Creado por:

Isabel Maniega

Contenido del temario (30 Horas)

1. REPASO DE CONCEPTOS SOBRE TIPOS Y POO
2. THREADS
3. PATRONES DE DISEÑO MÁS COMUNES EN PYTHON
4. ANÁLISIS DE DATOS CON LAS LIBRERÍAS: NUMPY Y PANDAS
5. DIFERENCIAS PYTHON Y JYTHON
6. FORMATEAR CÓDIGO JSON / XML CON PYTHON
7. SERVICIOS REST
8. ARQUITECTURA SOAP
9. MÓDULO LOGGING
10. LIBRERÍA DE SERIALIZACIÓN
11. LIBRERÍA CTYPES

Programación Orientada a Objetos (POO) - Repaso

Explicación general

- La clase es una plantilla
- CON ESA PLANTILLA...creamos **OBJETOS**
- UN OBJETO...ES la **instancia de esa clase**
- los **atributos** son las **variables**
- los **métodos** son las **funciones**

Ejemplo - Introduccion I

In [1]:
```
class Empleado:
    # funciones se les llama MÉTODOS
    # variables se les atributos
    def __init__(self, Id, Name, Age, Role):
        self.Id = Id
        self.Name = Name
        self.Age = Age
        self.Role = Role
# Instancia
# genero tantos clientes/empleado como quiera de esta forma
# empleado1 es el objeto1

empleado1 = Empleado(1, "Ana", 30, "ingeniera")
```

```
empleado2 = Empleado(2, "Pedro", 45, "arquitecto")
empleado3 = Empleado(3, "Andrea", 25, "abogada")
```

In [3]:
```
empleado1.Age
```

Out[3]: 30

In [4]:
```
empleado2.Age
```

Out[4]: 45

In [5]:
```
empleado3.Age
```

Out[5]: 25

In [6]:
```
empleado1.Role
```

Out[6]: 'ingeniera'

In [7]:
```
empleado3.Name
```

Out[7]: 'Andrea'

In [8]:
```
empleado2.Id
```

Out[8]: 2

Ejemplo - Introducción II

In [9]:
```
# creamos la clase
class Cliente:
    # creamos la primera clase para inicializar el atributo nombre:
    def inicializar(self, nombre):
        self.nombre = nombre

    # creamos el segundo método
    def imprimir(self):
        print("Nombre: ", self.nombre)
```

In [10]:
```
# bloque principal
# creamos una instancia de la clase Cliente
# (primer objeto)
cliente1 = Cliente()

# era: NombreClase.NombreFuncion
# entonces: NombreInstancia.NombreMétodo
cliente1.inicializar("María")
# el otro método
cliente1.imprimir()
```

```
Nombre:  María
```

In [11]:
```
# segundo objeto
cliente2 = Cliente()
cliente2.inicializar("Luis")
cliente2.imprimir()
```

```
Nombre:  Luis
```

Polimorfismo

La misma función puede estar en diferentes clases

- Ejemplo que ya le habíamos visto

In [12]:
```python
class Clase1:
    def funcion1():
        print("Estamos en Clase1 - funcion1")

class Clase2:
    def funcion1():
        print("Estamos en Clase2 - funcion1")

class Clase3:
    def funcion1():
        print("Estamos en Clase3 - funcion1")
```

In [13]:
```python
Clase1.funcion1()
```

```
Estamos en Clase1 - funcion1
```

In [14]:
```python
Clase2.funcion1()
```

```
Estamos en Clase2 - funcion1
```

In [15]:
```python
Clase3.funcion1()
```

```
Estamos en Clase3 - funcion1
```

En POO es lo mismo

In [16]:
```python
class Clase1:
    def metodo1(self):
        print("Estamos en: Clase1 - metodo1")

class Clase2:
    def metodo1(self):
        print("Estamos en: Clase2 - metodo1")

class Clase3:
    def metodo1(self):
        print("Estamos en: Clase3 - metodo1")
```

In [17]:
```python
clase_1 = Clase1()
clase_2 = Clase2()
clase_3 = Clase3()
```

In [18]:
```python
clase_1.metodo1()
```

```
Estamos en: Clase1 - metodo1
```

In [19]:
```python
clase_2.metodo1()
```

```
Estamos en: Clase2 - metodo1
```

In [20]:
```
clase_3.metodo1()
```

```
Estamos en: Clase3 - metodo1
```

Encapsulamiento

¿Qué es?

Sirve para proteger a las variables de modificaciones no deseadas.

VARIABLES PRIVADAS / MÉTODOS PRIVADOS

= SOLO ACCESIBLES DESDE LA CLASE DONDE SE ENCUENTREN

en java y c++ se consigue la encapsulación de otra manera en python se hace con:

- (__nombre)
- Doble barra baja antes de la variable (atributo)
- Doble barra baja antes de la función (método)

Ejemplo 1

In [21]:
```
class Clase1:
    def __metodo1(self):
        print("Estamos en método1")
class Clase2:
    def metodo2():
        print("estamos en metodo2")
```

In [24]:
```
# AttributeError: type object 'Clase1' has no attribute '__metodo1'
# Clase1.__metodo1()
```

In [25]:
```
# AttributeError: type object 'Clase1' has no attribute 'metodo1'
# Clase1.metodo1()
```

In [27]:
```
obj = Clase1()

obj._Clase1__metodo1()
```

```
Estamos en método1
```

In [28]:
```
Clase2.metodo2()
```

```
estamos en metodo2
```

In [30]:
```
class Clase1:
    def __metodo1():
        print("Estamos en método1")
    # __metodo1() # Si lo declaramos dentro de la clase podemos hacer al
```

```
class Clase2:
    def metodo2():
        print("estamos en metodo2")
```

Ejemplo 2

In [31]:
```
class Clase1:
    def __metodo1():
        print("Estamos en método1")
        print("llamada a: metodo2")
        Clase2.metodo2()

class Clase2:
    def metodo2():
        print("estamos en metodo2")
        print("llamada a: -metodo1-")
        Clase1._Clase1__metodo1()
```

In [ ]:
```
# Clase1.__metodo1()

# AttributeError: type object 'Clase1' has no attribute '__metodo1'
```

In [ ]:
```
# Clase1.metodo1()

# AttributeError: type object 'Clase1' has no attribute '__metodo1'
```

In [33]:
```
# Clase2.metodo2()

# AttributeError: type object 'Clase1' has no attribute '_Clase2__metodo1
```

Ejemplo 3

1ª Parte

In [34]:
```
class Fecha_albaran:
    def __init__(self, dia, mes, año):
        self.dia = dia
        self.mes = mes
        self.año = año
albaran1 = Fecha_albaran(15, 1, 2015)
```

In [35]:
```
albaran1.dia
```

Out[35]: 15

In [36]:
```
albaran1.mes
```

Out[36]: 1

In [37]:
```
albaran1.año
```

Out[37]: 2015

2ª Parte

In [38]:
```
class Fecha_albaran:
    def __init__(self, dia, mes, año):
        self.__dia = dia # atributo privado
        self.mes = mes
        self.año = año
albaran1 = Fecha_albaran(15, 1, 2015)
```

In [39]:
```
albaran1.mes
```

Out[39]: 1

In [40]:
```
albaran1.año
```

Out[40]: 2015

In [45]:
```
# albaran1.dia

# AttributeError: 'Fecha_albaran' object has no attribute 'dia'
```

In [44]:
```
# albaran1.__dia

# AttributeError: 'Fecha_albaran' object has no attribute 'dia'
```

3ª Parte

In [46]:
```
class Fecha_albaran:
    def __init__(self):
        self.__dia = 15 # atributo privado
        self.__mes = 4
        self.__año = 2015

    def setMes(self, mes):
        if mes > 0 and mes < 13:
            self.__mes = mes
        else:
            print("mes no valido - No está entre Enero y Diciembre")

    def getMes(self):
        return self.__mes

albaran1 = Fecha_albaran()
```

In [47]:
```
albaran1.setMes(45)
```

```
mes no valido - No está entre Enero y Diciembre
```

In [48]:
```
albaran1.getMes()
```

Out[48]: 4

In [49]:
```
albaran1.setMes(6)
```

In [50]:
```
albaran1.getMes()
```

Out[50]: 6

Ejemplo 4

In [51]:
```
class Clase:
    """
        Clase para diferenciar variables y métodos
        publicos y privados
    """
    # Variables de clase (atributos en este caso)
    x = 10 # Atributo ACCESIBLE
    __y = 20 # SOLO ES ACCESIBLE DENTRO DE LA CLASE

    # MÉTODO ACCESIBLE
    def metodoPublico(self):
        # SE PUEDE ACCEDER A SU INFORMACIÓN
        print("el método es público")

    # MÉTODO SÓLO ACCESIBLE EN LA CLASE
    def __metodoPrivado(self):
        print("Este método es privado")
```

In [52]:
```
# objeto
clase1 = Clase()
```

In [ ]:
```
# Acceso a elementos PÙBLICOS
```

In [53]:
```
clase1.x
```

Out[53]: 10

In [54]:
```
clase1.metodoPublico()
```

```
el método es público
```

In [ ]:
```
# Acceso a los elementos PRIVADOS
```

In [57]:
```
# Clase.y

# AttributeError: type object 'Clase' has no attribute 'y'
```

In [59]:
```
# Clase.__metodoPrivado()

# AttributeError: type object 'Clase' has no attribute '__metodoPrivado'
```

In [60]:
```
class Clase:
    # Variables de clase (atributos en este caso)
    x = 10 # Atributo ACCESIBLE
    __y = 20 # SOLO ES ACCESIBLE DENTRO DE LA CLASE

    # MÉTODO ACCESIBLE
```

```
    def metodoPublico(y):
        # SE PUEDE ACCEDER A SU INFORMACIÓN
        print("el método es público")
        print("el valor de y es ", y)


    # MÉTODO SÓLO ACCESIBLE EN LA CLASE
    def __metodoPrivado(x):
        print("Este método es privado")
        print("el valor de x es : ", x)

    __metodoPrivado(x)
    # metodoPublico(__y)
```

```
Este método es privado
el valor de x es :  10
```

In [61]:
```
class Clase:
    """
        Clase para diferenciar variables y métodos
        publicos y privados
    """
    # Variables de clase (atributos en este caso)
    x = 10 # Atributo ACCESIBLE
    __y = 20 # SOLO ES ACCESIBLE DENTRO DE LA CLASE

    # MÉTODO ACCESIBLE
    def metodoPublico(y):
        # SE PUEDE ACCEDER A SU INFORMACIÓN
        print("el método es público")
        print("el valor de y es ", y)


    # MÉTODO SÓLO ACCESIBLE EN LA CLASE
    def __metodoPrivado(x):
        print("Este método es privado")
        print("el valor de x es : ", x)

    # __metodoPrivado(x)
    metodoPublico(__y)
```

```
el método es público
el valor de y es  20
```

Herencia

Herencia

La **herencia** es una práctica común (en la programación de objetos) de pasar atributos y métodos de la superclase (definida y existente) a una clase recién creada, llamada subclase.

En otras palabras, la herencia es una forma de construir una nueva clase, no desde cero, sino utilizando un repertorio de rasgos ya definido. La nueva clase hereda (y esta es la clave) todo el equipamiento ya existente, pero puedes agregar algo nuevo si es necesario.

Gracias a eso, es posible construir clases más especializadas (más concretas) utilizando algunos conjuntos de reglas y comportamientos generales predefinidos.

In [ ]:
```
class Vehicle:
    pass


class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass
```

Podemos decir que:

- La clase Vehicle es la superclase para clases LandVehicle y TrackedVehicle.
- La clase LandVehicle es una subclase de Vehicle y la superclase de TrackedVehicle al mismo tiempo.
- La clase TrackedVehicle es una subclase tanto de Vehicle y LandVehicle.

El conocimiento anterior proviene de la lectura del código (en otras palabras, lo sabemos porque podemos verlo).

Python ofrece una función que es capaz de identificar una relación entre dos clases, y aunque su diagnóstico no es complejo, puede verificar si una clase particular es una subclase de cualquier otra clase.

issubclass(): La función devuelve True si ClassOne es una subclase de ClassTwo, y False de lo contrario.

Hay dos bucles anidados. Su propósito es verificar todos los pares de clases ordenadas posibles y que imprima los resultados de la verificación para determinar si el par coincide con la relación subclase-superclase:

In [62]:
```
class Vehicle:
    pass


class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass


for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:
    for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:
        print(issubclass(cls1, cls2), end="\t")
    print()
```

```
True    False   False
True    True    False
True    True    True
```

isinstance(): La función devuelve True si el objeto es una instancia de la clase, o False de lo contrario.

Hemos creado tres objetos, uno para cada una de las clases. Luego, usando dos bucles anidados, verificamos todos los pares posibles de clase de objeto para averiguar si los objetos son instancias de las clases.

In [63]:
```
class Vehicle:
    pass


class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass


my_vehicle = Vehicle()
my_land_vehicle = LandVehicle()
my_tracked_vehicle = TrackedVehicle()

for obj in [my_vehicle, my_land_vehicle, my_tracked_vehicle]:
    for cls in [Vehicle, LandVehicle, TrackedVehicle]:
        print(isinstance(obj, cls), end="\t")
    print()
```

```
True    False   False
True    True    False
True    True    True
```

is: El operador 'is' verifica si dos variables, en este caso (object_one y object_two) se refieren al mismo objeto.

No olvides que las variables no almacenan los objetos en sí, sino solo los identificadores que apuntan a la memoria interna de Python.

Asignar un valor de una variable de objeto a otra variable no copia el objeto, sino solo su identificador. Es por ello que un operador como is puede ser muy útil en ciertas circunstancias.

Echa un vistazo al código en el editor. Analicémoslo:

- Existe una clase muy simple equipada con un constructor simple, que crea una sola propiedad. La clase se usa para instanciar dos objetos. El primero se asigna a otra variable, y su propiedad val se incrementa en uno.
- Luego, el operador 'is' se aplica tres veces para verificar todos los pares de objetos posibles, y todos los valores de la propiedad val son mostrados en pantalla.
- La última parte del código lleva a cabo otro experimento. Después de tres tareas, ambas cadenas contienen los mismos textos, pero estos textos se almacenan en diferentes objetos.

In [65]:
```
class SampleClass:
    def __init__(self, val):
        self.val = val


object_1 = SampleClass(0) # val = 0
object_2 = SampleClass(2) # val = 2
object_3 = object_1 # val = 0 (para los dos 3, 1)
object_3.val += 1 # val = 0 + 1 = 1

print(object_1 is object_2)
print(object_2 is object_3)
print(object_3 is object_1)
print(object_1.val, object_2.val, object_3.val)

string_1 = "Mary tenía un "
string_2 = "Mary tenía un corderito"
string_1 += "corderito"

print(id(string_1))
print(id(string_2))

print(string_1 == string_2, string_1 is string_2)
```

```
False
False
True
1 2 1
128217595561616
128217605873696
True False
```

Los resultados prueban que object_1 y object_3 son en realidad los mismos objetos, mientras que string_1 y string_2 no lo son, a pesar de que su contenido sea el mismo.

In [66]:
```
class Super:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "Mi nombre es " + self.name + "."


class Sub(Super):
    def __init__(self, name):
        Super.__init__(self, name)


obj = Sub("Andy")

print(obj)
```

```
Mi nombre es Andy.
```

- Existe una clase llamada Super, que define su propio constructor utilizado para asignar la propiedad del objeto, llamada name.
- La clase también define el método __ str __(), lo que permite que la clase pueda presentar su identidad en forma de texto.

- La clase se usa luego como base para crear una subclase llamadaSub. La clase Sub define su propio constructor, que invoca el de la superclase. Toma nota de como lo hemos hecho: Super.__ init __(self, name).
- Hemos nombrado explícitamente la superclase y hemos apuntado al método para invocar a __ init __(), proporcionando todos los argumentos necesarios.
- Hemos instanciado un objeto de la clase Sub y lo hemos impreso.

In [67]:
```
class Super:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "Mi nombre es " + self.name + "."


class Sub(Super):
    def __init__(self, name):
        super().__init__(name)


obj = Sub("Andy")

print(obj)
```

```
Mi nombre es Andy.
```

La función super() accede a la superclase sin necesidad de conocer su nombre.

La función super() crea un contexto en el que no tiene que (además, no debe) pasar el argumento propio al método que se invoca; es por eso que es posible activar el constructor de la superclase utilizando solo un argumento.

Nota: puedes usar este mecanismo no solo para invocar al constructor de la superclase, pero también para obtener acceso a cualquiera de los recursos disponibles dentro de la superclase.

In [68]:
```
class Super:
    supVar = 1


class Sub(Super):
    subVar = 2


obj = Sub()

print(obj.subVar)
print(obj.supVar)
```

```
2
1
```

In [69]:
```
class ExampleClass:

    counter = 0

    def __init__(self, val = 1):
```

```
        self.__first = val
        ExampleClass.counter += 1

example_object_1 = ExampleClass() # __first = 1, counter = 0 + 1 = 1
example_object_2 = ExampleClass(2) # __first = 2, counter = 1 + 1 = 2
example_object_3 = ExampleClass(4) # __first = 4, counter = 2 + 1 = 3

print(example_object_1.__dict__, example_object_1.counter)
print(example_object_2.__dict__, example_object_2.counter)
print(example_object_3.__dict__, example_object_3.counter)

print('\n')
print(ExampleClass.__dict__)
```

```
{'_ExampleClass__first': 1} 3
{'_ExampleClass__first': 2} 3
{'_ExampleClass__first': 4} 3


{'__module__': '__main__', 'counter': 3, '__init__': <function ExampleClas
s.__init__ at 0x749cfc34b400>, '__dict__': <attribute '__dict__' of 'Examp
leClass' objects>, '__weakref__': <attribute '__weakref__' of 'ExampleClas
s' objects>, '__doc__': None}
```

La clase Super define una variable de clase llamada supVar, y la clase Sub define una variable llamada subVar. Ambas variables son visibles dentro del objeto de clase Sub.

In [70]:
```
class Super:
    def __init__(self):
        self.supVar = 11


class Sub(Super):
    def __init__(self):
        super().__init__()
        self.subVar = 12


obj = Sub()

print(obj.subVar)
print(obj.supVar)
```

```
12
11
```

El constructor de la clase Sub crea una variable de instancia llamada subVar, mientras que el constructor de Super hace lo mismo con una variable de nombre supVar. Al igual que el ejemplo anterior, ambas variables son accesibles desde el objeto de clase Sub.

Nota: La existencia de la variable supVar obviamente está condicionada por la invocación del constructor de la clase Super. Omitirlo daría como resultado la ausencia de la variable en el objeto creado (pruébalo tu mismo).

Herencia Multiple

La herencia múltiple ocurre cuando una clase tiene más de una superclase.

In [71]:
```
class SuperA:
    var_a = 10
    def fun_a(self):
        return 11


class SuperB:
    var_b = 20
    def fun_b(self):
        return 21


class Sub(SuperA, SuperB):
    pass


obj = Sub()

print(obj.var_a, obj.fun_a())
print(obj.var_b, obj.fun_b())
```

```
10 11
20 21
```

La clase Sub tiene dos superclases: SuperA y SuperB. Esto significa que la clase Sub hereda todos los bienes ofrecidos por ambas clases SuperA y SuperB.

In [72]:
```
class Level1:
    var = 100
    def fun(self):
        return 101


class Level2(Level1):
    var = 200
    def fun(self):
        return 201


class Level3(Level2):
    pass


obj = Level3()

print(obj.var, obj.fun())
```

```
200 201
```

Tanto la clase, Level1 y Level2 definen un método llamado fun() y una propiedad llamada var. ¿Significará esto el objeto de la claseLevel3 podrá acceder a dos copias de cada entidad? De ningún modo.

La entidad definida después (en el sentido de herencia) anula la misma entidad definida anteriormente.

Como puedes ver, la variable de clase var y el método fun() de la clase Level2 anula las entidades de los mismos nombres derivados de la clase Level1.

Esta característica se puede usar intencionalmente para modificar el comportamiento predeterminado de las clases (o definido previamente) cuando cualquiera de tus clases necesite actuar de manera diferente a su ancestro.

In [73]:
```
class Left:
    var = "L"
    var_left = "LL"
    def fun(self):
        return "Left"


class Right:
    var = "R"
    var_right = "RR"
    def fun(self):
        return "Right"


class Sub(Left, Right):
    pass


obj = Sub()

print(obj.var, obj.var_left, obj.var_right, obj.fun())
```

```
L LL RR Left
```

La clase Sub hereda todos los bienes de dos superclases Left y Right (estos nombres están destinados a ser significativos).

No hay duda de que la variable de clase var_right proviene de la clase Right, y var_left proviene de la clase Left respectivamente.

Esto es claro. Pero, ¿De donde proviene la variable var? ¿Es posible adivinarlo? El mismo problema se encuentra con el método fun() - ¿Será invocada desde Left o desde Right?

Esto prueba que ambos casos poco claros tienen una solución dentro de la clase Left.

Podemos decir que Python busca componentes de objetos en el siguiente orden:

- Dentro del objeto mismo.
- En sus superclases, de abajo hacia arriba.
- Si hay más de una clase en una ruta de herencia, Python las escanea de izquierda a derecha.

In [74]:
```
class Left:
    var = "L"
    var_left = "LL"
    def fun(self):
        return "Left"
```

```
class Right:
    var = "R"
    var_right = "RR"
    def fun(self):
        return "Right"


class Sub(Right, Left): # modificamos las posisciones de Right y left
    pass


obj = Sub()

print(obj.var, obj.var_left, obj.var_right, obj.fun())
```

```
R LL RR Right
```

DIR

In [76]:
```
import numpy as np
```

In [77]:
```
print(np.array(dir(clase1)))
```

```
['_Clase__metodoPrivado' '_Clase__y' '__class__' '__delattr__' '__dict__'
 '__dir__' '__doc__' '__eq__' '__format__' '__ge__' '__getattribute__'
 '__gt__' '__hash__' '__init__' '__init_subclass__' '__le__' '__lt__'
 '__module__' '__ne__' '__new__' '__reduce__' '__reduce_ex__' '__repr__'
 '__setattr__' '__sizeof__' '__str__' '__subclasshook__' '__weakref__'
 'metodoPublico' 'x']
```

In [78]:
```
clase1.__dir__()
```

Out[78]:
```
['__module__',
 '__doc__',
 'x',
 '_Clase__y',
 'metodoPublico',
 '_Clase__metodoPrivado',
 '__dict__',
 '__weakref__',
 '__new__',
 '__repr__',
 '__hash__',
 '__str__',
 '__getattribute__',
 '__setattr__',
 '__delattr__',
 '__lt__',
 '__le__',
 '__eq__',
 '__ne__',
 '__gt__',
 '__ge__',
 '__init__',
 '__reduce_ex__',
 '__reduce__',
 '__subclasshook__',
 '__init_subclass__',
 '__format__',
 '__sizeof__',
 '__dir__',
 '__class__']
```

In [79]:
```
dir(np)
```

Out[79]:
```
['ALLOW_THREADS',
 'BUFSIZE',
 'CLIP',
 'DataSource',
 'ERR_CALL',
 'ERR_DEFAULT',
 'ERR_IGNORE',
 'ERR_LOG',
 'ERR_PRINT',
 'ERR_RAISE',
 'ERR_WARN',
 'FLOATING_POINT_SUPPORT',
 'FPE_DIVIDEBYZERO',
 'FPE_INVALID',
 'FPE_OVERFLOW',
 'FPE_UNDERFLOW',
 'False_',
 'Inf',
 'Infinity',
 'MAXDIMS',
 'MAY_SHARE_BOUNDS',
 'MAY_SHARE_EXACT',
 'NAN',
 'NINF',
 'NZERO',
 'NaN',
 'PINF',
 'PZERO',
 'RAISE',
 'RankWarning',
 'SHIFT_DIVIDEBYZERO',
 'SHIFT_INVALID',
 'SHIFT_OVERFLOW',
 'SHIFT_UNDERFLOW',
 'ScalarType',
 'True_',
 'UFUNC_BUFSIZE_DEFAULT',
 'UFUNC_PYVALS_NAME',
 'WRAP',
 '_CopyMode',
 '_NoValue',
 '_UFUNC_API',
 '__NUMPY_SETUP__',
 '__all__',
 '__builtins__',
 '__cached__',
 '__config__',
 '__deprecated_attrs__',
 '__dir__',
 '__doc__',
 '__expired_functions__',
 '__file__',
 '__former_attrs__',
 '__future_scalars__',
 '__getattr__',
 '__git_version__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
```

```
'__spec__',
'__version__',
'_add_newdoc_ufunc',
'_builtins',
'_distributor_init',
'_financial_names',
'_get_promotion_state',
'_globals',
'_int_extended_msg',
'_mat',
'_no_nep50_warning',
'_pyinstaller_hooks_dir',
'_pytesttester',
'_set_promotion_state',
'_specific_msg',
'_typing',
'_using_numpy2_behavior',
'_utils',
'_version',
'abs',
'absolute',
'add',
'add_docstring',
'add_newdoc',
'add_newdoc_ufunc',
'all',
'allclose',
'alltrue',
'amax',
'amin',
'angle',
'any',
'append',
'apply_along_axis',
'apply_over_axes',
'arange',
'arccos',
'arccosh',
'arcsin',
'arcsinh',
'arctan',
'arctan2',
'arctanh',
'argmax',
'argmin',
'argpartition',
'argsort',
'argwhere',
'around',
'array',
'array2string',
'array_equal',
'array_equiv',
'array_repr',
'array_split',
'array_str',
'asanyarray',
'asarray',
'asarray_chkfinite',
'ascontiguousarray',
```

```
 'asfarray',
 'asfortranarray',
 'asmatrix',
 'atleast_1d',
 'atleast_2d',
 'atleast_3d',
 'average',
 'bartlett',
 'base_repr',
 'binary_repr',
 'bincount',
 'bitwise_and',
 'bitwise_not',
 'bitwise_or',
 'bitwise_xor',
 'blackman',
 'block',
 'bmat',
 'bool_',
 'broadcast',
 'broadcast_arrays',
 'broadcast_shapes',
 'broadcast_to',
 'busday_count',
 'busday_offset',
 'busdaycalendar',
 'byte',
 'byte_bounds',
 'bytes_',
 'c_',
 'can_cast',
 'cast',
 'cbrt',
 'cdouble',
 'ceil',
 'cfloat',
 'char',
 'character',
 'chararray',
 'choose',
 'clip',
 'clongdouble',
 'clongfloat',
 'column_stack',
 'common_type',
 'compare_chararrays',
 'compat',
 'complex128',
 'complex256',
 'complex64',
 'complex_',
 'complexfloating',
 'compress',
 'concatenate',
 'conj',
 'conjugate',
 'convolve',
 'copy',
 'copysign',
 'copyto',
```

```
 'corrcoef',
 'correlate',
 'cos',
 'cosh',
 'count_nonzero',
 'cov',
 'cross',
 'csingle',
 'ctypeslib',
 'cumprod',
 'cumproduct',
 'cumsum',
 'datetime64',
 'datetime_as_string',
 'datetime_data',
 'deg2rad',
 'degrees',
 'delete',
 'deprecate',
 'deprecate_with_doc',
 'diag',
 'diag_indices',
 'diag_indices_from',
 'diagflat',
 'diagonal',
 'diff',
 'digitize',
 'disp',
 'divide',
 'divmod',
 'dot',
 'double',
 'dsplit',
 'dstack',
 'dtype',
 'dtypes',
 'e',
 'ediff1d',
 'einsum',
 'einsum_path',
 'emath',
 'empty',
 'empty_like',
 'equal',
 'errstate',
 'euler_gamma',
 'exceptions',
 'exp',
 'exp2',
 'expand_dims',
 'expm1',
 'extract',
 'eye',
 'fabs',
 'fastCopyAndTranspose',
 'fft',
 'fill_diagonal',
 'find_common_type',
 'finfo',
 'fix',
```

```
 'flatiter',
 'flatnonzero',
 'flexible',
 'flip',
 'fliplr',
 'flipud',
 'float128',
 'float16',
 'float32',
 'float64',
 'float_',
 'float_power',
 'floating',
 'floor',
 'floor_divide',
 'fmax',
 'fmin',
 'fmod',
 'format_float_positional',
 'format_float_scientific',
 'format_parser',
 'frexp',
 'from_dlpack',
 'frombuffer',
 'fromfile',
 'fromfunction',
 'fromiter',
 'frompyfunc',
 'fromregex',
 'fromstring',
 'full',
 'full_like',
 'gcd',
 'generic',
 'genfromtxt',
 'geomspace',
 'get_array_wrap',
 'get_include',
 'get_printoptions',
 'getbufsize',
 'geterr',
 'geterrcall',
 'geterrobj',
 'gradient',
 'greater',
 'greater_equal',
 'half',
 'hamming',
 'hanning',
 'heaviside',
 'histogram',
 'histogram2d',
 'histogram_bin_edges',
 'histogramdd',
 'hsplit',
 'hstack',
 'hypot',
 'i0',
 'identity',
 'iinfo',
```

```
'imag',
'in1d',
'index_exp',
'indices',
'inexact',
'inf',
'info',
'infty',
'inner',
'insert',
'int16',
'int32',
'int64',
'int8',
'int_',
'intc',
'integer',
'interp',
'intersect1d',
'intp',
'invert',
'is_busday',
'isclose',
'iscomplex',
'iscomplexobj',
'isfinite',
'isfortran',
'isin',
'isinf',
'isnan',
'isnat',
'isneginf',
'isposinf',
'isreal',
'isrealobj',
'isscalar',
'issctype',
'issubclass_',
'issubdtype',
'issubsctype',
'iterable',
'ix_',
'kaiser',
'kernel_version',
'kron',
'lcm',
'ldexp',
'left_shift',
'less',
'less_equal',
'lexsort',
'lib',
'linalg',
'linspace',
'little_endian',
'load',
'loadtxt',
'log',
'log10',
'log1p',
```

```
'log2',
'logaddexp',
'logaddexp2',
'logical_and',
'logical_not',
'logical_or',
'logical_xor',
'logspace',
'longcomplex',
'longdouble',
'longfloat',
'longlong',
'lookfor',
'ma',
'mask_indices',
'mat',
'matmul',
'matrix',
'max',
'maximum',
'maximum_sctype',
'may_share_memory',
'mean',
'median',
'memmap',
'meshgrid',
'mgrid',
'min',
'min_scalar_type',
'minimum',
'mintypecode',
'mod',
'modf',
'moveaxis',
'msort',
'multiply',
'nan',
'nan_to_num',
'nanargmax',
'nanargmin',
'nancumprod',
'nancumsum',
'nanmax',
'nanmean',
'nanmedian',
'nanmin',
'nanpercentile',
'nanprod',
'nanquantile',
'nanstd',
'nansum',
'nanvar',
'nbytes',
'ndarray',
'ndenumerate',
'ndim',
'ndindex',
'nditer',
'negative',
'nested_iters',
```

```
 'newaxis',
 'nextafter',
 'nonzero',
 'not_equal',
 'numarray',
 'number',
 'obj2sctype',
 'object_',
 'ogrid',
 'oldnumeric',
 'ones',
 'ones_like',
 'outer',
 'packbits',
 'pad',
 'partition',
 'percentile',
 'pi',
 'piecewise',
 'place',
 'poly',
 'poly1d',
 'polyadd',
 'polyder',
 'polydiv',
 'polyfit',
 'polyint',
 'polymul',
 'polynomial',
 'polysub',
 'polyval',
 'positive',
 'power',
 'printoptions',
 'prod',
 'product',
 'promote_types',
 'ptp',
 'put',
 'put_along_axis',
 'putmask',
 'quantile',
 'r_',
 'rad2deg',
 'radians',
 'random',
 'ravel',
 'ravel_multi_index',
 'real',
 'real_if_close',
 'rec',
 'recarray',
 'recfromcsv',
 'recfromtxt',
 'reciprocal',
 'record',
 'remainder',
 'repeat',
 'require',
 'reshape',
```

```
 'resize',
 'result_type',
 'right_shift',
 'rint',
 'roll',
 'rollaxis',
 'roots',
 'rot90',
 'round',
 'round_',
 'row_stack',
 's_',
 'safe_eval',
 'save',
 'savetxt',
 'savez',
 'savez_compressed',
 'sctype2char',
 'sctypeDict',
 'sctypes',
 'searchsorted',
 'select',
 'set_numeric_ops',
 'set_printoptions',
 'set_string_function',
 'setbufsize',
 'setdiff1d',
 'seterr',
 'seterrcall',
 'seterrobj',
 'setxor1d',
 'shape',
 'shares_memory',
 'short',
 'show_config',
 'show_runtime',
 'sign',
 'signbit',
 'signedinteger',
 'sin',
 'sinc',
 'single',
 'singlecomplex',
 'sinh',
 'size',
 'sometrue',
 'sort',
 'sort_complex',
 'source',
 'spacing',
 'split',
 'sqrt',
 'square',
 'squeeze',
 'stack',
 'std',
 'str_',
 'string_',
 'subtract',
 'sum',
```

```
 'swapaxes',
 'take',
 'take_along_axis',
 'tan',
 'tanh',
 'tensordot',
 'test',
 'testing',
 'tile',
 'timedelta64',
 'trace',
 'tracemalloc_domain',
 'transpose',
 'trapz',
 'tri',
 'tril',
 'tril_indices',
 'tril_indices_from',
 'trim_zeros',
 'triu',
 'triu_indices',
 'triu_indices_from',
 'true_divide',
 'trunc',
 'typecodes',
 'typename',
 'ubyte',
 'ufunc',
 'uint',
 'uint16',
 'uint32',
 'uint64',
 'uint8',
 'uintc',
 'uintp',
 'ulonglong',
 'unicode_',
 'union1d',
 'unique',
 'unpackbits',
 'unravel_index',
 'unsignedinteger',
 'unwrap',
 'ushort',
 'vander',
 'var',
 'vdot',
 'vectorize',
 'version',
 'void',
 'vsplit',
 'vstack',
 'where',
 'who',
 'zeros',
 'zeros_like']
```

In [ ]:
```
# '_Clase__metodoPrivado'

# Veíamos que está ahi, pero no es accesible
```

```
# pero...

# SE PUEDE IMPRIMIR ASI:
```

In [81]:
```
clase1.x
```

Out[81]:
```
10
```

In [82]:
```
clase1._Clase__y
```

Out[82]:
```
20
```

In [83]:
```
clase1.metodoPublico()
```

```
el método es público
```

In [84]:
```
clase1._Clase__metodoPrivado()
```

```
Este método es privado
```

In [85]:
```
albaran1.__dict__
```

Out[85]:
```
{'_Fecha_albaran__dia': 15,
 '_Fecha_albaran__mes': 6,
 '_Fecha_albaran__año': 2015}
```

In [86]:
```
# visualizar el contenido del objeto
vars(albaran1)
```

Out[86]:
```
{'_Fecha_albaran__dia': 15,
 '_Fecha_albaran__mes': 6,
 '_Fecha_albaran__año': 2015}
```

In [87]:
```
Fecha_albaran.__dict__
```

Out[87]:
```
mappingproxy({'__module__': '__main__',
              '__init__': <function __main__.Fecha_albaran.__init__(sel
f)>,
              'setMes': <function __main__.Fecha_albaran.setMes(self, me
s)>,
              'getMes': <function __main__.Fecha_albaran.getMes(self)>,
              '__dict__': <attribute '__dict__' of 'Fecha_albaran' objec
ts>,
              '__weakref__': <attribute '__weakref__' of 'Fecha_albaran'
objects>,
              '__doc__': None})
```

Creado por:

Isabel Maniega