Python: funciones, clases y paquetes

Sistemas de gestión empresarial – 148fa (DAM)

Funciones

Una función es un grupo de sentencias que realizan una tarea concreta. Esta forma de agrupar código es una forma de ordenar nuestra aplicación en pequeños bloques, facilitando así su lectura y permitiendo reutilizar el código que contienen sin esfuerzo..

- **Definir y llamar a una función**: son variables que guardan datos de diferentes tipos, pero que sí que pueden ser modificados.

```
1  def saludo(nombre):
2  # codigo de la función
3      print("Hola, " + nombre+ ". ¡Bienvenido!")
```

Se escribe la palabra reservada **def** seguida del nombre de la función y sus parámetros entreparéntesis. Para llamar a una función solo hay que escribir el nombre de la función seguida de los parámetros(si los hubiera) entre paréntesis.

```
1  >>> saludo('Maitane')
2  Hola, Maitane. ¡Bienvenida!
```

Funciones

- Es posible asignar al parámetro un valor por defecto:

```
1  def saludo(nombre = "Anónimo"):
2      print("Hola, " + nombre+ ". ¡Bienvenido!")
3
4  saludo("Leire") # Hola, Maitane. ¡Bienvenida!
5  saludo() # Hola, Anónimo. ¡Bienvenida!
```

Funciones

- Parámetros, existen dos tipos de parámetros o argumentos:
  - *Parámetros posicionales*: la posición en la que se pasan importa
  - *Parámetros con palabra clave* (keyword arguments): la posición no importa, se indica una clave para cada parámetro.
- Pueden devolver un valor:

```
1  def suma(a, b):
2          resultado = a + b
3          return resultado
4
5  print(suma(4,5)) # 9
```

Funciones con argumentos múltiples

Es posible recibir un número desconocido de parámetros añadiendo un * en la definición de la función:

```
1  def suma_todo(*args):
2      resultado = 0
3      for i in args:
4          resultado += i
5          return resultado
6  v, w, x, y, z = 5, 2, 12, 6, 9
7  total = suma_todo(v, w, x, y, z)
8  print("La suma total es:" + str(total))  # La suma total es: 34
```

Ámbitos de las variables

- El ámbito de una variable (scope) se refiere a la zona del programa dónde una variable “existe”.
- **Fuera** del ámbito de una variable **no** podremos acceder a su valor ni manejarla.
- **Ámbito local:** Los parámetros y variables definidos en una función no estarán accesibles fuera de la función.
    - Es importante mencionar que una vez ejecutada una función, el valor de las variables locales no se almacena, por lo que la próxima vez que se llame a la función, ésta no recordará ningún valor de llamadas anteriores.

Ámbitos de las variables

```
1  def calcula():
2      a = 1
3      print("Dentro de la función:", a)
4
5  a = 5
6  calcula()
7  print("Fuera de la función:", a)
8
9  ### Output ###
10 # Dentro de la función:1
11 # Fuera de la función:5
```

Ámbitos de las variables

- Por el contrario, las variables definidas fuera de una función sí que están accesibles desde dentro de la función. Se considera que están en el **ámbito global**.

- No obstante, no se podrán modificar dentro de la función a no ser que estén definidas con la palabra clave *global*.

Excepciones

- Las excepciones son errores en la ejecución de un programa que hacen que el programa termine de forma inesperada.
- Normalmente ocurren debido a un uso indebido de los datos (p.ej. una división entre cero).
- La manera de controlar las excepciones es agrupando el código en 2 bloques (más 1 opcional):

```
1  try:
2          numero = int(input('Introduce un número: '))
3          dividendo = 150
4          resultado = dividendo / numero
5          print(resultado)
6  except ValueError:
7          print('Número inválido')
8  except ZeroDivisionError:
9          print('No se puede dividir entre 0')
10 finally:
11         print("Ejecutando finally antes de salir")
```

Excepciones

- Hay algunas excepciones que son bastante comunes a la hora de programar en Python y que deberíamos contemplar en nuestros programas:
  - **TypeError**:

```
>>> '1'+1
Traceback (most recent call last):
File "<pyshell#23>", line 1, in <module>
'2'+2
TypeError: must be str, not int
```

  - **ValueError**:

```
>>> int('hola')
Traceback (most recent call last):
File "<pyshell#14>", line 1, in <module>
int('xyz')
ValueError: invalid literal for int() with base 10: 'hola'
```

  - **ModuleNotFound**: es lanzado cuando no se encuentra el módulo indicado.

Excepciones

- **NameError:** es lanzado cuando no utiliza un objeto que no existe.

```
>>> persona
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>
age
NameError: name 'persona' is not defined
```

- **IndexError:** es lanzado al intentar acceder a un índice que no existe en un array.

```
>>> lista = [1,2,3]
>>> lista[5]
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>
lista[5]
IndexError: list index out of range
```

- **KeyError:** es lanzado cuando no se encuentra la clave (key).

```
>>> diccionario={'1':"esto", '2':"es", '3':"python"}
>>> diccionario['4']
Traceback (most recent call last):
File "<pyshell#15>", line 1, in <module>
diccionario['4']
KeyError: '4'
```

Clases y objetos

- Python soporta la programación orientada a objetos. Esto quiere decir que podemos definir entidades agrupando (encapsulando) sus atributos y comportamiento (métodos) en clases:

```
1  class Persona:
2          # atributos
3          nombre = "Josune"
4          edad = 24
5
6          # metodos
7          def camina(self):
8                  print(self.nombre + " está caminando")
```

Clases y objetos

- Las clases contienen el método especial __**init**__ conocido como constructor y que sirve para inicializar un objeto.
- Al crear un objeto siempre se llama al constructor. Una diferencia importante con otros lenguajes como Java es que solo se puede definir **un único constructor**.

```
1  class Persona:
2          def __init__(self, nombre, apellidos, edad):
3                  self.nombre= nombre
4                  self.apellidos = apellidos
5                  self.edad = edad
6
7          def camina(self):
8                  print(self.nombre + " está caminando")
```

Atributos de instancia y de clase

- Los atributos definidos dentro del constructor se conocen como **atributos de instancia**, por lo tanto, los atributos definidos dentro de la clase pero fuera del constructor se conocen como **atributos de clase**.

- La principal diferencia es que un atributto de clase puede ser accedido aunque no existan instancias de la clase. Además, si se modifica su valor, se modificará el valor en todas las instancias existentes de dicha clase.

```
1  class Demo:
2          atrib_estatico = 123 # compartido por todos los objetos
3          def __init__(self,numero):
4                  self.atrib_instancia = numero # específico de cada objeto
5
6  c1 = Demo(456)
7  c2 = Demo(789)
8
9  # Valor inicial
10 print(f"C1: Estatico {1.atrib_estatico} - Instancia: {c1.atrib_instancia}")
11 # output: C1: Estatico 123 - Instancia: 456
12 print(f"C2: Estatico {c2.atrib_estatico} - Instancia: {c2.atrib_instancia}")
```

Atributos de instancia y de clase

- Es importante remarcar que para acceder a los atributos de instancia se debe utilizar la palabra reservada *self*, la cual hace referencia al objeto actual.

- En Python no podemos utilizar *self* en cualquier momento, para utilizarlo hay que indicarlo en los métodos cómo el primer parámetro recibido.

```
1  class Persona:
2          def __init__(self, nombre, apellidos, edad):
3                  self.nombre= nombre
4                  self.apellidos = apellidos
5                  self.edad = edad
6
7          def camina(self): # es necesario indicar 'self' como primer argumento
8                  print(self.nombre + " está caminando")
9
10 p1 = Persona("Mike", "Mendiola", 25) # no hay que pasarle 'self'
11 p1.camina() # no hay que pasarle 'self'
12 print(p1.nombre)
13 print(p1.edad)
```

Herencia

- La herencia es una técnica de la Programación Orientada a Objetos en la que una clase (conocida como clase hija o subclase) hereda todos los métodos y propiedades de otra clase (conocida como padre o clase base).
- La sintaxis para definir una clase que herede de otra es la siguiente:

```python
class Dispositivo:
        def __init__(self,identificador,marca):
                self.identificador = identificador
                self.marca = marca

        def conectar(self):
                print("¡Conectado!")

# la clase base se indica entre paréntesis
class Teclado(Dispositivo):
        def __init__(self,identificador,marca,tipo):
                # llamada al constructor del padre
                Dispositivo.__init__(self,identificador,marca)
                self.tipo = tipo
        # metodo de la subclase
        def pulsar_tecla(self,tecla):
                print(tecla)

t1 = Teclado("0001", "Logitech", "AZERTY")
print(f"Id: {t1.identificador}, Marca: {t1.marca}, tipo: {t1.tipo}")
t1.conectar()
t1.pulsar_tecla("a")
```

```python
# en caso de conflicto Dispositivo tendrá prioridad sobre Periférico
class Teclado(Dispositivo, Periférico):
        # cuerpo de la clase
```

Módulos

- Un módulo es un archivo de Python que contiene variables, funciones y clases. Es una forma de ordenar y reutilizar código ya que todo el contenido de un módulo es accesible por los archivos que lo importen.

```python
# mundo.py

def hola_mundo():
    print("¡Hola Mundo!")

def adios_mundo():
    print("¡Adios Mundo!")
```

```python
# app.py

import mundo

# Llamada a la función
mundo.hola_mundo()
```

```python
# app.py

from mundo import adios_mundo

# Llamada a la función
adios_mundo()
```

Paquetes

- Es posible agrupar los módulos que tienen relación en un mismo directorio. Estos directorios son conocidos en Python como paquetes y deben contener siempre un archivo llamado ***__init__.py*** para que Python lo reconozca como un paquete.

- A medida que desarrollamos una aplicación es habitual agrupar los archivos en directorios (paquetes) para tener el código organizado.

- Para cargar un módulo ubicado en un paquete lo haremos de la siguiente forma:

```
1  import mipaquete.mundo
```

```
1  from mipaquete import mundo
```

```
1  from mipaquete.mundo import adios_mundo, hola_mundo
```