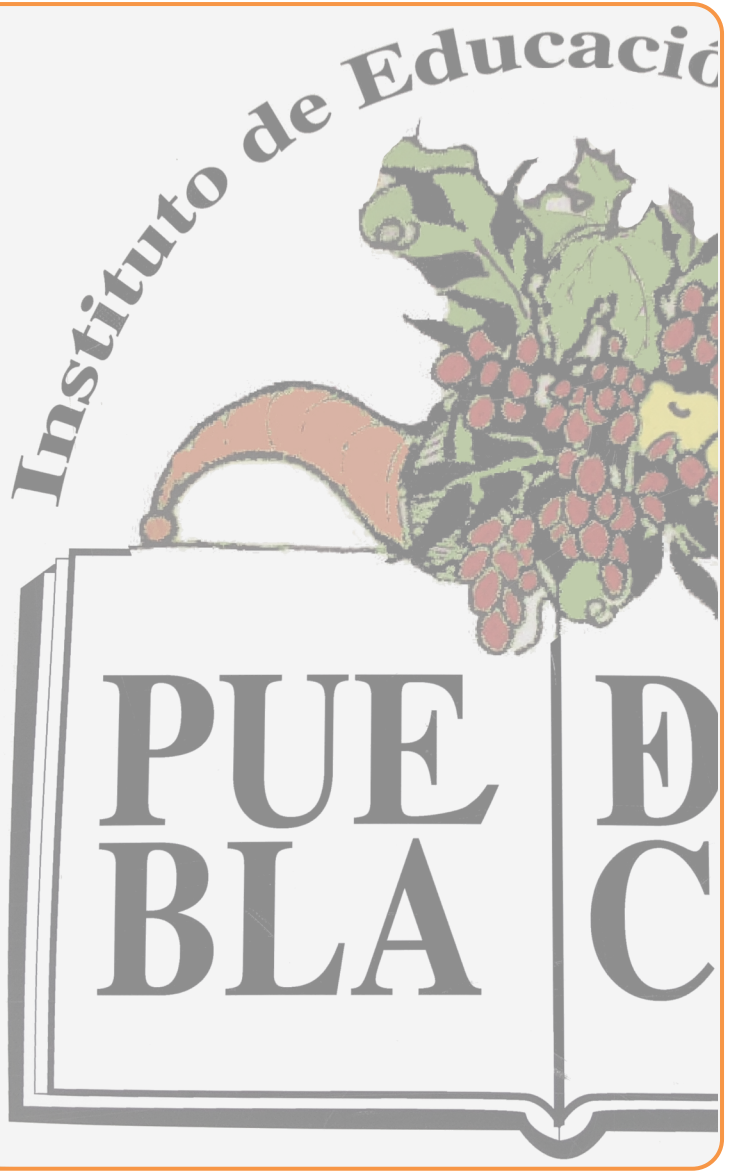
Funciones y excepciones

Programación de Inteligencia Artificial

Junta de Andalucía

FPA
FORMACIÓN PROFESIONAL ANDALUZA

Fondo Social Europeo



Índice | Funciones | Excepciones

Funciones:

- Introducción.
- Definición.
- Parámetros.
- Tipos.

Excepciones:

- try-except.
- else/finally.
- raise.

- Nos ayudan a organizar el código y dividirlo en subproblemas más sencillos (divide y vencerás).
- Nos permite reutilizar código de una forma clara y sencilla creándonos módulos que podemos importar en nuestros proyectos.
- Lo ideal es que las instrucciones que tiene una función no sean muy largas. Esto ayuda al mantenimiento del código.
- Son objetos de la clase *function*, por lo que nos permite crearnos variables que referencien una función.

- El formato de una función es: *def nombre(parámetros):*

```
def nombre_funcion(parametros):
    # Instrucciones de la función.
```

- Se usa la palabra reservada *def* para “definir” una función.
- *nombre* es el nombre que recibe la función y que identifica el proceso que va a realizar.
- Los *parámetros* son las variables que reciben los datos con los que va a operar la función. Podemos tener funciones sin parámetros o con más de 1. En este último caso van separados por comas.
- Si queremos que la función devuelva un dato usamos la palabra reservada *return*. Se pueden devolver varios datos. Lo que devolverá es una tupla.
- Se puede añadir un comentario con ”””” que servirá de ayuda si se usa *help*.

```
def suma(numero1, numero2):
    print("La suma de %d y %d es %d" %(numero1, numero2, numero1 + numero2))

def suma(numero1, numero2):
    """ Función que suma dos números y devuelve el resultado"""
    resultado = numero1 + numero2
    return resultado
```

- Nos permite recibir información en una función.
- Los parámetros de una función en Python se pasan por referencia. Por tanto, hay que tener cuidado cuando se pasa un tipo de dato mutable ya que las modificaciones realizadas en la función se verá reflejado en el programa principal (es recomendable pasar una copia).
- Parámetro formal: nombre que recibe la variable de la función en la que podemos recibir información. Pueden tomar valores por defecto en el caso de que no reciban ninguno.

```
def nombre_funcion(p_formal, p_valor_defecto = 0):
def suma(n1 = 0, n2 = 0):
```

- Parámetro real: los datos que se mandan a la función. También reciben el nombre de parámetro posicional cuando coincide la posición del parámetro real con el formal.

```
numero = 7
suma(numero, 800)
```

- Parámetro clave (*keyword*): cuando se llama a la función se indica el nombre del parámetro real en el que se va a almacenar el dato.

```
suma(n2 = 800, n1 = numero)
```

- Si añadimos un *"*"* los parámetros que hay después de él se tienen que referenciar mediante keyword.

```
def suma(n1, n2, *, n3 = 0, n4, n5):
    return n1 + n2 + n3 + n4 + n5

print(suma(5, 4, n5 = 3, n4 = 9))
```

- Parámetros extensibles: permite enviar un número variable de datos mediante una n-tupla. Para ello se pone un * seguido del nombre de la tupla (normalmente se usa **args*). Estos parámetros son posicionales.

```
def suma(n1, *args):
    resultado = n1
    for n in args: resultado += n
    return resultado

print(suma(4, 5, 1))
```

- Para guardar un número variable de *keywords* se usan 2 asteriscos seguidos del nombre (***kwargs*). De esta forma tendremos acceso a un diccionario de datos.

```
def suma(**kwargs):
    resultado = 0
    for n in kwargs.values():
        resultado += n
    return resultado
```

```
print(suma(n1 = 5, n2 = 7, n3 = 8))
numeros = {"n1": 7, "n2": 10, "n3": 10,
           "n4": 100}
print(suma(**numeros))
```

- Cuando se realiza la llamada de una función es posible pasar una serie de datos no nombrados. Esta puede ser mediante una secuencia con un * o un diccionario si se usan dos *.

```
def suma(*args, **kwargs):
    return sum(args) + sum(kwargs.values())

print(suma(*[2, 3, 5], **{"n1": 5, "n2": 5}))
```

- La última definición que hemos hecho de la función *suma* recibe el nombre de *función universal*, ya que acepta todo tipo de datos.
- Si se añade "/" al final de la lista de parámetros, obliga a que estos sean posicionales.

```
def suma (n1, n2 = 0, n3 = 0, /):
    return n1 + n2 + n3

print(suma(5, n3 = 4)) # Esta llamada da error
print(suma(5, 0, 4))
```

- Anotaciones: permite indicar el tipo de datos esperado, así como el dato que devuelve. Hay que aclarar que solo es una *anotación* para los desarrolladores.

```
def suma(n1:int, n2:int)->int:
    return n1 + n2
```

- Recursivas: es una función que se llama así misma y su código se divide en 2: un caso base donde finalizaría la llamada así misma, y la parte recursiva, que se va llamando así misma y modificando el dato que pasa por parámetro para que este tienda al caso base.

```
def factorial(numero):
    # Caso base
    if numero == 0 or numero == 1:
        return 1
    # Parte recursiva
    else:
        return numero * factorial(numero - 1)
```

- Generadoras: permiten devolver un dato de una secuencia cada vez que sea llamada. Para esto usa la palabra reservada *yield* en vez de *return*.

```
def del_1_al_10():
    for valor in range(1,11):
        yield valor

gen = del_1_al_10()
print(next(gen))
```

```
def del_1_al_3():
    yield 1
    yield 2
    yield 3

gen = del_1_al_3()
print(next(gen))
print(next(gen))
print(next(gen))
```

- Lambda: o funciones anónimas, se usan cuando necesitamos realizar una operación simple o un conjunto de instrucciones sencillas.

```
# Ejemplo 1
cuadrado = lambda x: x ** 2
print(cuadrado(7))

# Ejemplo 2
es_par = lambda x: True if x % 2 == 0 else False
print(es_par(7))

# Ejemplo 3
numeros = [(1, "uno"), (2, "dos"), (3, "tres")]
print(numeros)

# Ordenamos la lista por el segundo valor de las tuplas
numeros.sort(key = lambda valor: valor[1])
print(numeros)
```

- Decoradoras: son funciones que reciben como parámetro una función y devuelven otra. Esto se puede realizar debido a que se puede asignar una función a una variable. (enlace a la explicación).

```
def funcion_a(funcion_b):
    def funcion_c():
        print('Antes de la ejecución de la función a decorar')
        funcion_b()
        print('Después de la ejecución de la función a decorar')

    return funcion_c
```

Cuando *decoramos* una función lo que queremos es modificar el comportamiento de ésta (queremos añadir nuevas funcionalidades).

```
def tablas(funcion):
    def envoltura(tabla=1):
        print('Tabla del %i:' %tabla)
        print('-' * 15)
        for numero in range(0, 11):
            funcion(numero, tabla)
            print('-' * 15)
    return envoltura
```

```
@tablas
def suma(numero, tabla=1):
    print('%2i + %2i = %3i'
          %(tabla, numero, tabla+numero))
@tablas
def multiplicar(numero, tabla=1):
    print('%2i X %2i = %3i'
          %(tabla, numero, tabla*numero))

Suma(5); multiplicar(5)
```

Ejemplo del curso Python 3 en OW

Excepciones

try-except

else/finally

raise

- Son errores que se dan durante la ejecución del programa.
- Para manejar una excepción se crea un bloque *try*, que es dónde se puede dar el error y uno o varios bloques *except* que nos permiten capturar diferentes tipos de excepciones.

```
def dividir(n1, n2):
    try:
        resultado = n1 / n2
        print(resultado)
    except TypeError:
        print("Error. Uno de los datos introducidos no es un número")
    except ZeroDivisionError:
        print("Error. El divisor es 0")
    except:
        print("Error desconocido")
```

- Se pueden referenciar las excepciones mediante una variable y obtener más información.

```
def dividir(n1, n2):
    try:
        resultado = n1 / n2
        print(resultado)
    except (TypeError, ZeroDivisionError) as error:
        print("Error: ", error)
```

Excepciones

- Se puede añadir la sentencia *else* a la excepciones. El código que contenga se ejecutará si no se da ningún error.

```
def dividir(n1, n2):
    try:
        resultado = n1 / n2
    except (TypeError, ZeroDivisionError) as error:
        print("Error: ", error)
    else:
        print(resultado)
```

- También se puede añadir un bloque *finally*. Se ejecutarán las instrucciones que contenga tanto si se produce un fallo como si no hay ninguno. Este bloque suele tener instrucciones de finalización como puede ser cerrar el flujo de datos con un archivo o una BD.

Excepciones

try-except

else/finally

raise

- Podemos crear excepciones manuales mediante *raise*. Por ejemplo, si solo quisiéramos realizar la división de números positivos, tendríamos:

```
def dividir(n1, n2):
    try:
        if n1 < 0 or n2 < 0:
            raise ValueError("Los valores han de ser positivos")
        resultado = n1 / n2
    except (ValueError, TypeError, ZeroDivisionError) as error:
        print("Error: ", error)
    else:
        print(resultado)
```

Instituto de Educación Secundaria PUEBLA DE VICAR

FPA
FORMACIÓN PROFESIONAL ANDALUZA

Fondo Social Europeo

Imágenes: Flaticon.com, IES La Puebla, Junta de Andalucía