Funciones

En Python, las funciones se pueden definir en cualquier punto de un programa. La primera línea de la definición de una función contiene:

- la palabra reservada def
- el nombre de la función (la guía de estilo de Python recomienda escribir todos los caracteres en minúsculas separando las palabras por guiones bajos)
- paréntesis (que pueden incluir los argumentos de la función) y dos puntos (:)

Las instrucciones que forman la función se escriben con sangría con respecto a la primera línea. Se puede indicar el final de la función con la palabra reservada return, aunque no es obligatorio. Para poder utilizar una función en un programa se tiene que haber definido antes. El ejemplo siguiente muestra un programa que contiene una función y el resultado de la ejecución de ese programa.

Definición de una función	Resultado de la ejecución
def mensajes(): print("Este es el primer ejemplo") mensajes() print("Programa terminado.")	Este es el primer ejemplo Programa terminado.

El ejemplo siguiente muestra un programa incorrecto que intenta utilizar una función antes de haberla definido.

mensajes()
print("Programa terminado.")
def mensajes():
  print("Este es el primer ejemplo")
	Traceback (most recent call last):
	  File " funcion01A.py", line 3, in <module>
	  mensajes()
	NameError: name 'mensajes' is not defined

Ámbitos de las variables en Python

- Python distingue tres tipos de ámbitos: las variables locales y dos tipos de variables “libres” (globales y no locales):
  - variables locales: las que pertenecen al ámbito de la función (y que pueden ser accesibles por niveles inferiores)
  - variables globales: las que pertenecen al ámbito del programa principal.
  - variables no locales: las que pertenecen a un ámbito superior al de la función, pero que no son globales.

Si el programa contiene solamente funciones que no contienen a su vez funciones, todas las variables libres son variables globales. Pero si el programa contiene una función que a su vez contiene una función, las variables libres de esas "sub-funciones" pueden ser globales (si pertenecen al programa principal) o no locales (si pertenecen a la función).

- Para identificar explícitamente las variables globales y no locales, se utilizan las palabras reservadas **global** y **nonlocal**.
- Las variables **locales** no necesitan identificación.

Variables locales

Si no se han declarado como globales o no locales, las variables **a las que se asigna valor** en una función se consideran variables **locales**, es decir, sólo existen en la propia función, incluso cuando en el programa exista una variable con el mismo nombre, como muestra el siguiente ejemplo:

def funcion():
  a = 2 #variable local en la función
  print(a)
a = 5
print(a)
funcion()
print(a)
	5
	2
	5

Las variables **locales** sólo existen en la propia función y no son accesibles desde niveles superiores, como puede verse en el siguiente ejemplo:

def funcion():
  a = 2
  print(a)
funcion()
print(a)
	Traceback (most recent call last):
	  File " funcion.py", line 6, in <module>
	  print(a)
	NameError: name 'a' is not defined

Si en el interior de una función **se asigna valor** a una variable que no se ha declarado como global o no local, esa variable es **local** a todos los efectos. Por ello el siguiente programa da error:

def funcion():
  print(a)
  a = 2
  print(a)
a = 5
funcion()
print(a)
	Traceback (most recent call last):
	  File " funcion.py", line 8, in <module>
	  funcion()
	  File " funcion.py", line 2, in funcion
	  print(a)
	UnboundLocalError: local variable 'a' referenced before assignment

El motivo es que en la función se asigna valor a la variable "a" (en la segunda instrucción) por lo tanto Python la considera variable local. Como la primera instrucción quiere imprimir el valor de "a", pero a esa variable todavía no se le ha dado valor en la función (el valor de la variable "a" del programa principal no cuenta *pues se trata de variables distintas*, aunque se llamen igual), se produce el mensaje de error.

Variables libres globales o no locales

Si a una variable **no se le asigna valor** en una función, Python la considera **libre** y busca su valor en los niveles superiores de esa función, empezando por el inmediatamente superior y continuando hasta el programa principal. Si a la variable se le asigna valor en algún nivel intermedio la variable se considera **nonlocal;** y si se le asigna en el programa principal la variable se considera **global**. En el ejemplo siguiente, la variable libre "a" de la función funcion() se considera **global** porque obtiene su valor del programa principal:

def funcion(): print(a) a = 5 funcion() print(a)	5 5

La última instrucción del programa escribe de nuevo el valor de "a", que sigue siendo 5 pues la función no ha modificado el valor de la variable. Al ser global es "visible" por cualquier función que puede utilizar su valor. Entonces:

x = 0 # Es global por estar fuera de las funciones def funcion(): print(x) funcion()	0
Pero hay un potencial conflicto. Si una función intenta cambiar el valor de x:	

x = 0
def funcion():
  x = 1
  print(x)
funcion()
	1

Debido a que aparece una asignación dentro de la función la variable x se considerará local. Se crea por tanto una nueva variable x específica para la función que impedirá acceder a la variable global del mismo nombre. Aunque print(x) emitirá 1, es un valor local. Cuando la función termine, dejará de existir. La x global seguirá en 0. Debido a que esta x es local, se nos puede plantear exactamente el mismo problema ya visto si la modificamos dentro	

de la función, pero la intentamos usar *antes* de haberla modificado, así:

```
x = 0
def funcion():
   print(x)
   x = 1
funcion()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in funcion
UnboundLocalError: local variable 'x' referenced before assignment
```

Antes de intentar ejecutar esta función Python ya ha visto que en alguna línea se asigna un valor a x, y considerará x local. Por lo tanto el primer print(x) intentará imprimir la variable local y como aún no tiene un valor asignado, se producirá el error. Para resolver este problema existe la palabra global que permite especificarle a Python que, aunque vea una asignación dentro de la función, no cree una nueva variable local, sino que haga uso de la global:

```
x = 0
def test():
   global x
   print(x)
   x = 1
funcion()
```

```
0
```

En este caso, las referencias que ocurren a x dentro de la función acceden a la x global, por lo que el código no sólo no da error, sino que cuando la función termine de ejecutarse el valor de la x global habrá cambiado.

nonlocal existe para un caso muy particular y es el de funciones anidadas que necesitan acceder a variables que, ni son locales de la propia función anidada, ni son globales, sino que son variables de la función dentro de la cual están anidadas.

```
def funcion():
   x = 1
   def anidada():
      x = 2
      print(x)
   anidada()
   print(x)
print(x)
funcion()
print(x)
```

```
0
2
1
0
```

Al ejecutar la función funcion, se crea una nueva variable local para la función funcion(). A esa x se le da el valor 1, la x global sigue en 0. Luego se define una función anidada, y se invoca. Dentro de ella hay una asignación x=2, pero como no se ha mencioado que la x sea global, nuevamente se creará una local y se le asigna a x el valor 2. La global sigue siendo 0. Así que se emitirá un 2 y la global seguirá en 0

```
x = 0
def funcion():
   x = 1
   def anidada():
      global x
      x = 2
      print(x)
   anidada()
   print(x)
print(x)
funcion()
print(x)
```

```
0
2
1
2
```

En este ejemplo se crea x=2 actúa sobre la global. Y si quisiéramos desde anidada() modificar la x local de funcion()?. Para esto es la palabra nonlocal:

```
x = 0
def funcion():
   x = 1
   def anidada():
      nonlocal x
```

```
0
2
2
0
```

```
        x = 2
        print(x)
    anidada()
    print(x)
print(x)
funcion()
print(x)
```

Cuando Python ve **nonlocal** x, sabe que aunque esa función intente asignar valores a una x, no debe crear una variable local para ello. Deberá usar la x que esté accesible dentro de su alcance (que será la definida en la asignación x=1)

Otro ejemplo, la variable libre "a" de la función sub_funcion() se considera no local porque obtiene su valor de una función intermedia:

```
def funcion():
    def sub_funcion():
        print(a)

    a = 3
    sub_funcion()
    print(a)

a = 4
funcion()
print(a)
```

```
3
3
4
```

En la función sub_funcion(), la variable "a" es libre pues no se le asigna valor. Se busca su valor en los niveles superiores, por orden. En este caso, el nivel inmediatamente superior es la función funcion(). Como en ella hay una variable local que también se llama "a", Python toma de ella el valor (en este caso, 3) y lo emite. Para la función sub_funcion(), la variable "a" es una variable **nonlocal**, porque su valor proviene de una función intermedia. Si a una variable que Python considera libre (porque no se le asigna valor en la función) tampoco se le asigna valor en niveles superiores, Python dará un mensaje de error.

Entonces…

Si queremos asignar valor a una variable en una subrutina, pero no queremos que Python la considere local, debemos declararla en la función como **global** o **nonlocal**. En el ejemplo siguiente la variable se declara como **global**, para que su valor sea el del programa principal:

```
def subrutina():
    global a
    print(a)
    a = 1
a = 5
subrutina()
print(a)
```

```
5
1
```

En el ejemplo siguiente la variable se declara como **nonlocal**, para que su valor sea el de la función intermedia:

```
def funcion():
    def sub_funcion():
        nonlocal a
        print(a)
        a = 1
    a = 3
    sub_funcion()
    print(a)
```

```
3
1
4
```

a = 4 funcion() print(a)	

Si a una variable declarada **global** o **nonlocal** en una función no se le asigna valor en el nivel superior correspondiente, Python dará un error de sintaxis, como muestra el programa siguiente:

Ejemplo de variable declarada nonlocal no definida

def funcion():
  def sub_funcion():
  nonlocal a
  print(a)
  a = 1
  sub_funcion()
  print(a)
a = 4
funcion()
print(a)
	File "funcion.py", line 3
	  nonlocal a
	  ^
	SyntaxError: no binding for nonlocal 'a' found

Funciones nativas locals() y globals():

locals(): Devuelve un diccionario que contiene las variables locales del ámbito actual.	

def demo1():
  print("Aquí no hay variable presente: ", **locals()**)
def demo2():
  name = "Python"
  print("Aquí hay variable presente: ", **locals()**)
demo1()
demo2()
	>>> demo1()
	Aquí no hay variable presente: {}
	>>> demo2()
	Aquí hay variable presente: {'name': 'Python'}
	>>>

globals(): Devuelve el diccionario que contiene las variables globales del ámbito actual.	

def demo1():
  print("Aquí no hay variable presente: \n ", **locals()**)
def demo2():
  name = "Python"
  print("Aquí hay variable presente: \n", **locals()**)
demo1()
demo2()
print("Esto está usando globals() : \n ", **globals()**)
	Aquí no hay variable presente: {}
	Aquí hay variable presente: {'name': 'Python'}
	Esto está usando globals() :
	{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x000002150A62CA48>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': 'c:/Users/Python Scripts/locals_globals.py', '__cached__': None, 'demo1': <function demo1 at 0x000002150C5A5558>, 'demo2': <function demo2 at 0x000002150C590CA8>}

callable()

La función **callable**() indica si un objecto puede ser llamado.

def demo1():
  name = "Python"
demo1()
callable(demo1)
	>>> demo1()
	>>> callable(demo1)
	True

Argumentos y retorno de valores

Las funciones en Python admiten argumentos en su llamada y permiten retornar valores. Esta posibilidad permite crear funciones más útiles y reutilizables.

Ejemplo de función sin parámetros y con emisión de un mensaje:

```
def media():
    media = (a + b) / 2
    print(f"La media de {a} y {b} es: {media}")
a = 3
b = 5
media()
print("Programa terminado")
```

```
La media de 3 y 5 es: 4.0
Programa terminado
```

Este tipo de función es muy difícil de reutilizar. Para evitar ese problema, las funciones admiten argumentos, es decir, permiten que se les envíen valores con los que trabajar. De esa manera, las funciones se pueden reutilizar más fácilmente, como muestra el ejemplo siguiente:

```
def media(x, y):
    media = (x + y) / 2
    print(f"La media de {x} y {y} es: {media}")
a = 3
b = 5
media(a, b)
print("Programa terminado")
```

```
La media de 3 y 5 es: 4.0
Programa terminado
```

Esta función puede ser utilizada con más flexibilidad que la del ejemplo anterior. Pero aún tiene un inconveniente, como las variables locales de una función son inaccesibles desde los niveles superiores, el programa principal no puede utilizar la variable "media" calculada por la función y tiene que ser la función la que calcule y emita el resultado. Para evitar ese problema, las funciones pueden retornar valores, es decir, *pueden enviar valores al programa o función de nivel superior*:

```
def calcula_media(x, y):
    resultado = (x + y) / 2
    return resultado
a = 3
b = 5
media = calcula_media(a, b)
print(f"La media de {a} y {b} es: {media}")
print("Programa terminado")
```

```
La media de 3 y 5 es: 4.0
Programa terminado
```

Esta función puede ser utilizada con más flexibilidad que la del ejemplo anterior. Pero esta función tiene todavía un inconveniente y es que sólo calcula la media de dos valores. Sería más interesante que la función calculara la media de cualquier cantidad de valores. Para evitar ese problema, las funciones pueden admitir una cantidad indeterminada de valores, como muestra el ejemplo siguiente:

```
def calcula_media(*args):
    total = 0
    for i in args:
        total += i
    resultado = total / len(args)
    return resultado

a, b, c = 3, 5, 10
media = calcula_media(a, b, c)
print(f"La media de {a}, {b} y {c} es: {media}")
print("Programa terminado")
```

```
La media de 3, 5 y 10 es: 6.0
Programa terminado
```

Esta última función puede ser utilizada aún con más flexibilidad que las anteriores, puesto que se puede elegir de cuántos valores hacer la media y qué hacer con el valor calculado por la función. Las funciones pueden devolver varios valores simultáneamente, como muestra el siguiente ejemplo:

```
def calcula_media_desviacion(*args):
    total = 0
    for i in args:
        total += i
```

```
Datos: 3 5 10 12
Media: 7.5
Desviación típica: 3.640054944640259
Programa terminado
```

```
    media = total / len(args)
    total = 0
    for i in args:
        total += (i - media) ** 2
    desviacion = (total / len(args)) ** 0.5
    return media, desviacion

a, b, c, d = 3, 5, 10, 12
media, desviacion_tipica = calcula_media_desviacion(a, b, c, d)
print(f"Datos: {a} {b} {c} {d}")
print(f"Media: {media}")
print(f"Desviación típica: {desviacion_tipica}")
print("Programa terminado")
```

*args y **kwargs

No es obligatorio usar los nombres args o kwargs, pueden reemplazarse con nombres descriptivos.

args representa un conjunto arbitrario de argumentos posicionales. Lo importante es que use el operador de desempaquetado (*).Tanto list como tuple admiten el corte y la iteración, pero se diferencian en que list es mutable y tuple inmutable

```
def saludo(tipoSaludo, *amigos):
    listaDeAmigos = ""
    print(type(amigos))
    for amigo in amigos:
        listaDeAmigos = listaDeAmigos + ', ' + amigo
    print(tipoSaludo + listaDeAmigos)
print("Ejemplo con *args\n")
saludo("Hola", "Diego", "Juan", "...a todos")
```

```
>>> print("Ejemplo con *args\n")
Ejemplo con *args
>>> saludo("Hola", "Diego", "Juan", "...a todos")
<class 'tuple'>
Hola, Diego, Juan, ...a todos
```

kwargs es lo mismo que args pero con argumentos *keyword*, o nombre. Lo importante es el uso del operador de desempaquetado (**). Hay que tener en cuenta que si se itera por un dict y se desea retornar el valor se debe usar .values()

```
def argsConClaveValor(**clave_valor_args):
    print(type(clave_valor_args))
    for nombre, valor in clave_valor_args.items():
        print(nombre + ': ' + valor)

print("Ejemplo con **kwargs\n")
argsConClaveValor(edad='32', profesion="Ingeniero", nacion
alidad="Argentino")
```

```
>>> print("Ejemplo con **kwargs\n")
Ejemplo con **kwargs
>>> argsConClaveValor(edad='32', profesion="Ingeniero",
nacionalidad="Argentino")
<class 'dict'>
edad: 32
profesion: Ingeniero
nacionalidad: Argentino
```

¿Qué sucede si desea crear una función que tome un número variable de argumentos posicionales y con nombre?. En este caso el orden cuenta. Del mismo modo que los argumentos no predeterminados tienen que preceder a los argumentos predeterminados, también *args debe aparecer antes **kwargs. El orden correcto es:

1. Argumentos estándar
2. *args argumentos
3. **kwargs argumentos

Por ejemplo, esta definición de función es correcta:

```
def funcion(a, b, *args, **kwargs):
    pass
```

Variables globales y objetos mutables e inmutables

En Python dependiendo de si a la función se le envía como parámetro un objeto mutable o inmutable, la función podrá modificar o no al objeto. En los dos siguientes ejemplos, el parámetro de la función ("b") se llama igual que una de las

dos variables del programa principal. En los dos ejemplos se llama dos veces a la función, enviando cada vez una de las dos variables ("a" y "b").

Ejemplo de conflicto entre nombre de parámetro y nombre de variable global. Objeto mutable.

Como en este caso las variables son listas (objetos mutables), la función modifica la lista que se envía como argumento: primero se modifica la lista "a" y a continuación la lista "b". La lista modificada no depende del nombre del parámetro en la función (que es "b"), sino de la variable enviada como argumento ("a" o "b").

```
def cambia(b):
    b += [5]
a, b = [3], [4]
print(f"Al principio       : a = {a} b = {b}")
cambia(a)
print(f"Después de cambia(a): a = {a} b = {b}")
cambia(b)
print(f"Después de cambia(b): a = {a} b = {b}")
print("Programa terminado")
```

```
Al principio       : a = [3] b = [4]
Después de cambia(a): a = [3, 5] b = [4]
Después de cambia(b): a = [3, 5] b = [4, 5]
Programa terminado
```

Ejemplo de conflicto entre nombre de parámetro y nombre de variable global. Objeto inmutable

Como en este caso las variables son números enteros (objetos inmutables), la función no puede modificar los números que se envían como argumentos, ni la variable "a" ni la variable "b".

```
def cambia(b):
    b += 1
a, b = 3, 4
print(f"Al principio       : a = {a} b = {b}")
cambia(a)
print(f"Después de cambia(a): a = {a} b = {b}")
cambia(b)
print(f"Después de cambia(b): a = {a} b = {b}")
print("Programa terminado")
```

```
Al principio       : a = 3 b = 4
Después de cambia(a): a = 3 b = 4
Después de cambia(b): a = 3 b = 4
Programa terminado
```

Paso por valor o paso por referencia

En los lenguajes en los que las variables son "cajas" en las que se guardan valores, cuando se envía una variable como argumento en una llamada a una función suelen existir dos posibilidades:

- paso por valor: se envía simplemente el valor de la variable, en cuyo caso la función no puede modificar la variable, pues la función sólo conoce su valor.
- paso por dirección: se envía la dirección de memoria de la variable, en cuyo caso la función sí que puede modificar la variable.

En Python no se hace ni una cosa ni otra. En Python cuando se envía una variable como argumento en una llamada a una función lo que se envía es la referencia al objeto al que hace referencia la variable. Dependiendo de si el objeto es mutable o inmutable, la función podrá modificar o no el objeto.

Ejemplo de paso de variable (objeto inmutable)

```
def aumenta(x):
    print(id(x))
    x += 1
    print(id(x))
    return x
a = 3
print(id(3), id(4))
print(id(a))
```

```
140707989057872 140707989057904
140707989057872
140707989057872
140707989057904
4
3
140707989057872
```

print(aumenta(a)) print(a) print(id(a))	

Ejemplo de paso de variable (objeto mutable)

def aumenta(x):
  print(id(x))
  x += [1]
  print(id(x))
  return x
a = [3]
print(id(a))
print(aumenta(a))
print(a)
print(id(a))
	2192164999752
	2192164999752
	2192164999752
	[3, 1]
	[3, 1]
	2192164999752

Funciones anónimas

Una función anónima, como su nombre indica es una función sin nombre. En Python podemos ejecutar una función sin definirla con **def**. De hecho son similares pero con una diferencia fundamental: ***El contenido de una función lambda debe ser una única expresión en lugar de un bloque de acciones.*** Por lo tanto se puede decir que, mientras las funciones anónimas **lambda** sirven para realizar funciones simples, las funciones definidas con **def** sirven para manejar tareas más extensas. Ejemplo:

Función que calcule el doble de un valor	Simplificamos un poco	Simplificamos más escribiéndola en una sola línea	La convertimos en una función anónima guardando en una variable el resultado y utilizarla tal como haríamos con una función normal:

def doble(num):
  resultado = num*2
  return resultado
doble(2)
	def doble (num):
	  return num*2
		def doble (num): return num*2	doble= **lambda** num: num*2
			doble(2)

O sea que:

Función anónima	Es equivalente a:
funcion = lambda argumentos: resultado	def funcion(argumentos): return resultado

Diferencias entre las funciones lambda y las definidas con def

Aunque aparentemente se ha obtenido el mismo resultado existen ciertas diferencias entre ambos métodos que es necesario tener en cuenta.

- Al utilizar la palabra clave lambda se crea un objeto función sin crearse al mismo tiempo un nombre en el espacio de nombres. Nombre que sí se crea al definir la función con def.
- Las funciones lambda se crea en una única línea de código, por lo que son adecuadas cuando se desea minimizar el número de estas.
- Las funciones lambda son generalmente menos legibles que las tradicionales.
- En el caso de que se desee una función lambda es necesario asignarla a una variable, porque, si no es así, al carecer de identificador, solamente se podrá utilizar en la línea donde se defina.

Módulos

En Python, cada uno de nuestros archivos .py se denominan **módulos**. Estos módulos, a la vez, pueden formar parte de **paquetes**. Un paquete, es una carpeta que contiene archivos .py.

Para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado __init__.py. Este archivo, no necesita contener ninguna instrucción. De hecho, puede estar completamente vacío.	Los paquetes, a la vez, también pueden contener otros sub-paquetes:	Y los módulos, no necesariamente, deben pertenecer a un paquete:
└── paquete ├── __init__.py ├── modulo1.py ├── modulo2.py └── modulo3.py	└── paquete ├── __init__.py ├── modulo1.py └── subpaquete ├── __init__.py ├── modulo1.py └── modulo2.py	├── modulo1.py └── paquete ├── __init__.py ├── modulo1.py └── subpaquete ├── __init__.py ├── modulo1.py └── modulo2.py

Importando módulos enteros

El contenido de cada módulo, podrá ser utilizado a la vez, por otros módulos. Para ello, es necesario **importar los módulos** que se quieran utilizar. Para importar un módulo, se utiliza la instrucción import, seguida del nombre del paquete (si aplica) más el nombre del módulo (sin el .py) que se desee importar.

importar un módulo que no pertenece a un paquete	importar un módulo que está dentro de un paquete	
import modulo	import paquete.modulo1	import paquete.subpaquete.modulo1

La instrucción `import seguida de nombre_del_paquete.nombre_del_modulo, nos permitirá hacer uso de todo el código que dicho módulo contenga. Python tiene sus propios módulos, los cuales forman parte de su librería de módulos estándar, que también pueden ser importados.

Namespaces

Para **acceder** (desde el módulo donde se realizó la importación), a cualquier elemento del módulo importado, se realiza mediante el *namespace*, seguido de un punto (.) y el nombre del elemento que se desee obtener. En Python, un namespace, es el nombre que se ha indicado luego de la palabra import, es decir la ruta (namespace) del módulo:

print modulo.CONSTANTE_1	print paquete.modulo1.CONSTANTE_1	print paquete.subpaquete.modulo1.CONSTANTE_1

Alias

Es posible también, abreviar los namespaces mediante un *alias*. Para ello, durante la importación, se asigna la palabra clave as seguida del alias con el cuál nos referiremos en el futuro a ese namespace importado:

import modulo as m	import paquete.modulo1 as pm	import paquete.subpaquete.modulo1 as psm

Luego, para acceder a cualquier elemento de los módulos importados, el namespace utilizado será el alias indicado durante la importación:

print m.CONSTANTE _1	print pm.CONSTANTE _1	print psm.CONSTANTE_1

Importar módulos sin utilizar namespaces

En Python, es posible también, importar de un módulo solo los elementos que se desee utilizar. Para ello se utiliza la instrucción from seguida del namespace, más la instrucción import seguida del elemento que se desee importar:

En este caso, se accederá directamente al elemento, sin recurrir a su namespace:	Es posible también, importar más de un elemento en la misma instrucción. Para ello, cada elemento irá separado por una coma (,) y un espacio en blanco:	Pero ¿qué sucede si los elementos importados desde módulos diferentes tienen los mismos nombres? En estos casos, habrá que prevenir fallos, utilizando alias para los elementos:
from paquete.modulo1 import CONSTANTE_1 print CONSTANTE_1	from paquete.modulo1 import CONSTANTE_1, CONSTANTE_2	from paquete.modulo1 import CONSTANTE_1 as C1, CONSTANTE_2 as C2 from paquete.subpaquete.modulo1 import CONSTANTE_1 as CS1, CONSTANTE_2 as CS2 print C1 print C2 print CS1 print CS2

PEP8 importación: La importación de módulos debe realizarse al comienzo del documento, en orden alfabético de paquetes y módulos. Primero deben importarse los módulos propios de Python. Luego, los módulos de terceros y finalmente, los módulos propios de la aplicación. Entre cada bloque de imports, debe dejarse una línea en blanco.
De forma alternativa (pero muy poco recomendada), también es posible importar todos los elementos de un módulo, sin utilizar su namespace pero tampoco alias. Es decir, que a todos los elementos importados se accederá con su nombre original:

```
from paquete.modulo1 import *
print CONSTANTE_1
print CONSTANTE_2
```

Definir nuestros propios módulos e importarlos:

mi_primer_modulo.py	uso_mi_primer_modulo.py	Resultado

def suma(a1,a2):
  print("La suma es: ",a1+a2)
def resta(a1,a2):
  print("La resta es: ",a1-a2)
def multiplicacion(a1,a2):
  print("La multiplicacion es: ",a1*a2)
def division(a1,a2):
  print("La division es: ",a1/a2)
	import **mi_primer_modulo**
	
	suma(7, 12)
	resta(19, 7)
	multiplicacion(8, 9)
	division(120, 4)
		>>> suma(7, 12)
		La suma es: 19
		>>> resta(19, 7)
		La resta es: 12
		>>> multiplicacion(8, 9)
		La multiplicacion es: 72
		>>> division(120, 4)
		La division es: 30.0

if __name__ == "__main__":

Básicamente, lo que se hace usando **if __name__ == " __main__":** es ver si el módulo ha sido ejecutado directamente o no (importado). Si se ha ejecutado como programa principal se ejecuta el código dentro del condicional.

Una de las razones para hacerlo es que, a veces, se escribe un módulo (un archivo .py) que se puede ejecutar directamente pero que, alternativamente, también se puede importar y reutilizar sus funciones, clases, métodos, etc en otro módulo.

Con el uso del condicional conseguimos que la ejecución sea diferente al ejecutar el módulo directamente que al importarlo desde otro módulo. Todo lo que hay dentro del condicional será completamente *ignorado* por el intérprete cuando se importa el módulo, pero no cuando se ejecute como módulo principal.

mi_modulo.py	principal.py	Si ejecutas el script mi_modulo.py:

print("¡Hola desde mi_modulo.py!")
def hacer_algo():
  print("¡Soy una función!")
if __name__ == "__main__":
  print('Ejecutando como programa principal')
  hacer_algo()
print("¡Adiós desde mi_módulo.py!")
		¡Hola desde mi_modulo.py!
		Ejecutando como programa principal
		¡Soy una función!
		¡Adiós desde mi_módulo.py!

		el valor de __name__ será "__main__" y se ejecutará el bloque que hay dentro del if __name__ == "__main__":

mi_modulo.py	principal.py	Si ejecutas el script principal.py:

print("¡Hola desde mi_modulo.py!")
def hacer_algo():
  print("¡Soy una función!")
if __name__ == "__main__":
  print('Ejecutando como programa principal')
  hacer_algo()
print("¡Adiós desde mi_módulo.py!")
	import mi_modulo
	
	print("¡Hola desde principal.py!")
	mi_modulo.hacer_algo()
	print("¡Adiós desde principal.py!")
		¡Hola desde mi_modulo.py!
		¡Adiós desde mi_módulo.py!
		¡Hola desde principal.py!
		¡Soy una función!
		¡Adiós desde principal.py!

		no se cumple la condición if __name__ == "__main__" por lo que solo se ejecuta el código global (el print inicial y se crea el objeto función hacer_algo en memoria). La función solo se ejecuta cuando la llamamos desde el módulo principal

mi_modulo.py	principal.py	Si ejecutas el script principal.py:
print("¡Hola desde mi_modulo.py!") def hacer_algo(): print("¡Soy una función!") hacer_algo() print("¡Adiós desde mi_módulo.py!")	import mi_modulo print("¡Hola desde principal.py!") mi_modulo.hacer_algo() print("¡Adiós desde principal.py!")	¡Hola desde mi_modulo.py! ¡Soy una función! ¡Adiós desde mi_módulo.py! ¡Hola desde principal.py! ¡Soy una función! ¡Adiós desde principal.py!
		al importar ejecutamos todo el código del módulo importado, *incluyendo la llamada a la función que ahora no está envuelta en el condicional.* Una vez importado el módulo principal llama a la función importada de nuevo.

- Cuando el intérprete lee un archivo de código, ejecuta todo el código global que se encuentra en él. Esto implica crear objetos para toda función o clase definida y variables globales.
- Todo módulo (archivo de código) en Python tiene un atributo especial llamado __name__ que define el espacio de nombres en el que se está ejecutando. Es usado para identificar de forma única un módulo en el sistema de importaciones.
- Por su parte "__main__" es el nombre del ámbito en el que se ejecuta el código de nivel superior (programa principal).
- El intérprete pasa el valor del atributo __name__ a la cadena '__main__' si el módulo se está ejecutando como programa principal.
- Si el módulo no es llamado como programa principal, sino que es importado desde otro módulo, el atributo __name__ pasa a contener el nombre del archivo en sí.