Unidad 2

Programacion Modular

1º Programación

Programación modular

Hasta ahora hemos visto lo que es la **programación estructurada**. En esta unidad, abordaremos la programación modular.

La **programación modular** es un paradigma de programación que consiste en **dividir un programa en módulos o subprogramas** con el fin de hacerlo más **legible y manejable**.

Al aplicar la programación modular, un problema complejo debe ser dividido en varios subproblemas más simples, y estos a su vez en otros subproblemas más simples.

Programación modular

Diseña Y Codifica:

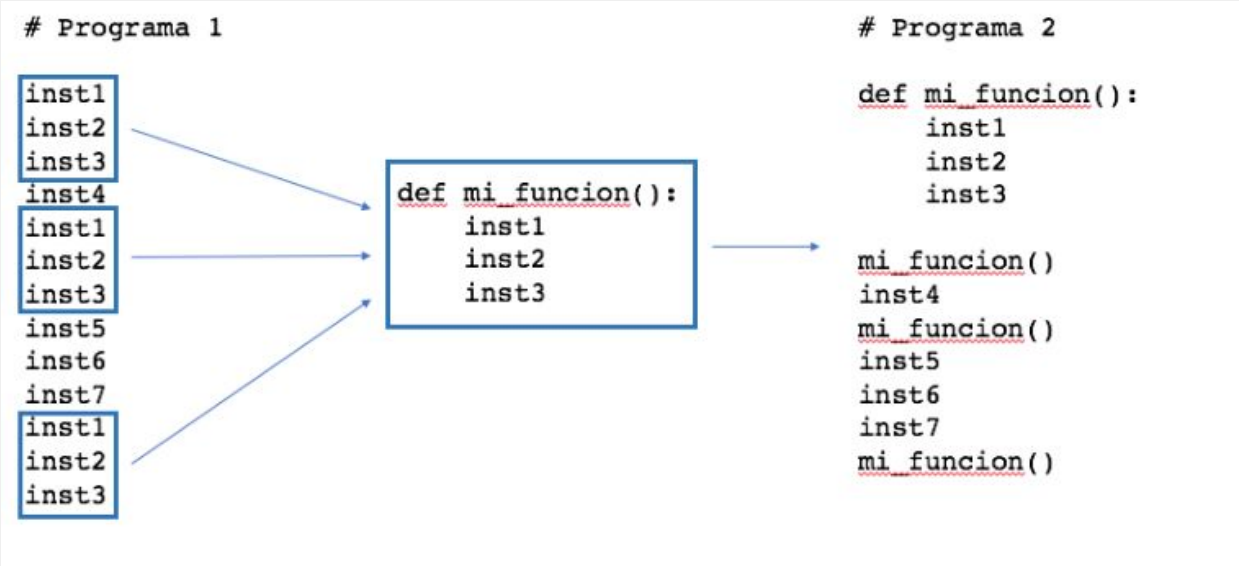
Un programa que pida al usuario que introduzca una cadena que debe tener más de 4 caracteres, si no lo tiene que la pida otra vez

Introduzca un número

Si el número es:

- par → calcule una cadena con el carácter 2 y el 4 concatenado tantas veces como sea el número introducido.
- si es múltiplo de 3 → calcule una cadena con el carácter 1 y 2 concatenado tantas veces como el número introducido.
- si es múltiplo de 7 → calcule una cadena con el carácter 0 y 3 concatenado tantas veces como el número introducido.

Funciones



Una **función** es un trozo de código que, tras ser definido, puede ser usado muchas veces invocando al lugar donde fue definido.

A su vez, ese código podrá ser invocado desde otros ficheros Python.

Las funciones constituyen un elemento esencial en los lenguajes de programación y nos permiten agrupar nuestro código en unidades con características o funcionalidades similares.

Funciones

Los **grupos de funciones** se reúnen así en lo que se conocen como **librerías o módulos**, donde encontramos funciones agrupadas por ámbitos.

Para usar una función que se incluye en otro fichero o librería usamos la instrucción import. Ya has usado algunas librerías como random o math.

```
import math, random
```

Definición de una función

La definición de una función se inicia con la palabra clave def seguida del nombre descriptivo de la acción que realiza la función.

```
def nombre_accion_funcion (parametro_1, parametro_1,..., parametro_n):
    linea 1 de código de la función
    ...
    linea n de código de la función
    return valor (opcional)
```

Definición de una función: return

Ante una llamada a función, el flujo del programa saltará a la primera instrucción del bloque de definición de la función y seguirá ejecutando este bloque hasta que:

- finalice
- o hasta que encuentre un return. Return hace que termine la ejecución de la función cuando aparece y el programa continúa por su flujo normal.

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. Aunque no es lo más elegante.

¿Y qué pasa si quiero devolver más de un valor? Para estos casos, la variable de retorno debe ser una lista o tupla.

Definición de una función

Vamos a definir una función que reciba dos números y devuelva su suma:

```
def suma(numero_1, numero_2):
    resultado = numero_1 + numero_2
    return resultado
```

A partir de ahora, cuando tenga que sumar dos número puedo usar mi función llamada suma.

Ejemplo de una función

Vamos a usar esa función . La llamada a la función sería de la siguiente forma:

```
suma(3, 2)
```

Mi función suma devuelve un valor con el resultado.

A partir de ahora, cuando tenga que sumar dos número puedo usar mi función llamada suma.

Importante. la llamada a la función debe ir, por lo general, fuera del bloque de definición de la función.

Llamada a una función

Una posible llamada para el ejemplo anterior sería:

```
print(suma(53, 2))
```

Ahora prueba a realizar la siguiente llamada:

```
print(suma("cadena ", "de texto"))
```

Además, el valor de retorno de una llamada puede ser el argumento de entrada para otra llamada a otro método:

```
suma(suma(53, 2), 3)
```

Primero se resolverá suma(53, 2) y una vez que haya devuelto su resultado, se hará la llamada más externa a suma

Argumentos por defecto

En Python podemos asignar un valor por defecto a los argumentos de una función. De forma que si lo recibe, ese será su valor pero si no lo recibe en la llamada, tomará el valor por defecto. Por ejemplo:

```
def saludar(nombre = "Desconocido"):
    print("Buenas tardes, señor ", nombre)
```

La llamada podrá realizarse de dos formas:

```
print(saludar()) # Nombre toma el valor desconocido
```

```
print(saludar("Pepe"))# Nombre toma el valor  Pepe
```

Módulos de funciones

Las funciones no sólo nos permiten reutilizar el código dentro del mismo fichero. También podemos reutilizar esas funciones desde otros ficheros.

Un **módulo** es un fichero que agrupa definición y declaración de funciones. De esta forma, podríamos organizar el código de manera que las funciones de un fichero tenga una temática común.

Es tu turno:

Investiga para qué sirve y qué funciones tienen las siguientes librerías Python:

- datetime
- random
- math
- sys
- os

Usar funciones de otros ficheros

Opción 1:

```
import menuFunciones #importamos el fichero
menuFunciones.operacionA()
#Llamamos a la función mediante el nombre del fichero.nombreFuncion
```

Opción 2:

```
from menuFunciones import operacionA, operacionB
#Importamos las funciones directamente de este modo
operacionA() #Ahora en la llamada no tengo q poner el fichero

from menuFunciones import * #importo todas las funciones
```

Variables locales y globales

- **Variables locales:** Son las que se declaran **dentro de la función** y no son accesibles desde fuera. Es decir, las variables locales son aquellas de ámbito local. Los parámetros de las funciones también se consideran variables locales, y por tanto su ámbito también es local.
- **Variables globales:** Se declaran fuera de las funciones, en el cuerpo principal del programa. Son accesibles tanto dentro de las funciones como fuera. También se puede acceder a las variables globales de otros programas o módulos importados. Por esto decimos que las variables globales son aquellas de ámbito global.

Variables locales y globales

- **Variables globales:** Se declaran fuera de las funciones y son accesibles tanto dentro de las funciones como fuera.

```
def operar(a,b):
    suma = a + b
    resta = a - b
    multiplica = a * num2
    print(suma)
    print(resta)
    print(multiplica)


num1 = 10
num2= 5
operar(num1, num2)
```

Variables locales y globales

Si intentamos acceder al valor de una variable local desde el cuerpo principal del programa o, en general, a una variable que no ha sido definida obtendremos un error típico de Python: ***NameError***.

```
def saludar():
    saludo = '¡Hola!'
    print(saludo)


saludar()
print(saludo)
```

```
NameError: name 'saludo' is not defined. Did you mean: 'saludar'?
```

Variables locales y globales

Las variables globales son accesibles desde otros ficheros o módulos de la misma forma que se importaba una función.

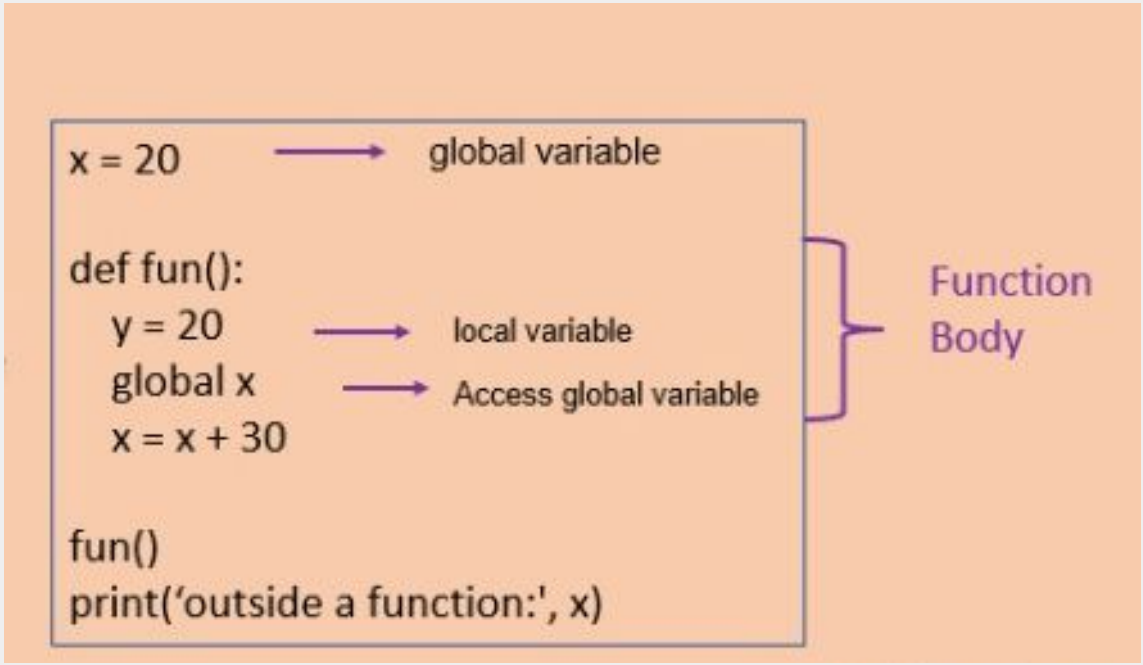
```
from modulo import saludo #importamos la variable saludo de modulo


def saludar():
    print(saludo)


saludo = saludar()
```

Variables locales y globales

¿Cómo modificar una variable global dentro de una función?

Antes de explicarte cómo, decirte que aunque Python lo permite, no es algo recomendable.

Para ello usamos la palabra reservada global



Variables locales y globales

A veces puede suceder, que dentro de una función existe una variable local con el mismo nombre que una variable global, es decir, sus nombres entran en **conflicto**.

Tienes que saber que cuando hacemos referencia a una variable Python busca **primero en el ámbito local** actual para ver si encuentra dicha variable y **si no, la busca en el ámbito global**. Es decir, en cierta forma, el ámbito local tiene preferencia sobre el global.

Paso de parámetros por valor o referencia

Al realizar una llamada podemos pasar los argumentos de dos formas:

- **Por valor**: En la llamada lo que aparece es un valor de un tipo concreto.
- **Por referencia**: En este caso, usamos una variable que tendrá un valor y un tipo.

En los pasos por referencia, debes tener en cuenta si la variable que pasas es inmutable o mutable.

- Si pasamos un objeto **inmutable**, aunque la variable sea modificada dentro del método, su valor no cambiará.
- Sin embargo, si el argumento es de un tipo **mutable**, sí podremos cambiar su valor. Esto NO es recomendable en ningún lenguaje de programación. Si deseamos devolver algo, lo hacemos con el return y construyendo la variable de salida.

Paso de parámetros por valor o referencia

Objetos inmutables:

```
def suma(n1, n2):
    n1 = n1+1
    return n1+n2


n1 = 2
n2= 3
print(suma(n1,n2))
print(n1)
```

¿Cuánto vale n1 al final de la ejecución?

Paso de parámetros por valor o referencia

Objetos mutables:

```
def suma(n1, n2):
    n1.insert(0, n2)
    return n1

n1 = [2, 3, 4]
n2= 3
print(suma(n1,n2))
print(n1)
```

¿Cuánto vale n1 al final de la ejecución?

Paso de parámetros por valor o referencia

```
#inmutable: tipos primitivos  Por valor
num1 = 3
num2 = num1 # copio el valor 3 a num2
num1 = num1+2
print(num1)
print(num2)


#mutable: ejemplo lista Por referencia
lista1 = [1,2,3]
lista2 = lista1 # lista2 apunta a la misma zona de memoria
(referencia de memoria)
lista1.append(6)
print(lista1)
print(lista2)
```

Tests

Usaremos la función assert para comprobar que la ejecución de una función se comporta como esperamos.
La función assert puede tener dos comportamientos:

1. Devuelve True si la comparación entre lo que se esperaba y lo que se ha devuelto es cierta.
2. Sin embargo, lanzará una excepción (`AssertionError`) si no lo que devuelve la función no es igual a lo que le decimos.

```
assert(nombreFuncion(par1, …parn)== valor)
```

Ejemplos:

```
def cuadradoDeUnNumero(numero):
    return numero**2
assert(cuadradoDeUnNumero(9)> 0)
assert(cuadradoDeUnNumero(9)==81)
assert(cuadradoDeUnNumero(-9)==81)
assert(cuadradoDeUnNumero(0)==0)
assert(cuadradoDeUnNumero(0)> 0)
```

```
    assert(cuadradoDeUnNumero(0)> 0)
AssertionError
```


¿Dudas?