Introducción a las funciones

Introducción a la programación estructurada y modular

La programación estructurada es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de ordenador, utilizando únicamente subrutinas (funciones o procedimientos) y tres estructuras: secuencia, alternativas y repetitivas.

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. Esto debe hacerse hasta obtener subproblemas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación (divide y vencerás).

La programación estructurada y modular se lleva a cabo en python3 con la definición de funciones.

Definición de funciones

Veamos un ejemplo de definición de función:

```
>>> def factorial(n):
...     """Calcula el factorial de un número"""
...     resultado = 1
...     for i in range(1,n+1):
...         resultado*=i
...     return resultado
```

Podemos obtener información de la función:

```
>>> help(factorial)
Help on function factorial in module __main__:

factorial(n)
    Calcula el factorial de un número
```

Y para utilizar la función:

```
>>> factorial(5)

120
```

Ámbito de variables. Sentencia global

Una variable local se declara en su ámbito de uso (en el programa principal y dentro de una función) y una global fuera de su ámbito para que se pueda utilizar en cualquier función que la declare como global.

```
>>> def operar(a,b):

...     global suma

...     suma = a + b

...     resta = a - b

...     print(suma,resta)

...

>>> operar(4,5)

9 -1

>>> resta

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

NameError: name 'resta' is not defined

>>> suma

9
```

Podemos definir variables globales, que serán visibles en todo el módulo. Se recomienda declararlas en mayúsculas:

```
>>> PI = 3.1415

>>> def area(radio):

...     return PI*radio**2

...

>>> area(2)
```

12.566

Parámetros formales y reales

- Parámetros formales: Son las variables que recibe la función, se crean al definir la función. Su contenido lo recibe al realizar la llamada a la función de los parámetros reales. Los parámetros formales son variables locales dentro de la función.

- Parámetros reales: Son expresiones que se utilizan en la llamada de la función, sus valores se copian en los parámetros formales.

Paso de parámetro por valor o por referencia

En Python el paso de parámetros es siempre por referencia. El lenguaje no trabaja con el concepto de variables sino con objetos y referencias. Al realizar la asignación a = 1 no se dice que "a contiene el valor 1" sino que "a referencia a 1". Así, en comparación con otros lenguajes, podría decirse que en Python los parámetros siempre se pasan por referencia.

Evidentemente si se pasa un valor de un objeto inmutable, su valor no se podrá cambiar dentro de la función:

```
>>> def f(a):
...     a=5
>>> a=1
>>> f(a)
>>> a
1
```

Sin embargo si pasamos un objeto de un tipo mutable, si podremos cambiar su valor:

```
>>> def f(lista):
...   lista.append(5)
...
>>> l = [1,2]
>>> f(l)
>>> l
[1, 2, 5]
```

Aunque podemos cambiar el parámetro real cuando los objetos pasados son de tipo mutables, no es recomendable hacerlo en Python. En otros lenguajes es necesario porque no tenemos opción de devolver múltiples valores, pero como veremos en Python podemos devolver tuplas o lista con la instrucción return.

Llamadas a una función

Cuando se llama a una función se tienen que indicar los parámetros reales que se van a pasar. La llamada a una función se puede considerar una expresión cuyo valor y tipo es el retornado por la función. Si la función no tiene una instrucción return el tipo de la llamada será None.

```
>>> def cuadrado(n):
...   return n*n


>>> a=cuadrado(2)
>>> cuadrado(3)+1
10
>>> cuadrado(cuadrado(4))
256
>>> type(cuadrado(2))
<class 'int'>
```

Cuando estamos definiendo una función estamos creando un objeto de tipo function.

```
>>> type(cuadrado)
<class 'function'>
```

Y por lo tanto puedo guardar el objeto función en otra variable:

```
>>> c=cuadrado
>>> c(4)
16
```

Conceptos avanzados sobre funciones

Tipos de argumentos: posicionales o keyword

Tenemos dos tipos de parámetros: los posicionales donde el parámetro real debe coincidir en posición con el parámetro formal:

```
>>> def sumar(n1,n2):
...   return n1+n2
...
>>> sumar(5,7)
12
>>> sumar(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sumar() missing 1 required positional argument: 'n2'
```

Además podemos tener parámetros con valores por defecto:

```
>>> def operar(n1,n2,operador='+',respuesta='El resultado es '):
...   if operador=="+":
...     return respuesta+str(n1+n2)
...   elif operador=="-":
...     return respuesta+str(n1-n2)
...   else:
...     return "Error"
...
>>> operar(5,7)
'El resultado es 12'
>>> operar(5,7,"-")
'El resultado es -2'
>>> operar(5,7,"-","La resta es ")
```

'La resta es -2'

Los parámetros keyword son aquellos donde se indican el nombre del parámetro formal y su valor, por lo tanto no es necesario que tengan la misma posición. Al definir una función o al llamarla, hay que indicar primero los argumentos posicionales y a continuación los argumentos con valor por defecto (keyword).

```
>>> operar(5,7)          # dos parámetros posicionales

>>> operar(n1=4,n2=6)         # dos parámetros keyword

>>> operar(4,6,respuesta="La suma es")      # dos parámetros posicionales y uno keyword

>>> operar(4,6,respuesta="La resta es",operador="-")        # dos parámetros posicionales y dos keyword
```

Parámetro *

Un parámetro * entre los parámetros formales de una función, nos obliga a indicar los parámetros reales posteriores como keyword:

```
>>> def sumar(n1,n2,*,op="+"):

...   if op=="+":

...     return n1+n2

...   elif op=="-":

...     return n1-n2

...   else:

...     return "error"

...

>>> sumar(2,3)

5

>>> sumar(2,3,"-")

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: sumar() takes 2 positional arguments but 3 were given

>>> sumar(2,3,op="-")
```

-1

Argumentos arbitrarios (*args y **kwargs)

Para indicar un número indefinido de argumentos posicionales al definir una función, utilizamos el símbolo *:

```
>>> def sumar(n,*args):
...   resultado=n
...   for i in args:
...     resultado+=i
...   return resultado
...
>>> sumar(2)
2
>>> sumar(2,3,4)
9
```

Para indicar un número indefinido de argumentos keyword al definir una función, utilizamos el símbolo **:

```
>>> def saludar(nombre="pepe",**kwargs):
...   cadena=nombre
...   for valor in kwargs.values():
...     cadena=cadena+" "+valor
...   return "Hola "+cadena
...
>>> saludar()
'Hola pepe'
>>> saludar("juan")
'Hola juan'
>>> saludar(nombre="juan",nombre2="pepe")
```

'Hola juan pepe'

>>> saludar(nombre="juan",nombre2="pepe",nombre3="maria")

'Hola juan maria pepe'

Por lo tanto podríamos tener definiciones de funciones del tipo:

>>> def f()

>>> def f(a,b=1)

>>> def f(a,*args,b=1)

>>> def f(*args,b=1)

>>> def f(*args,b=1,*kwargs)

>>> def f(*args,*kwargs)

>>> def f(*args)

>>> def f(*kwargs)

Desempaquetar argumentos: pasar listas y diccionarios

En caso contrario es cuando tenemos que pasar parámetros que tenemos guardados en una lista o en un diccionario.

Para pasar listas utilizamos el símbolo *:

>>> lista=[1,2,3]

>>> sumar(*lista)

6

>>> sumar(2,*lista)

8

>>> sumar(2,3,*lista)

11

Podemos tener parámetros keyword guardados en un diccionario, para enviar un diccionario utilizamos el símbolo **:

```
>>> datos={"nombre":"jose","nombre2":"pepe","nombre3":"maria"}

>>> saludar(**datos)

'Hola jose maria pepe'
```

Devolver múltiples resultados

La instrucción return puede devolver cualquier tipo de resultados, por lo tanto es fácil devolver múltiples datos guardados en una lista o en un diccionario. Veamos un ejemplo en que devolvemos los datos en una tupla:

```
>>> def operar(n1,n2):

...   return (n1+n2,n1-n2,n1*n2)


>>> suma,resta,producto = operar(5,2)

>>> suma

7

>>> resta

3

>>> producto

10
```

Tipos especiales de funciones

Funciones recursivas

Una función recursiva es aquella que al ejecutarse hace llamadas a ella misma. Por lo tanto tenemos que tener "un caso base" que hace terminar el bucle de llamadas. Veamos un ejemplo:

```
>>> def factorial(numero):

...     if(numero == 0 or numero == 1):

...         return 1

...     else:
```

```
...         return numero * factorial(numero-1)

...

>>> factorial(5)

120
```

Funciones lambda

Las funciones lambda nos sirven para crear pequeñas funciones anónimas, de una sola línea sobre la marcha.

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

>>> cuadrado(2)
```

Como podemos notar las funciones lambda no tienen nombre. Pero gracias a que lambda crea una referencia a un objeto función, la podemos llamar.

```
>>> lambda x: x**2

<function <lambda> at 0xb74469cc>

>>>

>>> (lambda x: x**2)(3)

9
```

Otro ejemplo:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]

>>> pairs.sort(key=lambda pair: pair[1])

>>> pairs

[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Decoradores

Los decoradores son funciones que reciben como parámetros otras funciones y retornan como resultado otras funciones con el objetivo de alterar el funcionamiento original de la función que se pasa como parámetro. Hay funciones que tienen en común muchas funcionalidades, por ejemplo las de manejo de errores de conexión de recursos I/O (que se

deben programar siempre que usemos estos recursos) o las de validación de permisos en las respuestas de peticiones de servidores, en vez de repetir el código de rutinas podemos abstraer, bien sea el manejo de error o la respuesta de peticiones, en una función decorador.

```
>>> 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))


# Muestra la tabla de sumar del 1
suma()
# Muestra la tabla de sumar del 4
suma(4)
# Muestra la tabla de multiplicar del 1
multiplicar()
# Muestra la tabla de multiplicar del 10
multiplicar(10)
```

Funciones generadoras

Un generador es un tipo concreto de iterador. Es una función que permite obtener sus resultados paso a paso.

```
>>> def par(inicio,fin):
...   for i in range(inicio,fin):
...     if i % 2==0:
...       yield i


>>> datos = par(1,5)
>>> next(datos)
2
>>> next(datos)
4


>>> for i in par(20,30):
...   print(i,end=" ")
20 22 24 26 28


>>> lista_pares = list(par(1,10))
>>> lista_pares
[2, 4, 6, 8]
```

Ejercicios con funciones

1. Escribir dos funciones que permitan calcular:

- La cantidad de segundos en un tiempo dado en horas, minutos y segundos.
- La cantidad de horas, minutos y segundos de un tiempo dado en segundos.

2. Realiza una función que dependiendo de los parámetros que reciba: convierte a segundos o a horas:

- Si recibe un argumento, supone que son segundos y convierte a horas, minutos y segundos.
- Si recibe 3 argumentos, supone que son hora, minutos y segundos y los convierte a segundos.

3. Queremos hacer una función que añade a una lista los contactos de una agenda. Los contactos se van a guardar en un diccionario, y al menos debe tener el campo de nombre, el campo del teléfono, aunque puede tener más campos. Los datos se irán pidiendo por teclado, se pedirá de antemanos cuántos contactos se van a guardar. Si vamos a guardar más información en el contacto, se irán pidiendo introduciendo campos hasta que introducimos el "*".

4. Amplía el programa anterior para hacer una función de búsqueda, que reciba un conjunto de parámetros keyword y devuelve los contactos (en una lista) que coincidan con los criterios de búsqueda.

5. Realizar una función recursiva que reciba una lista y que calcule el producto de los elementos de la lista:

Programación orientada a objetos

Introducción a la Programación Orientada a Objetos

La Programación Orientado a Objetos (POO) se basa en la agrupación de objetos de distintas clases que interactúan entre sí y que, en conjunto, consiguen que un programa cumpla su propósito. En Python cualquier elemento del lenguaje pertenece a una clase y todas las clases tienen el mismo rango y se utilizan del mismo modo.

Definición de clase, objeto, atributos y métodos

- Llamamos clase a la representación abstracta de un concepto. Por ejemplo, "perro", "número entero" o "servidor web".
- Las clases se componen de atributos y métodos.
- Un objeto es cada una de las instancias de una clase.
- Los atributos definen las características propias del objeto y modifican su estado. Son datos asociados a las clases y a los objetos creados a partir de ellas.
- Un atributo de clase es compartida por todas las instancias de una clase. Se definen dentro de la clase (después del encabezado de la clase) pero nunca dentro de un método. Este tipo de variables no se utilizan con tanta frecuencia como las variables de instancia.
- Un atributo de objeto se define dentro de un método y pertenece a un objeto determinado de la clase instanciada.
- Los métodos son bloques de código (o funciones) de una clase que se utilizan para definir el comportamiento de los objetos.

Definimos nuestra primera clase:

```
>>> class clase():
...    at_clase=1
...    def metodo(self):
...      self.at_objeto=1
...
>>> type(clase)
<class 'type'>
>>> clase.at_clase
1
>>> objeto=clase()
>>> objeto.at_clase
1
>>> objeto.at_objeto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'clase' object has no attribute 'at_objeto'
>>> objeto.metodo()
>>> objeto.at_objeto
1
```

Atributos de objetos

Para definir atributos de objetos, basta con definir una variable dentro de los métodos, es una buena idea definir todos los atributos de nuestras instancias en el constructor, de modo que se creen con algún valor válido.

Método constructor init

Como hemos visto anteriormente los atributos de objetos no se crean hasta que no hemos ejecutado el método. Tenemos un método especial, llamado constructor __init__, que nos permite inicializar los atributos de objetos. Este método se llama cada vez que se crea una nueva instancia de la clase.

Definiendo métodos. El parámetro self

El método constructor, al igual que todos los métodos de cualquier clase, recibe como primer parámetro a la instancia sobre la que está trabajando. Por convención a ese primer parámetro se lo suele llamar self (que podríamos traducir como yo mismo), pero puede llamarse de cualquier forma.

Para referirse a los atributos de objetos hay que hacerlo a partir del objeto self.

Definición de objetos

Vamos a crear una nueva clase:

```
import math

class punto():

""" Representación de un punto en el plano, los atributos son x e y

que representan los valores de las coordenadas cartesianas."""


def __init__(self,x=0,y=0):

        self.x=x

        self.y=y


def distancia(self, otro):

        """ Devuelve la distancia entre ambos puntos. """

        dx = self.x - otro.x

        dy = self.y - otro.y

        return math.sqrt((dx*dx + dy*dy))
```

Para crear un objeto, utilizamos el nombre de la clase enviando como parámetro los valores que va a recibir el constructor.

```
>>> punto1=punto()

>>> punto2=punto(4,5)

>>> print(punto1.distancia(punto2))

6.4031242374328485
```

Podemos acceder y modificar los atributos de objeto:

```
>>> punto2.x

4

>>> punto2.x = 7

>>> punto2.x

7
```

Conceptos avanzados de programación orientada a objetos I

Atributos de clase (estáticas)

En Python, las variables definidas dentro de una clase, pero no dentro de un método, son llamadas variables estáticas o de clase. Estas variables son compartidas por todos los objetos de la clase.

Para acceder a una variable de clase podemos hacerlo escribiendo el nombre de la clase o a través de self.

```
>>> class Alumno():

...    contador=0

...    def __init__(self,nombre=""):

...       self.nombre=nombre

...       Alumno.contador+=1

...

>>> a1=Alumno("jose")

>>> a1.contador

1

>>> Alumno.contador

1
```

Usamos las variables estáticas (o de clase) para los atributos que son comunes a todos los atributos de la clase. Los atributos de los objetos se definen en el constructor.

Atributos privados y ocultos

Las variables que comienzan por un guión bajo _ son consideradas privadas. Su nombre indica a otros programadores que no son públicas: son un detalle de implementación del que no se puede depender — entre otras cosas porque podrían desaparecer en cualquier momento. **Pero nada nos impide acceder a esas variables.**

```
>>> class Alumno():
...     def __init__(self,nombre=""):
...         self.nombre=nombre
...         self._secreto="asdasd"
...
>>> a1=Alumno("jose")
>>> a1.nombre
'jose'
>>> a1._secreto
'asdasd'
```

Dos guiones bajos al comienzo del nombre __ llevan el ocultamiento un paso más allá, "enmaráñando" (name-mangling) la variable de forma que sea más difícil verla desde fuera.

```
>>> class Alumno():
...     def __init__(self,nombre=""):
...         self.nombre=nombre
...         self.__secreto="asdasd"
...
>>> a1=Alumno("jose")
>>> a1.__secreto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Alumno' object has no attribute '__secreto'
```

Pero en realidad sigue siendo posible acceder a la variable.

```
>>> a1._Alumno__secreto

'asdasd'
```

Se suelen utilizar cuando una subclase define un atributo con el mismo nombre que la clase madre, para que no coincidan los nombres.

Métodos de clase (estáticos)

Los métodos estáticos (static methods) son aquellos que no necesitan acceso a ningún atributo de ningún objeto en concreto de la clase.

```
>>> class Calculadora():
...   def __init__(self,nombre):
...     self.nombre=nombre
...   def modelo(self):
...     return self.nombre
...   @staticmethod
...   def sumar(x,y):
...     return x+y
...
>>> a=Calculadora("basica")
>>> a.modelo()
'basica'
>>> a.sumar(3,4)
7
>>> Calculadora.sumar(3,4)
7
```

Nada nos impediría mover este método a una función fuera de la clase, ya que no hace uso de ningún atributo de ningún objeto, pero la dejamos dentro porque su lógica (hacer sumas) pertenece conceptualmente a Calculadora.

Lo podemos llamar desde el objeto o desde la clase.

Funciones getattr,setattr,delattr,hasattr

```
>>> a1=Alumno("jose")

>>> getattr(a1,"nombre")

'jose'

>>> getattr(a1,"edad","no tiene")

'no tiene'


>>> setattr(a1,"nombre","pepe")

>>> a1.nombre

'pepe'


>>> hasattr(a1,"nombre")

True


>>> delattr(a1,"nombre")
```

Conceptos avanzados de programación orientada a objetos II

Propiedades: getters, setters, deleter

Para implementar la encapsulación y no permitir el acceso directo a los atributos, podemos poner los atributos ocultos y declarar métodos específicos para acceder y modificar los atributos (mutadores). Estos métodos se denominan getters y setters.

```
class circulo():

    def __init__(self,radio):

        self.set_radio(radio)

    def set_radio(self,radio):
```

```
            if radio>=0:
                self._radio = radio
            else:
                raise ValueError("Radio positivo")
                self._radio=0
    def get_radio(self):
            print("Estoy dando el radio")
            return self._radio


>>> c1=circulo(3)
>>> c1.get_radio()
Estoy dando el radio
3
>>> c1.set_radio(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jose/github/curso_python3/curso/u51/circulo.py", line 8, in set_radio
    raise ValueError("Radio positivo")
ValueError: Radio positivo
```

En Python, las propiedades nos permiten implementar la funcionalidad exponiendo estos métodos como atributos.

```
class circulo():
    def __init__(self,radio):
            self.radio=radio


    @property
    def radio(self):
            print("Estoy dando el radio")
```

```
            return self._radio

    @radio.setter
    def radio(self,radio):
            if radio>=0:
                    self._radio = radio
            else:
                    raise ValueError("Radio positivo")
                    self._radio=0
```

```
>>> c1=circulo(3)
>>> c1.radio
Estoy dando el radio
3
>>> c1.radio=4
>>> c1.radio=-1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
  File "/home/jose/github/curso_python3/curso/u52/circulo2.py", line 15, in radio
    raise ValueError("Radio positivo")
ValueError: Radio positivo
```

Hay un tercera property que podemos crear: el deleter

...

```
@radio.deleter
def radio(self):
        del self._radio
```

```
>>> c1=circulo(3)

>>> c1.radio

Estoy dando el radio

3

>>> del c1.radio

>>> c1.radio

Estoy dando el radio

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

  File "/home/jose/github/curso_python3/curso/u52/circulo2.py", line 8, in radio

    return self._radio

AttributeError: 'circulo' object has no attribute '_radio'

>>> c1.radio=3
```

Representación de objetos __str__ y __repr__

La documentación de Python hace referencia a que el método __str()__ ha de devolver la representación "informal" del objeto, mientras que __repr()__ la "formal".

- La función __str()__ debe devolver la cadena de texto que se muestra por pantalla si llamamos a la función str(). Esto es lo que hace Python cuando usamos print. Suele devolver el nombre de la clase.
- De __repr()__, por el otro lado, se espera que nos devuelva una cadena de texto con una representación única del objeto. Idealmente, la cadena devuelta por __repr()__ debería ser aquella que, pasada a eval(), devuelve el mismo objeto.

Continuamos con la clase circulo:

```
...

def __str__(self):

        clase = type(self).__name__

        msg = "{0} de radio {1}"

        return msg.format(clase, self.radio)
```

```
def __repr__(self):
        clase = type(self).__name__
        msg = "{0}({1})"
        return msg.format(clase, self.radio)
```

Suponemos que estamos utilizando la clase circulo sin la instrucción print en el getter.

```
>>> c1=circulo(3)
>>> print(c1)
circulo de radio 3
>>> repr(c1)
'circulo(3)'
>>> type(eval(repr(c1)))
<class 'circulo2.circulo'>
```

Comparación de objetos __eq__

Tampoco podemos comparar dos circulos sin definir __eq()__, ya que sin este método Python comparará posiciones en memoria.

Continuamos con la clase circulo:

```
...
def __eq__(self,otro):
        return self.radio==otro.radio
```

```
>>> c1=circulo(5)
>>> c2=circulo(3)
>>> c1 == c2
False
```

Si queremos utilizar <, <=, > y >= tendremos que rescribir los métodos: __lt()__, __le()__, __gt()__ y __ge()__

Operar con objetos __add__ y __sub__

Si queremos operar con los operadores + y -:

```
def __add__(self,otro):
        self.radio+=otro.radio


def __sub__(self,otro):
        if self.radio-otro.radio>=0:
                self.radio-=otro.radio
        else:
                raise ValueError("No se pueden restar")
```

```
>>> c1=circulo(5)
>>> c2=circulo(3)
>>> c1 + c2
>>> c1.radio
8
```

```
>>> c1=circulo(5)
>>> c2=circulo(3)
>>> c1 - c2
>>> c1.radio
2
>>> c1 - c2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "/home/jose/github/curso_python3/curso/u52/circulo2.py", line 42, in __sub__

    raise ValueError("No se pueden restar")

ValueError: No se pueden restar
```

Más métodos especiales

Existen muchos más métodos especiales que podemos sobreescribir en nuestras clases para añadir funcionalidad a las mismas. Puedes ver la documentación oficial para aprender más sobre ellas.

Polimorfismo, herencia y delegación

Polimorfismo

El polimorfismo es la técnica que nos posibilita que al invocar un determinado método de un objeto, podrán obtenerse distintos resultados según la clase del objeto. Esto se debe a que distintos objetos pueden tener un método con un mismo nombre, pero que realice distintas operaciones.

Lo llevamos usando desde principio del curso, por ejemplo podemos recorrer con una estructura for distintas clases de objeto, debido a que el método especial __iter__ está definido en cada una de las clases. Otro ejemplo sería que con la función print podemos imprimir distintas clases de objeto, en este caso, el método especial __str__ está definido en todas las clases.

Además esto es posible ya que python es dinámico, es decir en tiempo de ejecución es cuando se determina el tipo de un objeto. Veamos un ejemplo:

```
class gato():

        def hablar(self):

                print("MIAU")


class perro():

        def hablar(self):

                print("GUAU")
```

```
def escucharMascota(animal):
    animal.hablar()


if __name__ == '__main__':
    g = gato()
    p = perro()
    escucharMascota(g)
    escucharMascota(p)
```

Herencia

La herencia es un mecanismo de programación orientada a objetos que sirve para crear clases nuevas a partir de clases preexistentes. Se toman (heredan) atributos y métodos de las clases viejas y se los modifica para modelar una nueva situación.

La clase desde la que se hereda se llama clase base y la que se construye a partir de ella es una clase derivada.

Si nuestra clase base es la clase punto estudiadas en unidades anteriores, puedo crear una nueva clase de la siguiente manera:

```
class punto3d(punto):
    def __init__(self,x=0,y=0,z=0):
        super().__init__(x,y)
        self.z=z
    @property
    def z(self):
        return self._z


    @z.setter
    def z(self,z):
        self._z=z


    def __str__(self):
```

```
        return super().__str__()+":"+str(self.z)


    def distancia(self,otro):
        dx = self.x - otro.x
        dy = self.y - otro.y
        dz = self.z - otro.z
        return (dx*dx + dy*dy + dz*dz)**0.5
```

Creemos dos objetos de cada clase y veamos los atributos y métodos que tienen definido:

```
>>> p=punto(1,2)
>>> dir(p)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_x', '_y', 'distancia', 'x', 'y']
>>> p3d=punto3d(1,2,3)
>>> dir(p3d)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_x', '_y', '_z', 'distancia', 'x', 'y', 'z']
```

La función super()

La función super() me proporciona una referencia a la clase base. Y podemos observar también que hemos reescrito el método distancia y __str__.

```
>>> p.distancia(punto(5,6))
5.656854249492381
>>> p3d.distancia(punto3d(2,3,4))
1.7320508075688772
>>> print(p)
1:2
```

```
>>> print(p3d)

1:2:3
```

Herencia múltiple

La herencia múltiple se refiere a la posibilidad de crear una clase a partir de múltiples clases superiores. Es importante nombrar adecuadamente los atributos y los métodos en cada clase para no crear conflictos.

```
class Telefono:

    "Clase teléfono"

    def __init__(self,numero):

        self.numero=numero

    def telefonear(self):

        print('llamando')

    def colgar(self):

        print('colgando')

    def __str__(self):

        return self.numero


class Camara:

    "Clase camara fotográfica"

    def __init__(self,mpx):

        self.mpx=mpx

    def fotografiar(self):

        print('fotografiando')

    def __str__(self):

        return self.mpx

class Reproductor:

    "Clase Reproductor Mp3"

    def __init__(self,capcidad):
```

```
        self.capacidad=capcidad
    def reproducirmp3(self):
        print('reproduciendo mp3')
    def reproducirvideo(self):
        print('reproduciendo video')
    def __str__(self):
        return self.capacidad


class Movil(Telefono, Camara, Reproductor):
    def __init__(self,numero,mpx,capacidad):
        Telefono.__init__(self,numero)
        Camara.__init__(self,mpx)
        Reproductor.__init__(self,capacidad)
    def __str__(self):
                                        return "Número: {0}, Cámara: {1},Capacidad: {2}".format(Telefono.__str__(self),Camara.__str__(self),Reproductor.__str__(self))
```

Veamos los atributos y métodos de un objeto Movil:

```
>>> mimovil=Movil("645234567","5Mpx","1G")
>>> dir(mimovil)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'capacidad', 'colgar', 'fotografiar', 'mpx', 'numero', 'reproducirmp3', 'reproducirvideo', 'telefonear']
>>> print(mimovil)
Número: 645234567, Cámara: 5Mpx,Capacidad: 1G
```

Funciones issubclass() y isinstance()

La función issubclass(SubClase, ClaseSup) se utiliza para comprobar si una clase (SubClase) es hija de otra superior (ClaseSup), devolviendo True o False según sea el

caso.

```
>>> issubclass(Movil,Telefono)

True
```

La función booleana isinstance(Objeto, Clase) se utiliza para comprobar si un objeto pertenece a una clase o clase superior.

```
>>> isinstance(mimovil,Movil)

True
```

Delegación

Llamamos delegación a la situación en la que una clase contiene (como atributos) una o más instancias de otra clase, a las que delegará parte de sus funcionalidades.

A partir de la clase punto, podemos crear la clase circulo de esta forma:

```
class circulo():


    def __init__(self,centro,radio):

        self.centro=centro

        self.radio=radio


    def __str__(self):

        return "Centro:{0}-Radio:{1}".format(self.centro.__str__(),self.radio)
```

Y creamos un objeto circulo:

```
>>> c1=circulo(punto(2,3),5)

>>> print(c1)

Centro:2:3-Radio:5
```