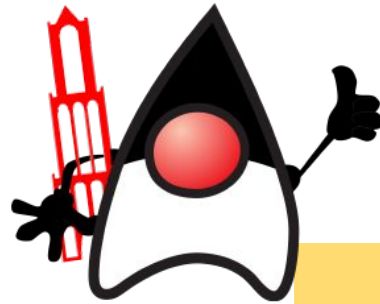




Curso FullStack Python

Codo a Codo 4.0



Python

Parte 4



Funciones en Python

Es un grupo de instrucciones que constituyen una unidad lógica del programa y resuelven un problema **específico**, nos sirven para trabajar con **programación modular** (modularidad del código). Debemos *pensar modularmente*, es decir, pensar el programa en bloque, en **módulos**, en *"partes de un programa"*.

¿Cuáles son sus beneficios?

- **Trabajo en equipo:** cada uno puede aportar algo al programa y es más fácil corregir errores
- **Reutilización del código:** una solución ya pensada y probada nos sirve para una parte de nuestro programa, a veces con pequeños cambios o a veces tal cual como fue probada.
- **Simplificar la lectura y reducir el cuerpo principal del programa:** lo modular nos permite pensar por partes, simplificar el problema y ayudar a la lectura del código a medida que va creciendo el programa.
- **Organización y encapsulamiento:** Dividir y organizar el código en partes más sencillas y encapsular el código que se repite a lo largo de un programa para ser reutilizado.
- **Mantenimiento:** El programa va ser más fácil de mantener. Evita tener código redundante.

Python ya tiene funciones predefinidas, por ejemplo: **len()**, que obtiene el número de elementos de un objeto contenedor como una lista, una tupla, un diccionario o un conjunto y **print()**, que muestra por consola un texto.

Programa 1

```
inst1
inst2
inst3
inst4
inst1
inst2
inst3
inst5
inst6
inst7
inst1
inst2
inst3
```

```
def mi_funcion():
    inst1
    inst2
    inst3
```

Programa 2

```
def mi_funcion():
    inst1
    inst2
    inst3

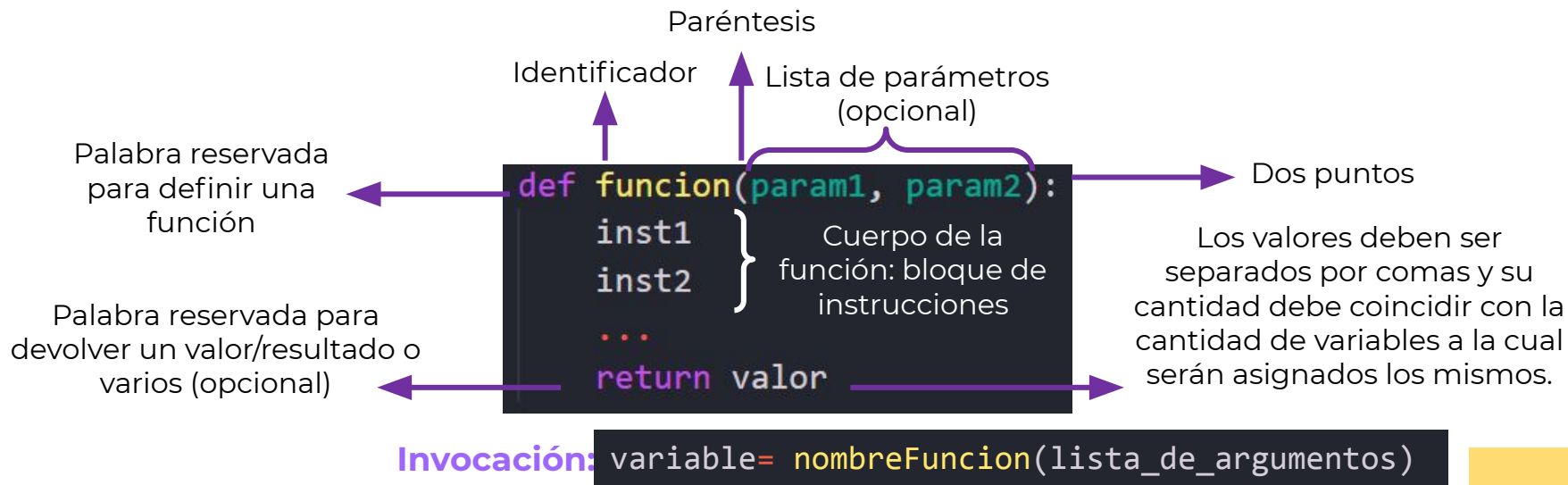
mi_funcion():
inst4
mi_funcion():
inst5
inst6
inst7
mi_funcion():
```

*El siguiente programa define una función llamada **mi_función()** que contiene 3 instrucciones. Llamando a la función con **mi_función()** evitamos escribir el mismo código 4 veces (reutilización de la función).*

Definiendo una función



- **Primera línea: cabecera o definición de la función:**
 - o **def:** palabra reservada que define una función.
 - o **nombre o identificador:** se utiliza para invocarla (llamarla).
 - o **Parámetros:** encerrados entre paréntesis, son opcionales
 - o **Dos puntos:** indican el cierre de la cabecera
- **Cuerpo:** con un sangrado mayor (en general 4 espacios), son las instrucciones que se encapsulan en dicha función y que le dan significado.
- **Return:** Es opcional y nos permite devolver un resultado.



Ejemplo: Definiendo una función

```
# sintaxis básica para definir (declarar) una función en Python
```

PY

```
def imprimir_mensaje():  
    print("Hola soy una función")
```

```
#Cuerpo principal
```

```
imprimir_mensaje()
```

Hola soy una función

terminal

Solemos nombrar a las funciones con verbos en infinitivo que denoten lo que va a hacer esa función, por ejemplo: **imprimir_mensaje**

Primero definimos la función (**def nombre_funcion()**) y luego la llamamos/invocamos en el programa principal: **nombre_funcion()**.

Una función sin parámetros realiza un objetivo sin requerir información.



funciones_intro.py

```
def imprimir_mensaje_cinco_veces():  
    for i in range(5):  
        print("Este es el mensaje " + str(i))
```

PY

Ejemplo de función que imprime mensaje 5 veces

```
imprimir_mensaje_cinco_veces()
```

Llamada a la función

```
Este es el mensaje 0  
Este es el mensaje 1  
Este es el mensaje 2  
Este es el mensaje 3  
Este es el mensaje 4
```

terminal

Parámetros de una función

Son los **datos de entrada** de una función.

Cuando queremos invocar o llamar a una función debemos escribir su nombre como si fuera una instrucción más, pasando los argumentos necesarios según los parámetros que defina la función, en caso de ser necesario:

```
def imprimir_mensaje_N_veces(N): #Tiene 1 parámetro
    for i in range(N):
        print("Este es el mensaje " + str(i))

imprimir_mensaje_N_veces(3) #Ejemplo sin datos ingresados por el usuario

veces= int(input("Ingrese la cantidad de veces que desea imprimir: "))
imprimir_mensaje_N_veces(veces) #Ejemplo con datos ingresados por el usuario
```

PY



funciones_parametros.py

Parámetros de una función

Caso con dos parámetros:

```
def mensaje_personalizado_N_veces(N, mje): #Tiene 2 parámetros
    for i in range(N):
        print(mje)
#Llamada a la función
mensaje_personalizado_N_veces(4, "Python")
```

PY

Variante: función con 2 datos de entrada que recibe como parámetros proporcionados por el usuario. Usamos la misma función pero le pasamos valores nosotros.

Python
Python
Python
Python

terminal

```
cant = int(input("¿Cuántas veces se repetirá el valor? "))
mensaje = input("¿Cuál es el mensaje? ")
mensaje_personalizado_N_veces(cant, mensaje)
```

PY

Con valor 3 y "Probando" devuelve:
¿Cuántas veces se repetirá el valor? 3
¿Cuál es el mensaje? Probando

Probando
Probando
Probando

terminal

Parámetros de una función

Esta es una variante validando el código:

```
cant = int(input("¿Cuántas veces se repetirá el valor? "))
while cant <= 0: #Validamos que el número sea positivo
    print("Dato no válido!")
    cant = int(input("¿Cuántas veces se repetirá el valor? "))
mensaje = input("¿Cuál es el mensaje? ")
```

PY

Ejemplo utilizando f-string: Esta función define un parámetro llamado **numero** que es el que se utiliza para multiplicar por 5.

```
def multiplicar_por_5(numero):
    print(f'{numero}*5 = {numero * 5}')
```

PY

```
print("Comienzo del programa: ")
multiplicar_por_5(7)
print("Siguiete instrucción")
multiplicar_por_5(113)
print("Fin del programa")
```

```
Comienzo del programa:
7*5 = 35
Siguiete instrucción
113*5 = 565
Fin del programa
```

El programa comienza su ejecución y va ejecutando las instrucciones una a una de manera ordenada. Cuando se encuentra el nombre de la función **multiplicar_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.

Parámetros de una función

Diferencia entre parámetro y argumento

La función `multiplicar_por_5()` define un **parámetro** llamado `numero`. Sin embargo, cuando desde el código se invoca a la función, por ejemplo, `multiplicar_por_5(7)`, se dice que se llama a `multiplicar_por_5` con el **argumento** 7.

Entonces, en la definición de la función se llaman **parámetros**, mientras que en la llamada se llaman **argumentos**. Es decir que defino la función con determinados parámetros y llamo a la función con determinados argumentos.

```
def multiplicar_por_5(numero):  
    print(f'{numero}*5 = {numero * 5}')
```



```
multiplicar_por_5(7)
```

Parámetro

Argumento

Más sobre parámetros de las funciones

Paso por valor y paso por referencia

- **Paso por valor:** el paso por valor de los argumentos, **hace una copia** del valor de las variables en los respectivos parámetros. Cualquier modificación del valor del parámetro, **no afecta** a la variable externa correspondiente porque estamos modificando la copia.
- **Paso por referencia:** el paso por referencia copia 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. Ejemplo: listas.

Ejemplos y más información:

<https://docs.hektorprofe.net/python/programacion-de-funciones/paso-por-valor-y-referencia/>

Más sobre parámetros de las funciones

Tipos de parámetros en una función

Por defecto, los valores de los argumentos se asignan a los parámetros en el mismo orden en el que los pasamos al llamar a la función. Si llamamos a una función y no le pasamos todos los argumentos el intérprete lanzará una excepción.

Parámetros opcionales

En una función Python se pueden indicar una serie de parámetros opcionales con el operador `=`. Son parámetros que tienen un valor por defecto y si no se pasan al invocar a la función entonces toman este valor.

```
# Parámetros opcionales y por defecto
def saludo(nombre, mensaje="encantado de saludarte"):
    print("Hola {}, {}".format(nombre, mensaje))
saludo("Juan Pablo")
saludo("Juan Pablo", "¿Cómo estás?")
```

PY

El parámetro **nombre** no indica un valor por defecto (es obligatorio). El parámetro **mensaje** tiene un valor por defecto (encantado de saludarte). En caso de no pasar este argumento, se tomará dicho valor por defecto. Por el contrario, si se indica, se sobrescribirá 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.

Uso de parámetros opcionales

En este ejemplo vamos a usar argumentos opcionales.
Calcularemos la raíz de un número:

```
# Parámetros por defecto
def fn_raiz(num, raiz=2): #Si no recibe un segundo parámetro
    #calcula la raiz cuadrada (2)
    return num**(1/raiz)

# Prog Ppal
print(fn_raiz(4))
print(fn_raiz(8))
print(fn_raiz(8,3))
```

PY

Este caso tiene dos parámetros, el número y la raíz, que no necesariamente tiene que estar y si no está tomo por defecto el 2.

Más sobre parámetros de las funciones

Parámetros posicionales y parámetros con nombre

Recordemos que, al invocar una función con diferentes argumentos, los valores se asignan a los parámetros **en el mismo orden** en que se indican.

En el ejemplo anterior si llamamos a la función `saludo("Juan Pablo", "¿Cómo estás?")` al parámetro **nombre** se le asignará el valor "Juan Pablo" 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:

```
# Parámetros posicionales y parámetros con nombre
saludo(mensaje="¿Cómo estás?", nombre="Juan Pablo")
saludo(nombre="Juan Pablo", mensaje="¿Cómo estás?")
saludo("Juan Pablo", mensaje="¿Cómo estás?")
```

PY

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.

Resumen: parámetros de las funciones

Posicionales	Por defecto
Se encuentran al comienzo de la lista de parámetros y se relacionan con los argumentos por su posición.	Se encuentran al final de la lista de parámetros y se relaciona con los argumentos por su posición o por nombre. Si no se especifica según nombre o posición toma el valor que tiene por defecto. Tiene restricciones en la forma de uso para accederse por posición.

Funciones que devuelven valores (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 (retornar) un valor, si queremos verlo a ese valor hay que **imprimirlo**.
- 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.
- No se debe tener el mismo nombre de la función que la variable interna que utilice. Lo que devuelve (return) no es la variable sino el **contenido** de la variable.

```
def restar(num1,num2): # recibe 2 parámetros
    ''' restar dos valores numéricos. '''
    resta= num1-num2
    return resta # La fx retorna (devuelve) un valor
# Prog Ppal
resultado= restar(10,3)
print("El 1er resultado de la resta es:", resultado)
print("El 2do resultado de la resta es:", restar(10,4))
```

PY

La función retorna el **contenido** de la variable resta.

En el primer caso imprimo el valor de la variable (resultado)

En el segundo caso podría imprimir lo que retorna la función restar(x,y)



funciones_return.py

Funciones que devuelven valores (return)

Ejemplo: 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) #nada, porque no es par
```

PY

Ejemplo: return que devuelve un valor u otro (con if)

La siguiente función muestra por pantalla si el número es par o no:

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

PY

Funciones que devuelven valores (return)

Ejemplo: Devolver más de un valor con return

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(2)  
print(cuad, cubo)
```

PY

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 = [] #creamos la lista  
    for i in range(11):  
        resultados.append(numero * i)  
    return resultados  
  
res = tabla_del(3)  
print(res)
```

PY

Funciones que devuelven valores (return)

Podemos tener una **función que recibe listas**, como en este ejemplo:

```
def sumar_lista(lista): #pre: recibe una lista.
    suma= 0 # variable que almacenará la sumatoria
    lista[0]= 100 #al primer valor le asigna el número 100
    for elem in lista:
        suma+=elem #acumulador de elementos
    return suma #pos: devuelve la sumatoria de los elementos de la lista.

# Llamada a la función:
numeros= [1,2,10,-5]
print("La suma es: " + str(sumar_lista(numeros)))
print(numeros)
```

PY

Devuelve...

La suma es: 107 □ La lista números va a quedar modificada porque en la función le dije que el valor 0 es 100

[100, 2, 10, -5] □ Toda lista que se modifica en la función queda modificada en el programa principal, por eso más allá que le diga que inicia con 1 la función la modifica y la hace iniciar con 100.



funciones_listas.py

Funciones que devuelven más de un valor (return)

Se puede llamar una función desde otra función.

Por ejemplo: si quiero hacer la suma y el promedio y ya calculé la suma con otra función llamo a esa función suma y luego calculo el promedio.

En el return se pone **return suma, promedio** porque devuelve dos valores.

```
def calcular_suma_prom(lista):  
    ''' retornar la sumatoria y el promedio de una lista. '''  
    suma= sumar_lista(lista) # Llamando a una fx dentro de otra fx  
    prom= suma/len(lista) #Calculo el promedio como la suma dividido  
        #la cantidad de elementos de la lista  
  
    return suma, prom # La fx retorna 2 (dos) valores  
        # con el return, salimos de la fx  
  
# Llamado a una fx que retorna 2 valores  
result1,result2 = calcular_suma_prom(numeros) # En cada variable guardamos  
        # lo que retorne la función  
print("La sumatoria es:", result1, "y el promedio es:", result2)
```

PY

Devuelve: La sumatoria es: 107 y el promedio es: 26.75

Ejemplo integrador

Ejemplo de carga de listas con valores positivos: validación y muestra

```
def ingresar_positivo():  
    cant= int(input("Ingrese un número: "))  
    while cant<=0:  
        print("Dato no válido!")  
        cant= int(input("Ingrese un número: "))  
    return cant
```

PY

```
def crear_lista(N):  
    lista= []  
    for i in range(N):  
        valor= ingresar_positivo()  
        lista.append(valor)  
    return lista
```

PY

```
def mostrar_lista(lista):  
    for valor in lista:  
        print(valor, end=" ")  
    print()
```

PY

```
# Prog Ppal  
N=int(input("¿Cuántos valores tendrá la lista?: "))  
numeros= crear_lista(N)  
mostrar_lista(numeros)
```

PY

Ámbito y ciclo de vida de las variables

Las variables están definidas dentro de un **ámbito** que determina dónde la variable puede ser utilizada. El *ciclo de vida de una variable* se refiere al tiempo en que una variable permanece en memoria.

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, es decir que 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.

Entonces: **las variables dentro de la función son locales**, no son para el programa principal, es una variable definida a nivel local, su ámbito de aplicación es a nivel local de la función y no existe para el programa principal.

```
def saludo(nombre):  
    x = 10  
    print(f'Hola {nombre}')  
saludo('Luis')  
print(x) # NameError: name 'x' is not defined
```

PY



variables.py

Al tratar de mostrar por pantalla el valor de la variable x, el intérprete mostrará un error.

Ámbito y ciclo de vida de las variables

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



```
x = 20  
muestra_x() # x vale 10  
print(x) #20
```

PY

En este ejemplo, 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 afuera, 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 #Global  
def muestra_x():  
    x = 10 #Local  
    print(f'x vale {x}')
```



```
    print(f'y vale {y}')
```



```
muestra_x() # x vale 10; y vale 20
```

PY

Alcance y visibilidad de las variables

Variables globales: Se pueden acceder en cualquier parte del programa.

Variables locales: Solo son visibles en el ámbito donde fueron declaradas.



Variables libres: Son visibles en una sub-función, pero no son accesibles desde el programa principal.



Buenas prácticas

- No utilizar variables globales desde dentro de una función
- No anidar definiciones de funciones
- Si se desea utilizar los valores de una variable del programa principal en una función, se debe pasar por parámetro

Ejemplos sobre funciones

- **Función que devuelve un valor:** ver archivo **funciones1.py**
 - **Función que devuelve varios valores:** ver archivo **funciones2.py**
 - **Uso de parámetros por defecto:** ver archivo **funciones3.py**
 - **Visibilidad de las variables:** ver archivo **variables_visibilidad.py**
- 
- 

Funciones lambda

También conocidas como **funciones anónimas**, ya que se definen sin un nombre. A diferencia de las funciones que se definen con la palabra reservada **def**, estas funciones anónimas se definen con la palabra reservada **lambda**. Su sintaxis es:

```
lambda parámetros: expresión
```

Sus principales características son:

- 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.
- Están restringidas al uso de una sola expresión.
- Se suelen usar en combinación con otras funciones, generalmente como argumentos de otra función.

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

PY



funciones_lambda.py

*En este ejemplo, x es el parámetro y $x ** 2$ la expresión que se evalúa y se devuelve. Esta función no tiene nombre y toda la definición devuelve una función que se asigna al identificador cuadrado.*

Funciones lambda

Comparación entre una función anónima y una función normal:

```
def alCubo(x):  
    return x*x*x  
  
cubo = lambda x: x*x*x  
print(alCubo(3)) #27  
print(cubo(5)) #125
```

PY

La función lambda map()

Una función lambda combinada con otras funciones adquiere más potencia, por ejemplo con **map()**.

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

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

Funciones lambda

Ejemplo: lista de enteros y sus cuadrados

Esta sería una solución sin usar funciones lambda:

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

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

PY

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]
```

PY

Un ejemplo más interesante es, en lugar de pasar 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]
```

PY

Funciones lambda: resumen

- Permite definir funciones pequeñas y anónimas
- Tienen el mismo comportamiento que las funciones definidas con **def**

Sintaxis

Definición

<nombre función> = lambda <lista de parám separados por ,> : <expresión>

Invocación

<nombre función> (<lista de argumentos separados por ,>

Ejemplo:

```
>>> suma = lambda x, y: x + y
```

```
>>> suma (5, 3)
```

```
8
```

Su equivalente usando una función def sería

```
>>> def suma ( x , y ):
```

```
    return x + y
```

```
>>> suma ( 5 , 3 )
```

```
8
```

- La funciones están restringidas a que devuelva un solo resultado
- La función lambda no necesariamente se asocia con un nombre
- Antes de usar una función lambda verificar si esta es o no más clara que una función regular.

Docstrings (documentar funciones)


Son un tipo de comentarios especiales que se usan para documentar un módulo, función, clase o método. Son la primera sentencia de cada uno de ellos y se encierran entre tres comillas simples o dobles. Nos permite documentar la función, es decir, indicar qué hace dicha función. Permiten generar la documentación de un programa. Además, suelen utilizarlos los entornos de desarrollo para mostrar la documentación al programador de forma fácil e intuitiva. Todos los módulos deberían tener normalmente docstrings, y todas las funciones y clases exportadas por un módulo también deberían tenerlos. En objetos, el docstring se convierte en el atributo especial `__doc__` de ese objeto.

```
def suma(a, b):  
    """Esta función devuelve la suma de los parámetros a y b"""  
    return a + b
```

PY

Consulte los PEP [257](#) y [258](#) para obtener más información sobre este tema.

Docstrings (documentar funciones): ejemplo



```
def cuad(x):  
    """Dado un número x, calcula x²"""  
    return x * x # También podríamos haber hecho x ** 2  
  
def modulo_vector(x, y):  
    """Calcula el módulo de un vector 2D.  
    Argumentos:  
        x: (float|int) coordenada de las abscisas  
        y: (float|int) coordenada de las ordenadas  
    Devuelve: (float) el módulo del vector  
    """  
    return (x ** 2 + y ** 2) ** 0.5
```

PY

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.

Un **módulo** es 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.

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.

Para **crear** un módulo debemos hacer lo siguiente:

1. Crea en una carpeta un fichero llamado mis_funciones.py con el siguiente contenido:

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

PY

Módulos y paquetes

Para **utilizar** el módulo debemos importarlo:

2. Ve a otro módulo que tengas en la misma carpeta y escribe:

```
import mis_funciones
```

3. Para llamar a la función podremos acceder a sus definiciones a través del operador punto .

```
mis_funciones.saludo("Juan Pablo")
```

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

from ... import ...

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

```
from mis_funciones import saludo, otra_funcion  
saludo("Juan")  
otra_funcion()
```

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



mis_funciones.py

Módulos y paquetes

from ... import *

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

```
from mis_funciones import *  
saludo("Juan")  
otra_funcion()
```

PY

***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("Juan Pablo") #Hola Juan Pablo
```

PY



Paquetes

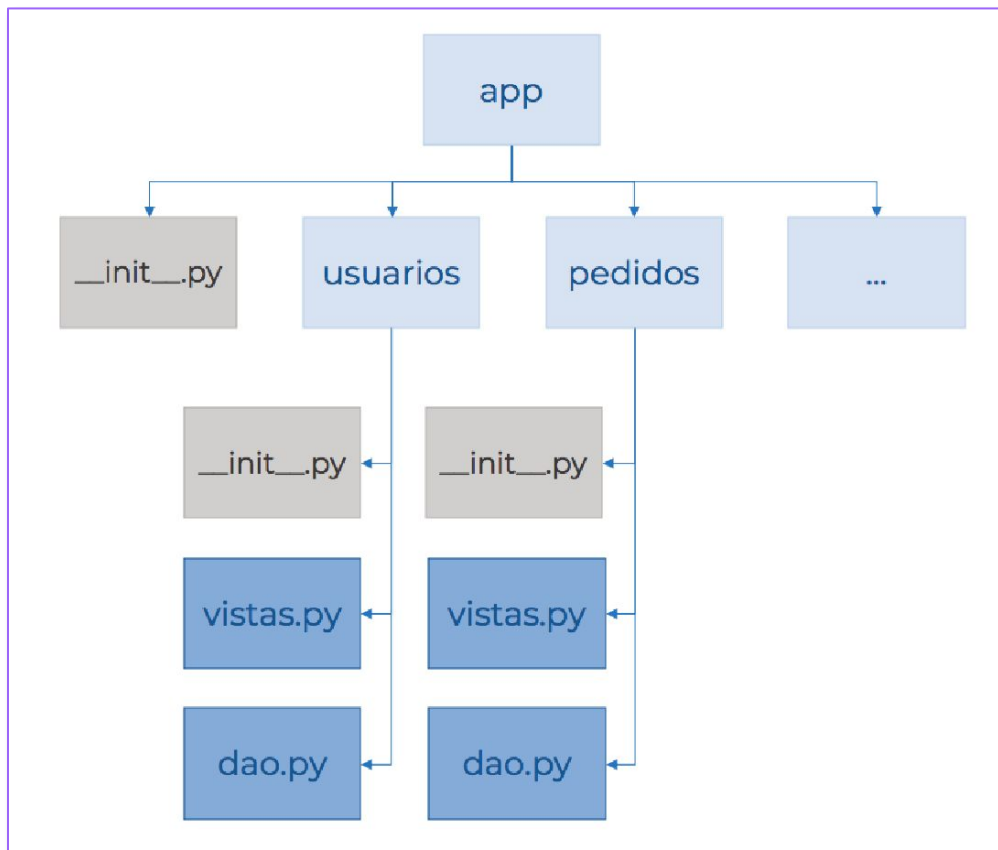
Del mismo modo en que agrupamos las funciones y demás definiciones en módulos, los **paquetes** en Python permiten organizar y estructurar de forma jerárquica los diferentes módulos que componen un programa. Además, los paquetes hacen posible que existan varios módulos con el mismo nombre y que no se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

Paquetes: ejemplo

Imagina que estás haciendo una aplicación para gestionar pedidos. Una forma de organizar los diferentes módulos podría ser la siguiente:



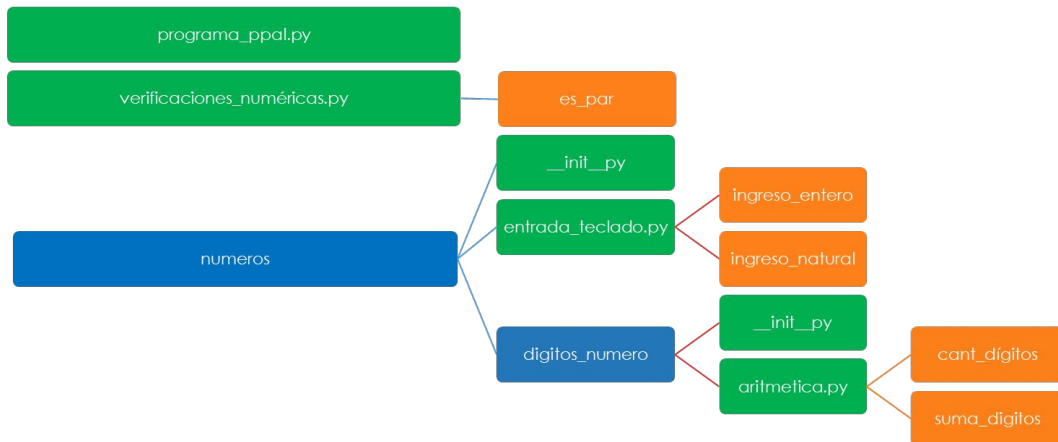
Módulos y paquetes: resumen y ejemplo

- **Módulo:** archivo que contiene un programa
- **Paquete:** conjunto de módulos que se encuentran en una carpeta. Este debe contener el archivo `__init__.py` el cual puede estar vacío
- **Sub-paquete:** un paquete puede contener otros paquetes
- **Namespace:** permite acceder a un elemento del módulo importado

Los módulos no necesitan pertenecer a un paquete. Para acceder a estos se utiliza la instrucción `import`.

Para acceder a un elemento de un modulo se debe especificar la ruta

Se dispone la siguiente estructura de paquetes, sub-paquetes y módulos



Módulos:

```
# aritmetica.py
def cant_digitos(n):
    ''' Tarea: calcula cantidad de dígitos de un número '''
    cant = 0
    while n > 0:
        cant = cant + 1
        n = n // 10
    return cant

def suma_digitos(n):
    ''' Tarea: calcula suma de los dígitos de un número '''
    suma = 0
    while n > 0:
        suma = suma + n % 10
        n = n // 10
    return suma
```

```
# entrada_teclado.py
def ingreso_entero():
    ''' Tarea: ingresar un número entero '''
    cn = input('Nro entero: ')
    while float(cn) != int(float(cn)):
        cn = input('Nro entero: ')
    return int(cn)

def ingreso_natural():
    ''' Tarea: ingresar un número natural '''
    cn = input('Nro natural: ')
    while float(cn) != int(float(cn)) or int(
float(cn)) <= 0 :
        cn = input('Nro natural: ')
    return int(cn)
```

```
# verificaciones_numericas.py
def es_par(n):
    ''' Tarea: determina si el número es par '''
    if n%2 == 0:
        return True
    else:
        return False
```

Módulos:

```
# programa_ppal.py
import verificaciones_numericas
import entrada_teclado
import aritmetica
'''
    Tarea: permite ingresar números
    enteros hasta que se ingresa el valor 0.
    Además informa cantidad de números pares
    ingresados y cantidad de números de dos cifras se
    ingresaron.
'''
cant_pares = 0
cant_nros_dos_dig = 0
n = entrada_teclado.ingreso_entero()
while n != 0:
    if verificaciones_numericas.es_par(n):
        cant_pares = cant_pares + 1
    if aritmetica.cant_digitos(n) == 2:
        cant_nros_dos_dig = cant_nros_dos_dig + 1
    n = entrada_teclado.ingreso_entero()

print('Cantidad de pares : ', cant_pares)
print('Cantidad de numeros de dos dígitos : ', cant_nros_dos_dig)
```