos iniciar por asignación indicando sus elementos.

**lista= [10, 20, 30, 40]**

También podemos agregarle elementos al final mediante el método append:

**lista.append(120)**

Si ahora imprimimos la lista tenemos como resultado:

**[10, 20, 30, 40, 120]**

Otra característica fundamental de las listas en Python es que podemos eliminar cualquiera de sus componentes llamando al método **pop** e indicando la posición del elemento a borrar: **lista.pop(0)**

Ahora si imprimimos la lista luego de eliminar el primer elemento el resultado es:

**[20, 30, 40, 120]**

Otra cosa que hay que hacer notar que cuando un elemento de la lista se elimina no queda una posición vacía, sino se desplazan todos los elementos de la derecha una posición a izquierda.

El método pop retorna el valor almacenado en la lista en la posición indicada, aparte de borrarlo.

**lista=[10, 20, 30, 40]**

**print(lista.pop(0)) # imprime un 10**

Para más información sobre listas dirigirse a:
https://www.w3schools.com/js/js_arrays.asp,
https://www.w3schools.com/js/js_array_const.asp,
https://www.w3schools.com/js/js_array_methods.asp,
https://www.w3schools.com/js/js_array_sort.asp y
https://www.w3schools.com/js/js_array_iteration.asp

**Fuente:**

http://www.tutorialesprogramacionya.com

https://j2logo.com

# Tuplas

Hemos desarrollado gran cantidad de algoritmos empleando tipos de datos simples como enteros, flotantes, cadenas de caracteres y estructuras de datos tipo lista.

Vamos a ver otra estructura de datos llamada Tupla.

Una tupla permite almacenar una colección de datos no necesariamente del mismo tipo. Los datos de la **tupla** son **inmutables** a diferencia de las **listas** que son **mutables**.

Una vez inicializada la tupla no podemos agregar, borrar o modificar sus elementos.

La sintaxis para definir una tupla es indicar entre paréntesis sus valores:

```
tupla= (1, 2, 3)

fecha= (25, "Diciembre", 2016)

punto= (10, 2)

persona= ("Rodriguez", "Pablo", 43)

print(tupla)

print(fecha)

print(punto)

print(persona)
```

Como vemos el lenguaje Python diferencia una tupla de una lista en el momento que la definimos:

```
tupla= (1, 2, 3)

fecha= (25, "Diciembre", 2016)

punto= (10, 2)

persona= ("Rodriguez", "Pablo", 43)
```

Utilizamos paréntesis para agrupar los distintos elementos de la tupla.

Podemos acceder a los elementos de una tupla en forma similar a una lista por medio de un subíndice:

```
print(punto[0]) # primer elemento de la tupla

print(punto[1]) # segundo elemento de la tupla
```

Es muy IMPORTANTE tener en cuenta que los elementos de la tupla son **inmutables**, es incorrecto tratar de hacer esta asignación a un elemento de la tupla:

punto[0]=70

Nos genera el siguiente error:

Traceback (most recent call last):

File "...py", line 11, in punto[0]=70

TypeError: 'tuple' object does not support item assignment

Utilizamos una tupla para agrupar datos que por su naturaleza **están relacionados** y que **no serán modificados** durante la ejecución del programa.

**Empaquetado y desempaquetado de tuplas**

Podemos generar una tupla asignando a una variable un conjunto de variables o valores separados por coma:

x= 10

y= 30

punto= x,y

print(punto)

Tenemos dos variables enteras x e y. Luego se genera una tupla llamada punto con dos elementos.

fecha= (25, "diciembre", 2016)

print(fecha)

dd,mm,aa= fecha

print("Dia",dd)

print("Mes",mm)

print("Año",aa)

El desempaquetado de la tupla "fecha" se produce cuando definimos tres variables separadas por coma y le asignamos una tupla:

dd,mm,aa= fecha

Es importante tener en cuenta definir el mismo número de variables que la cantidad de elementos de la tupla.

**Tuplas anidadas**

Ahora que vimos tuplas también podemos crear tuplas anidadas.

En general podemos crear y combinar tuplas con elementos de tipo lista y viceversa, es decir listas con componente tipo tupla.

```
empleado= ["juan", 53, (25, 11, 1999)]

print(empleado)

empleado.append((1, 1, 2016))

print(empleado)

alumno= ("pedro",[7, 9])

print(alumno)

alumno[1].append(10)

print(alumno)
```

Por ejemplo definimos la lista llamada **empleado** con tres elementos: en el primero almacenamos su nombre, en el segundo su edad y en el tercero la fecha de ingreso a trabajar en la empresa (esta se trata de una tupla). Podemos más adelante, durante la ejecución del programa, agregar otro elemento a la lista con por ejemplo la fecha que se fue de la empresa:

```
empleado=["juan", 53, (25, 11, 1999)]

print(empleado)

empleado.append((1, 1, 2016))

print(empleado)
```

Tenemos definida la tupla llamada alumno con dos elementos, en el primero almacenamos su nombre y en el segundo una lista con las notas que ha obtenido hasta ahora:

```
alumno= ("pedro",[7, 9])

print(alumno)
```

Podemos, durante la ejecución del programa, agregar una nueva nota a dicho alumno:

```
alumno[1].append(10)

print(alumno)
```

Para más información sobre tuplas dirigirse a:
https://www.w3schools.com/python/python_tuples.asp

**Fuente:**

http://www.tutorialesprogramacionya.com

https://j2logo.com

# Diccionarios

Hasta ahora hemos presentado dos estructuras fundamentales de datos en Python: listas y tuplas. Ahora presentaremos y comenzaremos a utilizar la estructura de datos tipo diccionario.

La estructura de datos tipo diccionario utiliza una clave para acceder a un valor. El subíndice puede ser un entero, un float, un string, una tupla, etc. (en general cualquier tipo de dato **inmutable**).

Podemos relacionarlo con conceptos que conocemos: un diccionario tradicional por ejemplo. Podemos utilizar un “diccionario” de Python para representarlo. La clave sería la palabra y el valor sería la definición de dicha palabra.

Una agenda personal también la podemos representar como un diccionario. La fecha sería la clave y las actividades de dicha fecha sería el valor.

Un conjunto de usuarios de un sitio web podemos almacenarlo en un diccionario. El nombre de usuario sería la clave y como valor podríamos almacenar su mail, clave, fechas de último login, etc.

Hay muchos problemas de la realidad que se pueden representar mediante un diccionario de Python.

Recordemos que las listas son mutables y las tuplas inmutables. Un diccionario es una estructura de datos mutable es decir podemos agregar elementos, modificar y borrar.

## Definición de un diccionario por asignación

**productos= {"manzanas":39, "peras":32, "lechuga":17}**

**print(productos)**

Como vemos debemos encerrar entre llaves los elementos separados por coma. A cada elemento debemos indicar del lado izquierdo del caracter: la clave y al lado derecho el valor asignado para dicha clave. Por ejemplo, para la clave "peras" tenemos asociado el valor entero 32.

Para agregar elementos a un diccionario procedemos a asignar el valor e indicamos como subíndice la clave:

**nombre= input("Ingrese el nombre del producto:")**

**precio= int(input("Ingrese el precio:"))**

**productos[nombre]= precio**

Si ya existe el nombre del producto en el diccionario se modifica el valor para esa clave.

## Operador in con diccionarios

Para consultar si una clave se encuentra en el diccionario podemos utilizar el operador **in**:

```
if clave in diccionario:
    print(diccionario[clave])
```

Si no existe la clave produce un error al tratar de accederlo:

```
print(diccionario[clave])
```

Para más información sobre diccionarios dirigirse a:

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

**Fuente:**

http://www.tutorialesprogramacionya.com

https://j2logo.com

# Conjuntos

El tipo set en Python es la clase utilizada por el lenguaje para representar los conjuntos. Un conjunto es una colección desordenada de elementos únicos, es decir, que no se repiten.

A continuación veremos la clase set de Python, sus principales usos y operaciones.

## Qué es el tipo set en Python

El tipo set en Python es utilizado para trabajar con conjuntos de elementos. La principal característica de este tipo de datos es que es una colección cuyos elementos no guardan ningún orden y que además son únicos.

Estas características hacen que los principales usos de esta clase sean conocer si un elemento pertenece o no a una colección y eliminar duplicados de un tipo secuencial (list, tuple o str).

Además, esta clase también implementa las típicas operaciones matemáticas sobre conjuntos: unión, intersección, diferencia, etc.

Para crear un conjunto, basta con encerrar una serie de elementos entre llaves {}, o bien usar el constructor de la clase set() y pasarle como argumento un objeto iterable (como una lista, una tupla, una cadena, etc.).

```
# Crea un conjunto con una serie de elementos entre llaves
# Los elementos repetidos se eliminan
>>> c = {1, 3, 2, 9, 3, 1}
>>> c
{1, 2, 3, 9}

# Crea un conjunto a partir de un string
# Los caracteres repetidos se eliminan
>>> a = set('Hola Pythonista')
>>> a
{'a', 'H', 'h', 'y', 'n', 's', 'P', 't', ' ', 'i', 'l', 'o'}

# Crea un conjunto a partir de una lista
# Los elementos repetidos de la lista se eliminan
>>> unicos = set([3, 5, 6, 1, 5])
>>> unicos
{1, 3, 5, 6}
```

Para crear un conjunto vacío, simplemente llama al constructor set() sin parámetros.

**IMPORTANTE:** {} NO crea un conjunto vacío, sino un diccionario vacío. Usa set() si quieres crear un conjunto sin elementos.

## Cómo acceder a los elementos de un conjunto en Python

Dado que los conjuntos son colecciones desordenadas, en ellos no se guarda la posición en la que son insertados los elementos como ocurre en los tipos list o tuple. Es por ello que no se puede acceder a los elementos a través de un índice.

Sin embargo, sí se puede acceder y/o recorrer todos los elementos de un conjunto usando un bucle for:

```
>>> mi_conjunto = {1, 3, 2, 9, 3, 1}
>>> for e in mi_conjunto:
...     print(e)
...
1
2
3
9
```

## Añadir elementos a un conjunto (set) en Python

Para añadir un elemento a un conjunto se utiliza el método add(). También existe el método update(), que puede tomar como argumento una lista, tupla, string, conjunto o cualquier objeto de tipo iterable.

```
>>> mi_conjunto = {1, 3, 2, 9, 3, 1}
>>> mi_conjunto
{1, 2, 3, 9}

# Añade el elemento 7 al conjunto
>>> mi_conjunto.add(7)
>>> mi_conjunto
{1, 2, 3, 7, 9}

# Añade los elementos 5, 3, 4 y 6 al conjunto
# Los elementos repetidos no se añaden al conjunto
>>> mi_conjunto.update([5, 3, 4, 6])
>>> mi_conjunto
{1, 2, 3, 4, 5, 6, 7, 9}
```

**NOTA:** add() y update() no añaden elementos que ya existen al conjunto.

## Eliminar un elemento de un conjunto en Python

La clase set ofrece cuatro métodos para eliminar elementos de un conjunto. Son: discard(), remove(), pop() y clear(). A continuación te explico qué hace cada uno de ellos.

**discard(elemento)** y **remove(elemento)** eliminan elemento del conjunto. La única diferencia es que si elemento no existe, discard() no hace nada mientras que remove() lanza la excepción KeyError.

**pop()** es un tanto peculiar. Este método devuelve un elemento aleatorio del conjunto y lo elimina del mismo. Si el conjunto está vacío, lanza la excepción KeyError.

Finalmente, **clear()** elimina todos los elementos contenidos en el conjunto.

```
>>> mi_conjunto = {1, 3, 2, 9, 3, 1, 6, 4, 5}
>>> mi_conjunto
{1, 2, 3, 4, 5, 6, 9}

# Elimina el elemento 1 con remove()
>>> mi_conjunto.remove(1)
>>> mi_conjunto
{2, 3, 4, 5, 6, 9}

# Elimina el elemento 4 con discard()
>>> mi_conjunto.discard(4)
>>> mi_conjunto
{2, 3, 5, 6, 9}

# Trata de eliminar el elemento 7 (no existe) con remove()
# Lanza la excepción KeyError
>>> mi_conjunto.remove(7)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 7

# Trata de eliminar el elemento 7 (no existe) con discard()
# No hace nada
>>> mi_conjunto.discard(7)
>>> mi_conjunto
{2, 3, 5, 6, 9}

# Obtiene y elimina un elemento aleatorio con pop()
>>> mi_conjunto.pop()
2
>>> mi_conjunto
{3, 5, 6, 9}

# Elimina todos los elementos del conjunto
>>> mi_conjunto.clear()
>>> mi_conjunto
set()
```

## Cómo saber si un elemento está en un conjunto

Con los conjuntos también se puede usar el operador de pertenencia in para comprobar si un elemento está contenido, o no, en un conjunto:

```
>>> mi_conjunto = set([1, 2, 5, 3, 1, 5])
>>> print(1 in mi_conjunto)
True
>>> print(6 in mi_conjunto)
False
>>> print(2 not in mi_conjunto)
False
```

## Operaciones sobre conjuntos en Python (set operations)

**Unión de conjuntos en Python:** se utiliza el operador | para realizar la unión de dos o más conjuntos.

```
>>> a = {1, 2, 3, 4}
>>> b = {2, 4, 6, 8}
>>> a | b
{1, 2, 3, 4, 6, 8}
```

Intersección de conjuntos en Python: se utiliza el operador & para realizar la intersección de dos o más conjuntos.

Diferencia de conjuntos en Python: se utiliza el operador **-** para realizar la diferencia de dos o más conjuntos.

Inclusión de conjuntos en Python: se utiliza el operador <= para comprobar si un conjunto A es subconjunto de B y el operador >= para comprobar si un conjunto A es superconjunto de B.

Igualdad de conjuntos en Python: En Python dos conjuntos son iguales si y solo si todos los elementos de un conjunto están contenidos en el otro. Esto quiere decir que cada uno es un subconjunto del otro.

```
>>> a = {1, 2}
>>> b = {1, 2}
>>> id(a)
4475070656
>>> id(b)
4475072096
>>> a == b
True
```

Para más información sobre conjuntos dirigirse a:
https://www.w3schools.com/python/python_sets.asp

**Fuente:**

https://j2logo.com/python/tutorial/tipo-set-python/

# Rebanadas (Slicing)

**Porciones de listas, tuplas y cadenas de caracteres**

El lenguaje Python nos facilita una sintaxis muy sencilla para recuperar un trozo de una lista, tupla o cadena de caracteres.

Veremos con una serie de ejemplos cómo podemos rescatar uno o varios elementos de las estructuras de datos mencionadas.

```
lista1=[0,1,2,3,4,5,6]

lista2=lista1[2:5]

print(lista2) # 2,3,4

lista3=lista1[1:3]

print(lista3) # 1,2

lista4=lista1[:3]

print(lista4) # 0,1,2

lista5=lista1[2:]

print(lista5) # 2,3,4,5,6
```

Para recuperar una "porción" o trozo de una lista debemos indicar en el subíndice dos valores separados por el caracter "**:**".

Del lado izquierdo indicamos a partir de qué elementos queremos recuperar y del lado derecho hasta cual posición sin incluir dicho valor.

Por ejemplo con la sintaxis:

```
lista1=[0,1,2,3,4,5,6]

lista2=lista1[2:5]

print(lista2) # 2,3,4
```

Estamos recuperando de la posición 2 hasta la 5 sin incluirla.

También es posible no indicar alguno de los dos valores:

```
lista4=lista1[:3]

print(lista4) # 0,1,2
```

Si no indicamos el primer valor estamos diciendo que queremos recuperar desde el principio de la lista hasta la posición menos uno indicada después de los dos puntos.

En cambio si no indicamos el valor después de los dos puntos se recupera hasta el final de la lista:

**lista5=lista1[2:]**

**print(lista5) # 2,3,4,5,6**

Hay que tener en cuenta que el concepto de "porciones" se puede aplicar en forma indistinta a listas, tuplas y cadenas de caracteres.

**Valores negativos**

Ahora veremos que podemos utilizar un valor negativo para acceder a un elemento de la estructura de datos.

lista1=[0,1,2,3,4,5,6]

print(lista1[-1]) # 6

print(lista1[-2]) # 5

En Python podemos acceder fácilmente al último elemento de la secuencia indicando un subíndice -1:

print(lista1[-1]) # 6

Luego el anteúltimo se accede con la sintaxis:

print(lista1[-2]) # 5

Para más información sobre rebanadas dirigirse a:
https://www.w3schools.com/python/python_strings_slicing.asp

**Fuente:**

http://www.tutorialesprogramacionya.com

https://j2logo.com

# Operaciones con Strings, Listas, Tuplas, Diccionarios y Conjuntos

## Operaciones con cadenas y listas

```
cadena1 = 'tengo una yama que Yama se llama' # declara variable
lista1 = ['pera', 'manzana', 'naranja', 'uva']  # declara lista

longitud = len(cadena1) # 32, devuelve longitud de la cadena
elem = len(lista1)  # 4, devuelve nº elementos de la lista

cuenta = cadena1.count('yama')  # 1, cuenta apariciones de 'yama'
print(cadena1.find('yama')) # 10, devuelve posición de búsqueda

cadena2 = cadena1.join('***')   # inserta cadena1 entre caracteres
lista1 = cadena1.split(' ') # divide cadena por separador → lista
tupla1 = cadena1.partition(' ') # divide cadena por separador → tupla

cadena2 = cadena1.replace('yama','cabra',1) # busca/sustituye términos
numero = 3.14   # asigna número con decimales

cadena3 = str(numero)   # convierte número a cadena

if cadena1.startswith("tengo"): # evalúa si comienza por “tengo” (Falta definir el bl
oque)
    pass # Esta parte se puede omitir (es sólo para que no de error el código de ejem
plo que figura a continuación)
if cadena1.endswith("llama"):   # evalúa si termina por “llama” (Falta definir el blo
que)
    pass # Ídem anterior
if cadena1.find("llama") != -
1: # evalúa si contiene “llama” (Falta definir el bloque)
    pass

cadena4 = 'Python'  # asigna una cadena a una variable
print(cadena4[0:4]) # muestra desde la posición 0 a 4: "Pyth"
print(cadena4[1])   # muestra la posición 1: "y"
print(cadena4[:3] + '-' + cadena4[3:])  # muestra "Pyt-hon"

print(cadena4[:-3]) # muestra todo menos las tres últimas: "Pyt"

# declara variable con cadena alfanumérica
cadena5 = " abc;123 "

# suprime caracteres en blanco por la derecha
print(cadena5.rstrip()) # " abc;123"

# suprime caracteres en blanco por la izquierda
print(cadena5.lstrip()) # "abc;123 "

# suprime caracteres en blanco por derecha e izquierda
print(cadena5.strip())  # "abc;123"

cadena6 = "Mar" # declara una variable
print(cadena6.upper())  # convierte a mayúsculas: "MAR"
print(cadena6.lower())  # convierte a minúsculas: "mar"
```

## Operaciones con listas y tuplas

```
lista1 = ['uno', 2, True]  # declara una lista heterogénea
lista2 = [1, 2, 3, 4, 5]  # declara lista numérica (homogénea)
lista3 = ['nombre', ['ap1', 'ap2']]  # declara lista dentro de otra
lista4 = [54,45,44,22,0,2,99]  # declara una lista numérica
print(lista1)  # ['uno', 2, True], muestra toda la lista
print(lista1[0])  # uno, muestra el primer elemento de la lista
print(lista2[-1])  # 5, muestra el último elemento de la lista
print(lista3[1][0])  # calle, primer elemento de la lista anidada
print(lista2[0:3:1])  # [1,2,3], responde al patrón inicio:fin:paso
print(lista2[::-1])  # devuelve la lista ordenada al revés
lista1[2] = False  # cambia el valor de un elemento de la lista
lista2[-2] = lista2[-2] + 1  # 4+1 → 5 – cambia valor de elemento
lista2.pop(0)  # borra elemento indicado o último si no indica
lista1.remove('uno')  # borra el primer elemento que coincida
del lista2[1]  # borra el segundo elemento (por índice)
lista2 = lista2 + [6]  # añade elemento al final de la lista
lista2.append(7)  # añade un elemento al final con append()
lista2.extend([8, 9])  # extiende lista con otra por el final
lista1.insert(1, 'dos')  # inserta nuevo elemento en posición
del lista2[0:3]  # borra los elementos desde:hasta
lista2[:] = []  # borra todos los elementos de la lista
print(lista1.count(2))  # cuenta el nº de veces que aparece 2
print(lista1.index("dos"))  # busca posición que ocupa elemento
lista3.sort()  # ordena la lista
lista3.sort(reverse=True)  # ordena la lista en orden inverso
lista5 = sorted(lista4)  # ordena lista destino
tupla1 = (1, 2, 3)  # declara tupla (se usan paréntesis)...
tupla2 = 1, 2, 3  # ...aunque pueden declararse sin paréntesis
tupla3 = (100,)  # con un elemento hay terminar con “,”
tupla4 = tupla1, 4, 5, 6  # anida tuplas
tupla5 = ()  # declara una tupla vacía
tupla6 = tuple([1, 2, 3, 4, 5])  # Convierte una lista en una tupla
tupla2[0:3]  # (1, 2, 3), accede a los valores desde:hasta
```

## Operaciones con diccionarios

```
dic1 = {'Lorca':'Escritor', 'Goya':'Pintor'} # declara diccionario
print(dic1) # {'Goya': 'Pintor', 'Lorca': 'Escritor'}
dic2 = dict((('mesa',5), ('silla',10))) # declara a partir de tupla
dic3 = dict(ALM=5, CAD=10) # declara a partir de cadenas simples
dic4 = dict([(z, z**2) for z in (1, 2, 3)]) # declara a partir patrón
print(dic4)  # muestra {1: 1, 2: 4, 3: 9}
print(dic1['Lorca'])  # escritor, acceso a un valor por clave
print(dic1.get('Gala', 'no existe'))  # acceso a un valor por clave
if 'Lorca' in dic1: print('está')  # comprueba si existe una clave
print(dic1.items())  # obtiene una lista de tuplas clave:valor
print(dic1.keys())  # obtiene una lista de las claves
print(dic1.values())  # obtiene una lista de los valores
dic1['Lorca'] = 'Poeta'  # añade un nuevo par clave:valor
dic1['Amenabar'] = 'Cineasta'  # añade un nuevo par clave:valor
dic1.update({'Carreras' : 'Tenor'})  # añadir con update()
del dic1['Amenábar']  # borra un par clave:valor
print(dic1.pop('Amenabar', 'no está'))  # borra par clave:valor
```

## Recorrer secuencias y diccionarios con for...in

```
artistas = {'Lorca':'Escritor', 'Goya':'Pintor'} # diccionario
paises = ['Chile','España','Francia','Portugal'] # declara lista
capitales = ['Santiago','Madrid','París','Lisboa']  # declara lista
for c, v in artistas.items(): print(c,':',v)  # recorre diccionario
for i, c in enumerate(paises): print(i,':',c)  # recorre lista
for p, c in zip(paises, capitales): print(c,' ',p) # recorre listas
```

```
for p in reversed(paises): print(p,)  # recorre en orden inverso
for c in sorted(paises): print(c,)  # recorre secuencia ordenada
```

## Operadores para secuencias: in, not in, is, is not

```
cadena = 'Python'  # asigna cadena a variable
lista = [1, 2, 3, 4, 5]  # declara lista
if 'y' in cadena: print('"y" está en "Python"')  # contiene
if 6 not in lista: print('6 no está en la lista') # no contiene
if 'abcabc' is 'abc' * 2: print('Son iguales')  # son iguales
```

## Operaciones con conjuntos

```
conjunto = set()  # Define un conjunto vacío
lista = ['vino', 'cerveza', 'agua', 'vino']  # define lista
bebidas = set(lista)  # define conjunto a partir de una lista
print('vino' in bebidas)  # True, 'vino' está en el conjunto
print('anis' in bebidas)  # False, 'anis' no está en el conjunto
print(bebidas)  # imprime {'agua', 'cerveza', 'vino'}
bebidas2 = bebidas.copy()  # crea nuevo conjunto a partir de copia
print(bebidas2)  # imprime {'agua', 'cerveza', 'vino'}
bebidas2.add('anis')  # añade un nuevo elemento
print(bebidas2.issuperset(bebidas)) # True, bebidas es subconjunto
bebidas.remove('agua')  # borra elemento
print(bebidas & bebidas2)  # imprime elementos comunes
tapas = ['croquetas', 'solomillo', 'croquetas']  # define lista
conjunto = set(tapas)  # crea conjunto (sólo una de croquetas)
if 'croquetas' in conjunto:  # evalúa si croquetas está
    pass
conjunto1 = set('Python')  # define conjunto: P,y,t,h,o,n
conjunto2 = set('Pitonisa')  # define conjunto: P,i,t,o,n,s,a
print(conjunto2 - conjunto1)  # aplica diferencia: s, i, a
print(conjunto1 | conjunto2)  # aplica unión: P,y,t,h,o,n,i,s,a
print(conjunto1 & conjunto2)  # intersección: P,t,o,n
print(conjunto1 ^ conjunto2)  # diferencia simétrica: y,h,i,s,a
```

**Fuente:**

https://python-para-impacientes.blogspot.com/2014/01/cadenas-listas-tuplas-diccionarios-y.html

# Funciones en Python

Las funciones en Python, y en cualquier lenguaje de programación, son estructuras esenciales de código. Una función es un grupo de instrucciones que constituyen una unidad lógica del programa y resuelven un problema muy concreto.

## Qué es una función

Las funciones constituyen unidades lógicas de un programa y tienen un doble objetivo:

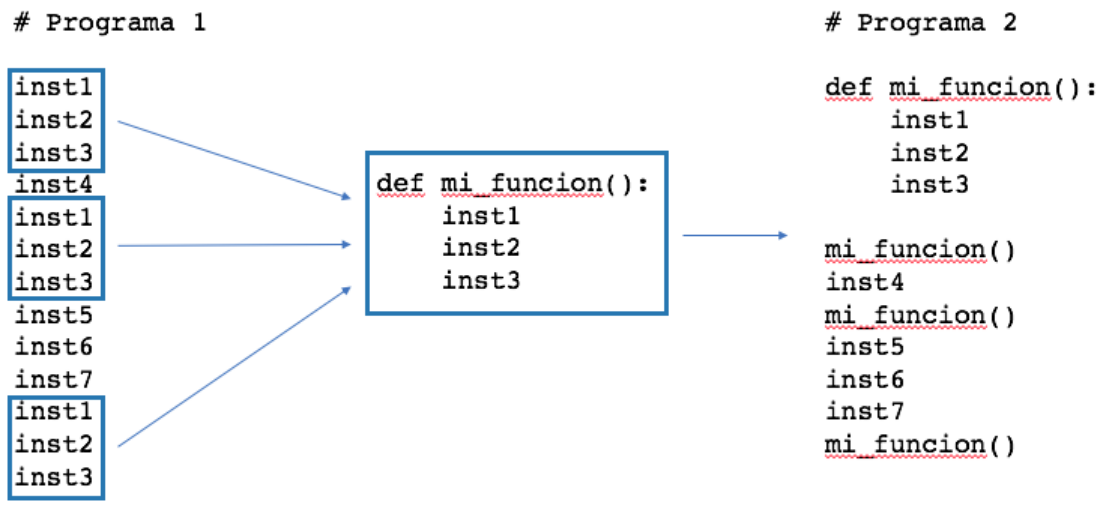
Dividir y organizar el código en partes más sencillas.

Encapsular el código que se repite a lo largo de un programa para ser reutilizado.

Python ya define de serie un conjunto de funciones que podemos utilizar directamente en nuestras aplicaciones. Por ejemplo, la función len(), que obtiene el número de elementos de un objeto contenedor como una lista, una tupla, un diccionario o un conjunto. También hemos visto la función print(), que muestra por consola un texto.

Sin embargo, tú puedes definir tus propias funciones para estructurar el código de manera que sea más legible y para reutilizar aquellas partes que se repiten a lo largo de una aplicación. Esto es una tarea fundamental a medida que va creciendo el número de líneas de un programa.

La idea la puedes observar en la siguiente imagen:




En principio, un programa es una secuencia ordenada de instrucciones que se ejecutan una a continuación de la otra. Sin embargo, cuando se utilizan funciones,

puedes agrupar parte de esas instrucciones como una unidad más pequeña que ejecuta dichas instrucciones y suele devolver un resultado.

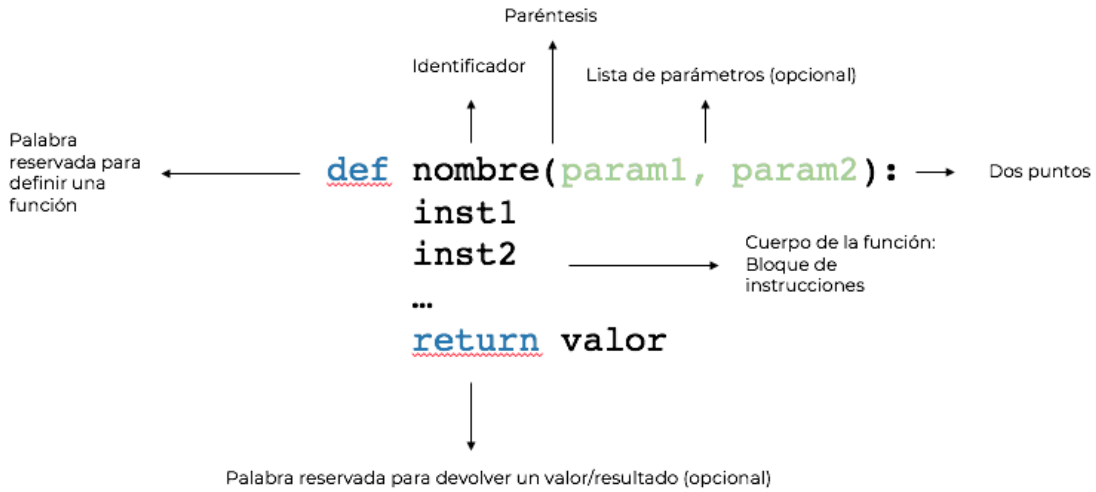
En el siguiente contenido veremos cómo definir una función en Python.

**Fuente:**

https://j2logo.com/python/tutorial/funciones-en-python/

# Cómo definir una función

La siguiente imagen muestra el esquema de una función en Python:




Para definir una función en Python se utiliza la palabra reservada def. A continuación viene el nombre o identificador de la función que es el que se utiliza para invocarla. Después del nombre hay que incluir los paréntesis y una lista opcional de parámetros. Por último, la cabecera o definición de la función termina con dos puntos.

Tras los dos puntos se incluye el cuerpo de la función (con un sangrado mayor, generalmente cuatro espacios) que no es más que el conjunto de instrucciones que se encapsulan en dicha función y que le dan significado.

En último lugar y de manera opcional, se añade la instrucción con la palabra reservada return para devolver un resultado.

**NOTA:** Cuando la primera instrucción de una función es un string encerrado entre tres comillas simples ''' o dobles """, a dicha instrucción se le conoce como docstring. El docstring es una cadena que se utiliza para documentar la función, es decir, indicar qué hace dicha función.

**Fuente:**

https://j2logo.com/python/tutorial/funciones-en-python/

# Cómo invocar o llamar a una función

Para usar o invocar a una función, simplemente hay que escribir su nombre como si de una instrucción más se tratara. Eso sí, pasando los argumentos necesarios según los parámetros que defina la función.

Veámoslo con un ejemplo. Vamos a crear una función que muestra por pantalla el resultado de multiplicar un número por cinco:

```
def multiplica_por_5(numero):
    print(f'{numero} * 5 = {numero * 5}')

print('Comienzo del programa')
multiplica_por_5(7)
print('Siguiente')
multiplica_por_5(113)
print('Fin')
```

La función multiplica_por_5() define un parámetro llamado numero que es el que se utiliza para multiplicar por 5. El resultado del programa anterior sería el siguiente:

```
Comienzo del programa
7 * 5 = 35
Siguiente
113 * 5 = 565
Fin
```

Como puedes observar, el programa comienza su ejecución en la línea 4 y va ejecutando las instrucciones una a una de manera ordenada. Cuando se encuentra el nombre de la función multiplica_por_5(), el flujo de ejecución pasa a la primera instrucción de la función. Cuando se llega a la última instrucción de la función, el flujo del programa sigue por la instrucción que hay a continuación de la llamada de la función.

**IMPORTANTE:** Diferencia entre parámetro y argumento. La función multiplica_por_5() define un parámetro llamado numero. Sin embargo, cuando desde el código se invoca a la función, por ejemplo, multiplica_por_5(7), se dice que se llama a multiplica por cinco con el argumento 7.

**Fuente:**

https://j2logo.com/python/tutorial/funciones-en-python/

# Sentencia return

Anteriormente se indicaba que cuando acaba la última instrucción de una función, el flujo del programa continúa por la instrucción que sigue a la llamada de dicha función. Hay una excepción: usar la sentencia return. **return** hace que termine la ejecución de la función cuando aparece y el programa continúa por su flujo normal.

Además, return se puede utilizar para devolver un valor.

La sentencia return es opcional, puede devolver, o no, un valor y es posible que aparezca más de una vez dentro de una misma función.

A continuación vemos varios ejemplos:

## return que no devuelve ningún valor

La siguiente función muestra por pantalla el cuadrado de un número solo si este es par:

```
>>> def cuadrado_de_par(numero):
...     if not numero % 2 == 0:
...         return
...     else:
...         print(numero ** 2)
...
>>> cuadrado_de_par(8)
64
>>> cuadrado_de_par(3)
```

## Varios return en una misma función

La función es_par() devuelve True si un número es par y False en caso contrario:

```
>>> def es_par(numero):
...     if numero % 2 == 0:
...         return True
...     else:
...         return False
...
>>> es_par(2)
True
>>> es_par(5)
False
```

## Devolver más de un valor con return en Python

En Python, es posible devolver más de un valor con una sola sentencia return. Por defecto, con return se puede devolver una tupla de valores. Un ejemplo sería la siguiente función cuadrado_y_cubo() que devuelve el cuadrado y el cubo de un número:

```
>>> def cuadrado_y_cubo(numero):
...     return numero ** 2, numero ** 3
...
>>> cuad, cubo = cuadrado_y_cubo(4)
>>> cuad
16
>>> cubo
64
```

Sin embargo, se puede usar otra técnica devolviendo los diferentes resultados/valores en una lista. Por ejemplo, la función tabla_del() que se muestra a continuación hace esto:

```
>>> def tabla_del(numero):
...     resultados = []
...     for i in range(11):
...         resultados.append(numero * i)
...     return resultados
...
>>> res = tabla_del(3)
>>> res
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

## En Python una función siempre devuelve un valor

Python, a diferencia de otros lenguajes de programación, no tiene procedimientos. Un procedimiento sería como una función pero que no devuelve ningún valor.

¿Por qué no tiene procedimientos si hemos vistos ejemplos de funciones que no retornan ningún valor? Porque Python, internamente, devuelve por defecto el valor None cuando en una función no aparece la sentencia return o esta no devuelve nada.

```
>>> def saludo(nombre):
...     print(f'Hola {nombre}')
...
>>> print(saludo('j2logo'))
Hola j2logo
None
```

Como puedes ver en el ejemplo anterior, el print que envuelve a la función saludo() muestra None.

**Fuente:**

https://j2logo.com/python/tutorial/funciones-en-python/

# Parámetros de las funciones en Python

Tal y como te hemos indicado, una función puede definir, opcionalmente, una secuencia de parámetros con los que invocarla. ¿Cómo se asignan en Python los valores a los parámetros? ¿Se puede modificar el valor de una variable dentro de una función?

Antes de contestar a estas dos preguntas, tenemos que conocer los conceptos de programación paso por valor y paso por referencia.

**Paso por valor:** un lenguaje de programación que utiliza paso por valor de los argumentos, lo que realmente hace es copiar el valor de las variables en los respectivos parámetros. Cualquier modificación del valor del parámetro, no afecta a la variable externa correspondiente.

**Paso por referencia:** un lenguaje de programación que utiliza paso por referencia, lo que realmente hace es copiar en los parámetros la dirección de memoria de las variables que se usan como argumento. Esto implica que realmente hagan referencia al mismo objeto/elemento y cualquier modificación del valor en el parámetro afectará a la variable externa correspondiente.

Muchos lenguajes de programación usan a la vez paso por valor y por referencia en función del tipo de la variable. Por ejemplo, paso por valor para los tipos simples: entero, float, etc. y paso por referencia para los objetos.

Sin embargo, en Python todo es un objeto. Entonces, ¿cómo se pasan los argumentos en Python, por valor o por referencia? Lo que ocurre en Python realmente es que se pasa por valor la referencia del objeto. ¿Qué implicaciones tiene esto? Básicamente que si el tipo que se pasa como argumento es **inmutable**, cualquier modificación en el valor del parámetro no afectará a la variable externa pero, si es **mutable** (como una lista o diccionario), sí se verá afectado por las modificaciones.

Para complementar esta explicación, no te pierdas las siguientes secciones.

**Fuente:**

https://j2logo.com/python/tutorial/funciones-en-python/

# Tipos de parámetros en una función Python

A la hora de definir una función, se puede indicar una serie de parámetros.

En el siguiente ejemplo, la función es_mayor devuelve si el parámetro x es mayor que el parámetro y:

```
def es_mayor(x, y):
    return x > y
```

Al invocar a la función, lo haremos del siguiente modo:

```
>>>es_mayor(5, 3)
True
```

Sin embargo, si al llamar a la función no pasamos todos los argumentos, el intérprete lanzará una excepción:

```
>>>es_mayor(5)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: es_mayor() missing 1 required positional argument: 'y'
```

Lo que nos está indicando es que el argumento posicional 'y' es obligatorio y no se ha especificado.

**IMPORTANTE:** Antes de seguir es importante que tengas en cuenta que, por defecto, los valores de los argumentos se asignan a los parámetros en el mismo orden en el que los pasas al llamar a la función. Más adelante veremos que esta circunstancia puede cambiar.

## Parámetros opcionales en una función Python

Además de como hemos visto hasta ahora, en una función Python se pueden indicar una serie de parámetros opcionales. Son parámetros que se indican con un valor por defecto y si no se pasan al invocar a la función entonces toman este valor.

**NOTA:** este ejemplo utiliza el concepto de objetos. Si aún no estás familiarizado, tal vez debas dejar este apartado pendiente y volver luego de ver ese tema en el material disponible. Sin embargo, esto podría ser una buena aproximación a un tema próximo a venir.

Imaginemos el constructor (método __init__) de una clase Punto que toma los valores de las coordenadas en las que se creará un objeto de dicha clase:

```
class Punto:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return "x: {}, y: {}".format(self.x, self.y)
```

Como vemos, en el método __init__ se indican dos parámetros: la coordenada x y la coordenada y de un punto:

```
>>>Punto(1, 2)
x: 1, y: 2
```

Según se indicó en el apartado anterior, si llamamos a la función sin pasar ningún argumento nuestro código fallará:

```
>>>Punto()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: __init__() missing 2 required positional arguments: 'x' and 'y'
```

Podemos mejorar nuestra función __init__ de manera que si no se indica alguno de los parámetros, entonces tome el valor por defecto 0. Para indicar un parámetro de forma opcional se usa el operador '='. Veamos cómo quedaría:

```
def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
```

Ahora podemos invocar a la función del siguiente modo:

```
>>>Punto()
x: 0, y: 0

>>>Punto(3)
x: 3, y: 0
```

Consideremos a continuación la siguiente función saludo:

```
def saludo(nombre, mensaje="encantado de saludarte"):
    print("Hola {}, {}".format(nombre, mensaje))
```

El parámetro nombre no indica un valor por defecto, por tanto, es obligatorio. No ocurre lo mismo con el parámetro mensaje, cuyo valor por defecto es «encantado de saludarte». En caso de no pasar este argumento, se tomará dicho valor por defecto. Por el contrario, si se indica, se sobreescribirá con el nuevo valor.

En una función se pueden especificar tantos parámetros opcionales como se quiera. Sin embargo, una vez que se indica uno, todos los parámetros a su derecha también deben ser opcionales. Esto quiere decir que los parámetros obligatorios no pueden seguir a los parámetros por defecto.

El siguiente ejemplo define una función de forma incorrecta:

```
def saludo(mensaje="encantado de saludarte", nombre)
```

Al tratar de hacer uso de la función, el intérprete de Python nos indicará el error:

```
def saludo(mensaje="encantado de saludarte", nombre):
                ^
SyntaxError: non-default argument follows default argument
```

## Parámetros posicionales y parámetros con nombre en una función

Tal y como te indiqué anteriormente, cuando invocamos a una función en Python con diferentes argumentos, los valores se asignan a los parámetros en el mismo orden en que se indican.

Siguiendo con el ejemplo anterior de la función saludo, imaginemos que llamamos a la función de la siguiente manera  saludo("j2logo", "¿cómo estás?"). De este modo, al parámetro nombre se le asignará el valor «j2logo» y al parámetro «mensaje» el valor «¿cómo estás?»

Esto es así porque el valor en el que se asignan los argumentos depende del orden con el que se llaman. Sin embargo, el orden se puede cambiar si llamamos a la función indicando el nombre de los parámetros. Para que lo veas más claro, los siguientes ejemplos son todos válidos:

```
>>>saludo(mensaje="¿cómo estás?", nombre="j2logo")
Hola j2logo, ¿cómo estás?

>>>saludo(nombre="j2logo", mensaje="¿cómo estás?")
Hola j2logo, ¿cómo estás?

>>>saludo("j2logo", mensaje="¿cómo estás?")
Hola j2logo, ¿cómo estás?
```

Como vemos, podemos mezclar el orden de los parámetros si indicamos su nombre. Eso sí, siempre deben estar a la derecha de los parámetros posicionales, es decir, aquellos que se indican sin nombre y cuyo valor se asigna en el orden en el que se indica.

## Conclusión

Por defecto, al llamar a una función los valores de los argumentos se asignan en el mismo orden en el que se pasan al invocar a dicha función.

Los parámetros opcionales se indican con el operador ‘=’, tienen un valor por defecto y siempre se definen después de los parámetros obligatorios.

Se puede modificar el orden de los argumentos con el que se invoca a una función si se indica el nombre de los parámetros. Los parámetros con nombre siempre aparecen después de los posicionales.

**Fuente:**

https://j2logo.com/python/tutorial/funciones-en-python/

# Número de parámetros indefinido (*args y **kwargs)

Hasta aquí hemos visto lo fundamental sobre los distintos tipos de parámetros al definir una función Python. No obstante, debes entender qué significan los parámetros ***args** y ****kwargs** en una función Python.

**NOTA:** todos los ejemplos se han implementado usando Python 3

En los lenguajes de programación de alto nivel, Python entre ellos, al declarar una función podemos definir una serie de parámetros con los que invocar a dicha función. Por regla general, el número y el nombre de estos parámetros son inmutables.

No obstante, hay situaciones en las que es mucho más apropiado que el número de parámetros sea opcional y/o variable.

## Qué significan *args y **kwargs como parámetros

## Entendiendo *args

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

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

El parámetro recibe los argumentos como una tupla.

Es un parámetro opcional. Se puede invocar a la función haciendo uso del mismo, o no.

El número de argumentos al invocar a la función es variable.

Son parámetros posicionales, por lo que, a diferencia de los parámetros con nombre, su valor depende de la posición en la que se pasen a la función.

La siguiente función toma dos parámetros y devuelve como resultado la suma de los mismos:

```
def sum(x, y):
    return x + y
```

Si llamamos a la función con los valores x=2 e y=3, el resultado devuelto será 5.

```
>>>sum(2, 3)
```

```
5
```

Pero, ¿qué ocurre si posteriormente decidimos o nos damos cuenta de que necesitamos sumar un valor más? ¿Cómo podemos solucionar este problema? Pues una opción sería añadir más parámetros a la función, pero ¿cuántos?

La mejor solución, la más elegante y la más al estilo Python es hacer uso de *args en la definición de esta función. De este modo, podemos pasar tantos argumentos como queramos. Pero antes de esto, tenemos que reimplementar nuestra función sum:

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

Con esta nueva implementación, podemos llamar a la función con cualquier número variable de valores:

```
>>>sum()
0

>>>sum(2, 3)
5

>>>sum(2, 3, 4)
9

>>>sum(2, 3, 4, 6, 9, 21)
45
```

## Entendiendo **kwargs

Veamos ahora el uso de **kwargs como parámetro.

En Python, el parámetro especial **kwargs en una función se usa para pasar, de forma opcional, un número variable de argumentos con nombre.

Las principales diferencias con respecto *args son:

Lo que realmente indica que el parámetro es de este tipo es el símbolo ‘**’, el nombre kwargs se usa por convención.

El parámetro recibe los argumentos como un diccionario.

Al tratarse de un diccionario, el orden de los parámetros no importa. Los parámetros se asocian en función de las claves del diccionario.

## ¿Cuándo es útil su uso?

Imaginemos que queremos implementar una función filter que nos devuelva una consulta SQL de una tabla clientes que tiene los siguientes campos: nombre, apellidos, fecha_alta, ciudad, provincia, tipo y fecha_nacimiento.

Una primera aproximación podría ser la siguiente:

```
def filter(ciudad, provincia, fecha_alta):
    return "SELECT * FROM clientes WHERE ciudad='{}' AND provincia='{}' AND
fecha_alta={};".format(ciudad, provincia, fecha_alta)
```

No es una función para sentirse muy contento. Entre los diferentes problemas que pueden surgir tenemos:

Si queremos filtrar por un nuevo parámetro, hay que cambiar la definición de la función así como la implementación.

Los parámetros son todos obligatorios.

Si queremos consultar otro tipo de clientes manteniendo esta consulta, debemos crear una nueva función.

La solución a todos estos problemas está en hacer uso del parámetro ****kwargs**. Veamos cómo sería la nueva función filter usando ****kwargs**:

```
def filter(**kwargs):
    query = "SELECT * FROM clientes"
    i = 0
    for key, value in kwargs.items():
        if i == 0:
            query += " WHERE "
        else:
            query += " AND "
        query += "{}='{}'".format(key, value)
        i += 1
    query += ";"
    return query
```

Con esta nueva implementación hemos resuelto todos nuestros problemas. A continuación podemos ver cómo se comporta la nueva función filter:

```
>>>filter()
SELECT * FROM clientes;

>>>filter(ciudad="Madrid")
SELECT * FROM clientes WHERE ciudad='Madrid';

>>>filter(ciudad="Madrid", fecha_alta="25-10-2018")
SELECT * FROM clientes WHERE ciudad='Madrid' AND fecha_alta='25-10-2018';
```

Para más información sobre *args y **kwargs dirigirse a: https://j2logo.com/args-y-kwargs-en-python/

## Conclusión

Utiliza ***args** para pasar de forma opcional a una función un número variable de argumentos posicionales.

El parámetro ***args** recibe los argumentos como una tupla.

Emplea ****kwargs** para pasar de forma opcional a una función un número variable de argumentos con nombre.

El parámetro ****kwargs** recibe los argumentos como un diccionario.

Por último, utilizar *args y **kwargs puede ahorrarte muchos problemas pero también puede llevarte a resultados inesperados si se utiliza correctamente.

**Fuente:**

https://j2logo.com/python/tutorial/funciones-en-python/

# Ámbito y ciclo de vida de las variables

En cualquier lenguaje de programación de alto nivel, toda variable está definida dentro de un ámbito. Esto es, los sitios en los que la variable tiene sentido y dónde se puede utilizar.

Los parámetros y variables definidos dentro de una función tienen un ámbito local, local a la propia función. Por tanto, estos parámetros y variables no pueden ser utilizados fuera de la función porque no serían reconocidos.

El ciclo de vida de una variable determina el tiempo en que una variable permanece en memoria. Una variable dentro de una función existe en memoria durante el tiempo en que está ejecutándose dicha función. Una vez que termina su ejecución, sus variables y parámetros desaparecen de memoria y, por tanto, no pueden ser referenciados.

```
>>> def saludo(nombre):
...     x = 10
...     print(f'Hola {nombre}')
...
>>> saludo('j2logo')
Hola j2logo
>>> print(x)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'x' is not defined
```

Como ves, en el ejemplo anterior, al tratar de mostrar por pantalla el valor de la variable x, el intérprete mostrará un error.

El siguiente ejemplo es diferente:

```
>>> def muestra_x():
...     x = 10
...     print(f'x vale {x}')
...
>>> x = 20
>>> muestra_x()
x vale 10
>>> print(x)
20
```

Lo que sucede en este ejemplo es que dentro de la función muestra_x() se está creando una nueva variable x que, precisamente, tiene el mismo nombre que la variable definida fuera de la función. Por tanto, x dentro de la función tiene el valor 10, pero una vez que la función termina, x hace referencia a la variable definida fuera, cuyo valor es 20.

Las variables definidas fuera de una función tienen un ámbito conocido como global y son visibles dentro de las funciones, dónde solo se puede consultar su valor.

```
>>> y = 20
>>> def muestra_x():
```

```
...     x = 10
...     print(f'x vale {x}')
...     print(f'y vale {y}')
...
>>> muestra_x()
x vale 10
y vale 20
```

Para modificar dentro de una función una variable definida fuera de la misma, hay que usar la palabra reservada global para declarar la variable dentro de la función. Aunque esto no siempre es recomendable ya que conlleva sus problemas.

**Fuente:**

https://j2logo.com/python/tutorial/funciones-en-python/

# Funciones lambda

A continuación veremos qué son las funciones lambda en Python, también conocidas como funciones anónimas. Aquí descubrirás qué son estas funciones, cómo se definen y sus principales usos.

## Qué son las funciones lambda en Python

En Python, las funciones lambda son también conocidas como funciones anónimas porque se definen sin un nombre.

Como sabes, una función en Python se define con la palabra reservada def. Sin embargo, una función anónima se define con la palabra reservada lambda.

## Cómo se define una función anónima en Python

La sintaxis para definir una función lambda es la siguiente:

```
lambda parámetros: expresión
```

A continuación, te detallo las características principales de una función anónima:

Son funciones que pueden definir cualquier número de parámetros pero una única expresión. Esta expresión es evaluada y devuelta.

Se pueden usar en cualquier lugar en el que una función sea requerida.

Estas funciones están restringidas al uso de una sola expresión.

Se suelen usar en combinación con otras funciones, generalmente como argumentos de otra función.

Ejemplo:

```
cuadrado = lambda x: x ** 2
```

En el ejemplo anterior, x es el parámetro y x ** 2 la expresión que se evalúa y se devuelve.

Como ves, la función no tiene nombre y toda la definición devuelve una función que se asigna al identificador cuadrado.

En el siguiente ejemplo se aprecian las similitudes y diferencias de usar una función anónima y una función normal:

```
>>> def cuadrado(x):
...     return x ** 2
```

```
>>> cuad = lambda x: x ** 2

>>> print(cuadrado(3))
9

>>> print(cuad(5))
25
```

## Ejemplo de uso de una función lambda: map()

En esta última parte vamos a descubrir el potencial de las funciones lambda, especialmente cuando se usan en combinación con otras funciones.

## map()

La función map() en Python aplica una función a cada uno de los elementos de una lista.

```
map(una_funcion, una_lista)
```

Imagina que tienes una lista de enteros y quieres obtener una nueva lista con el cuadrado de cada uno de ellos.

Seguramente, lo primero que se te ha ocurrido es algo similar a lo siguiente:

```
enteros = [1, 2, 4, 7]
cuadrados = []
for e in enteros:
    cuadrados.append(e ** 2)

print(cuadrados)
[1, 4, 16, 49]
```

Sin embargo, podemos usar una función anónima en combinación con map() para obtener el mismo resultado de una manera mucho más simple:

```
enteros = [1, 2, 4, 7]
cuadrados = list(map(lambda x : x ** 2, enteros))

print(cuadrados)
[1, 4, 16, 49]
```

La cosa se vuelve todavía más interesante cuando, en lugar de una lista de valores, pasamos como segundo parámetro una lista de funciones:

```
enteros = [1, 2, 4, 7]

def cuadrado(x):
    return x ** 2

def cubo(x):
    return x ** 3
```

```
funciones = [cuadrado, cubo]
for e in enteros:
    valores = list(map(lambda x : x(e), funciones))
    print(valores)

[1, 1]
[4, 8]
[16, 64]
[49, 343]
```

Para más información sobre funciones lambda dirigirse a:
https://j2logo.com/python/funciones-lambda-en-python/

**Fuente:**

https://j2logo.com/python/tutorial/funciones-en-python/

# Documentar una función (Docstrings)

Como vimos anteriormente, los docstrings son un tipo de comentarios especiales que se usan para documentar un módulo, función, clase o método. En realidad son la primera sentencia de cada uno de ellosy se encierran entre tres comillas simples o dobles.

Los docstrings son utilizados para generar la documentación de un programa. Además, suelen utilizarlos los entornos de desarrollo para mostrar la documentación al programadorde forma fácil e intuitiva.

Veámoslo con un ejemplo:

def suma(a, b):

"""Esta función devuelve la suma de los parámetros a y b"return a + b

## Especificación

Un **docstring** es una cadena que aparece como la primera declaración en una definición de módulo, función, clase o método. Tal **docstring** se convierte en el atributo especial __doc__ de ese objeto.

Todos los módulos deberían tener normalmente **docstrings**, y todas las funciones y clases exportadas por un módulo también deberían tener **docstrings**. Los métodos públicos (incluido el constructor __init__) también deben tener cadenas de documentación. Un paquete puede estar documentado en el módulo **docstring** del archivo __init__.py en el directorio del paquete.

Consulte los PEP 257 y 258 para obtener más información sobre este tema.

Para mantener la coherencia, utilice siempre "" "comillas dobles triples" "" alrededor de las cadenas de documentos. Utilice r "" "comillas dobles triples sin formato" "" si utiliza barras invertidas en sus cadenas de documentos. Para cadenas de documentos Unicode, utilice u "" "cadenas Unicode entre comillas triples" "".

**Fuente:**

https://j2logo.com

https://www.python.org/dev/peps/pep-0257/

https://www.python.org/dev/peps/pep-0258/

# Módulos y paquetes

Los módulos y paquetes en Python son la forma de organizar los scripts y programas a medida que estos crecen en número de líneas de código.

## Qué es un módulo en Python

En Python, un módulo no es más que un fichero que contiene instrucciones y definiciones (variables, funciones, clases, etc.). El fichero debe tener extensión .py y su nombre se corresponde con el nombre del módulo.

**NOTA:** Dentro de un módulo, puedes acceder al nombre del mismo a través de la variable global __name__.

Los módulos tienen un doble propósito:

Dividir un programa con muchas líneas de código en partes más pequeñas.

Extraer un conjunto de definiciones que utilizas frecuentemente en tus programas para ser reutilizadas. Esto evita, por ejemplo, tener que estar copiando funciones de un programa a otro.

**Es una buena práctica que un módulo solo contenga instrucciones y definiciones relacionadas entre sí.**

## ¿Cómo crear tu primer módulo?

Sitúate en un directorio para un nuevo proyecto y crea en él un fichero llamado mis_funciones.py con el siguiente contenido:

```
def saludo(nombre):
    print(f'Hola {nombre}')
```

Ahora, desde una consola, sitúate en el directorio anterior y ejecuta el comando python3 para lanzar el intérprete de Python.

Una vez dentro del intérprete, ejecuta el siguiente comando:

```
>>> import mis_funciones
```

Con el comando anterior estamos importando el módulo mis_funciones en el intérprete. Prueba a llamar ahora a la función saludo() de este modo:

```
>>> mis_funciones.saludo('j2logo')
Hola j2logo
```

## Cómo importar módulos en Python

Como has visto al final del apartado anterior, para usar las definiciones de un módulo en el intérprete o en otro módulo, primero hay que importarlo. Para ello, se usa la palabra reservada **import**. Una vez que un módulo ha sido importado, se puede acceder a sus definiciones a través del operador punto **.**.

**Aunque puedes importar los módulos y sus definiciones dónde y cuando quieras, es una buena práctica que aparezcan al principio del módulo.**

Ya sabes que un módulo puede contener instrucciones y definiciones. Normalmente, las instrucciones son utilizadas para inicializar el módulo y solo son ejecutadas la primera vez que aparece el nombre del módulo en una sentencia import.

## from … import …

Podemos importar uno o varios nombres de un módulo del siguiente modo:

```
from mis_funciones import saludo, otra_funcion

saludo('j2logo')
```

Esto nos permite acceder directamente a los nombres definidos en el módulo sin tener que utilizar el operador **.**.

## from … import *

```
from mis_funciones import *

saludo('j2logo')
```

Es similar al caso anterior, solo que importa todas las definiciones del módulo a excepción de los nombres que comienzan por guión bajo _.

**IMPORTANTE:** no es una buena práctica importar así las definiciones de un módulo porque dificulta la lectura y los nombres importados pueden ocultar identificadores y nombres usados en el módulo en el que se importan.

## from … import as

Por último, podemos redefinir el nombre con el que una definición será usada dentro de un módulo utilizando la palabra reservada **as**:

```
>>> from mis_funciones import saludo as hola

>>> hola('j2logo')
Hola j2logo
```

Ejecutar módulos como script

Un módulo puede ser ejecutado como un script o como punto de entrada de un programa cuando se pasa directamente como parámetro al intérprete de Python:

```
>>> python mis_funciones.py
```

Cuando esto ocurre, el código del módulo se ejecuta como si se hubiera importado, con la particularidad de que el nombre del módulo, __name__, es __main__.

Esto hace realmente interesante añadir al final del módulo las siguientes líneas de código, que solo se ejecutarán en caso de que dicho módulo se haya ejecutado como el principal:

```
if __name__ == '__main__':
    hola('j2logo')
```

No se ejecutarán en caso de que el módulo se importe en otro módulo.

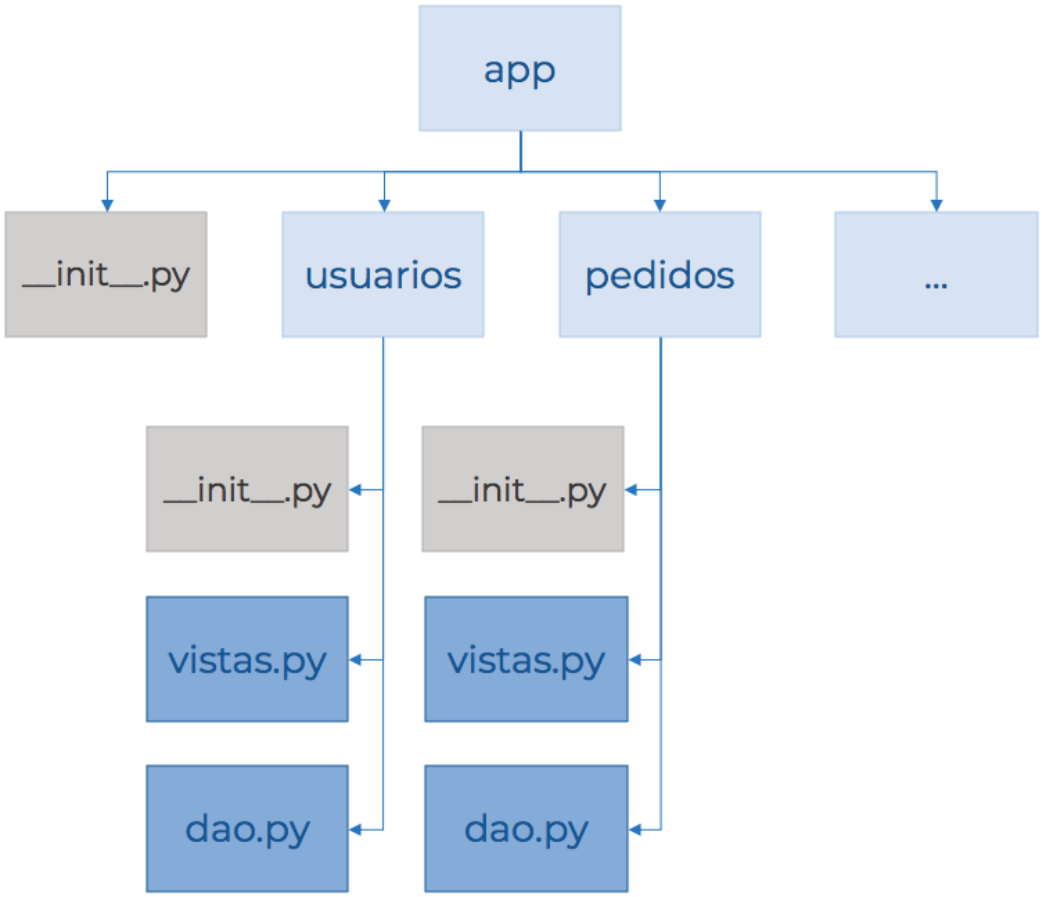
## Paquetes en Python

Del mismo modo en que agrupamos las funciones y demás definiciones en módulos, los paquetes en Python permiten organizar y estructurar de forma jerárquica los diferentes módulos que componen un programa. Además, los paquetes hacen posible que existan varios módulos con el mismo nombre y que no se produzcan errores.

**Un paquete es simplemente un directorio que contiene otros paquetes y módulos.** Además, en Python, para que un directorio sea considerado un paquete, este debe incluir un módulo llamado __init__.py. En la mayoría de ocasiones, el fichero __init__.py estará vacío, sin embargo, se puede utilizar para inicializar código relacionado con el paquete.

Al igual que sucede con los módulos, cuando se importa un paquete, Python busca a través de los directorios definidos en sys.path el directorio perteneciente a dicho paquete.

Para que lo veas todo de forma gráfica, te mostramos los conceptos con una imagen. Imagina que estás haciendo una aplicación para gestionar pedidos. Una forma de organizar los diferentes módulos podría ser la siguiente:




Más adelante seguiremos desarrollando estos conceptos.

Para más información sobre módulos y paquetes puede dirigirse a:
https://j2logo.com/python/tutorial/espacios-de-nombres-modulos-y-paquetes/

**Fuente:**

https://j2logo.com/python/tutorial/