Sesión 3

Control de flujo avanzado

List Comprenhension

A estas alturas podemos reconocer que Python permite crear código descriptivo, permitiendo cierta sintáxis elegante que facilita la lectura del código. Una de las técnicas más distintivas de Python que cumple esta premisa es las *list comprenhension*, **que permite crear listas, diccionarios o sests en base a un iterable existente**.

Esta sintaxis es muy popular ya que permite realizar operaciones complejas de forma compacta.

Dependiendo de la sintaxis podemos crear:

- List Comprenhension -> `[expr for x in interable]`
- Dictionary Comprenhension -> `{key_expr: value_expr for x in interable}`
- Set Comprenhension -> `(expr for x in iterable)`

```
# Create a list to be used in comprehensions
numbers = [1, 2, 3, -3, -2, -1]

# Not comprehension
list_new = []
for number in numbers:
  list_new.append(number * number)
print(list_new)

# Create a new list of these numbers' squares
new_list_comprehension = [number*number for number in numbers]
print(new_list_comprehension)

# Create a new dictionary of these numbers' exponentiation
# pow(2, 4) => 2^4
print({x: pow(10, x) for x in numbers})

# Create a set of these numbers' absolutes
print([abs(x) for x in numbers])
```

```
[1, 4, 9, 9, 4, 1]
[1, 4, 9, 9, 4, 1]
{1: 10, 2: 100, 3: 1000, -3: 0.001, -2: 0.01, -1: 0.1}
[1, 2, 3, 3, 2, 1]
```

También es posible añadir condicionales para filtrar el resultado:

```
# Create a list to be used in comprehensions
numbers = [1, 2, 3, -3, -2, -1]

# Create a new list of these numbers' squares
print([x*x for x in numbers if x > 0])

# Create a new dictionary of these numbers' exponentiation
print({x: pow(10, x) for x in numbers if x % 2 == 0})

# Create a set of these numbers' absolutes
print({abs(x) for x in numbers if x < -1})
```

Con esto podremos recrear de forma eficiente la función lambda que hemos declarado en la sección siguiente

```
# Filtrar numberos pares
li_2 = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
li_filter_2 = [x for x in li_2 if x%2 != 0]
print(li_filter_2)

li_filter_square_2 = [x*x for x in li_2 if x%2 != 0]
print(li_filter_square_2)
```

```
[5, 7, 97, 77, 23, 73, 61]
[25, 49, 9409, 5929, 529, 5329, 3721]
```

Iteradores y Generadores

Antes que nada vamos a definir qué son los **iteradores**. Para ello hay que definir 3 conceptos:

- **Iterable ->** Objeto en Python que tiene métodos `__iter__` y `__getitem__` que devuelven un **iterator** o puede coger índices. Básicamente es un objeto que nos aporta un **iterator**
- **Iterator ->** Objeto en python que tiene un método `__next__` definido.
- **Iteration ->** El proceso de coger un elemento de una lista. Es el proceso de recorrer la lista y coger los elementos.

```
# Iterator
my_list = [2, 4, 6, 1]

my_iter = iter(my_list)

print(next(my_iter))

print(my_iter.__next__())
```

```
2
4
```

Es posible crear iteradores en Python, solo hay que crear una clase con los métodos __iter__ y __next__

```
class OddIterator:
  """Iterator to return all odd numbers"""

  def __iter__(self):
    self.num = 1
    return self

  def __next__(self):
    num = self.num
    self.num += 2
    return num


oddIt = iter(OddIterator())

print(next(oddIt))
print(next(oddIt))
print(next(oddIt))
print(next(oddIt))
print(next(oddIt))
```

Generadores

Los generadores son iteradores, pero solo se puede iterar una vez. Es simplemente porque no guarda los elementos en memoria, se generarn al vuelo.

La mayoría de veces los **generadores** se implementan como **funciones**, además no hacen `return` de un elemento si no que realizan `yield` .

Al igual que las **funciones lambda** generan **funciones anónimas**, los generadores crean **funciones anónimas generadoras**

¿Por qué usar generadores en Python? Hay varios motivos:

1. Son sencillos de implementar
2. Eficientes con la gestión de memorias

3. Pueden representar una iteración infinita
4. Se pueden concatenar (pipeline)

```
# Generator
def generator_function():
    for i in range(10):
        yield i

for item in generator_function():
    print(item)
```

```
# Generator creating the power of two

def PowTwoGen(max=0):
  n = 0
  while n < max:
    yield 2 ** n
    n += 1

for num in PowTwoGen(10):
  print(num)
```

```
# Pipeline geneartor

def fibonacci_numbers(nums):
  x, y = 0, 1
  for _ in range(nums):
    x, y = y, x + y
    yield x

def square(nums):
  for num in nums:
    yield num**2

print(sum(square(fibonacci_numbers(10))))
```

Los generadores son muy útiles en **grandes datasets** donde hay una gran cantidad de elementos y penaliza mantenerlos en memoria.

Operador ternario

El operador ternario condicional es una forma concisa de realizar asignaciones condicionales. Se utiliza para asignar un valor a una variable basado en alguna condición. Es una forma más corta y legible de usar una instrucción if-else.

La sintaxis básica del operador ternario condicional es la siguiente:

```
 x = valor_si_verdadero if condicion else valor_si_falso
```

```
# Check largest number

a = 10
b = 20
largest = a if a > b else b
print(f"The largest number is {largest}")

# Check even or odd

number = 13
result = "Even" if number % 2 == 0 else "Odd"
print(f"The number {number} is {result}")

if number % 2 == 0:
  print(f"The number {number} is Even")
else:
  print(f"The number {number} is Odd")
```

```
The largest number is 20
The number 13 is Odd
The number 13 is Odd
```

Funciones Avanzadas

Vamos a entrar en un terreno más complicado y menos intuitivo. Aunque la sintaxis es sencilla una vez dominas el concepto puede que sea difícil entender la intencionalidad de las funciones lambda y anónimas.

Hay que incidir en que este concepto se usa mucho en *Data Science* y hay muchos modelos de tensorflow que toman funciones anónimas como argumentos de otras para ejecutar una serie de pasos.

Por ello es importante conocer al menos un poco de su funcionamiento y su finalidad

Funciones con argumentos de longitud variable

En algunas funciones encontraréis los parámetros *args y **kwargs. Estas son palabras reservadas que al principio es difícil de comprender. Lo primero de todo es que el nombre args y kwargs no es necesario, solo los asteriscos, pero por convenio se usan esas palabras, se podría tener *var y **vars pero no suele ser muy usual.

Estas palabras se utilizan para pasar una cantidad variable de argumentos a una función sin saber con exactitud la cantidad exacta. En el caso de *args podemos ver su función en esta función:

```
def test_var_args(f_arg, *argv):
    print(f"first normal arg: {f_arg}")
    print(argv)
    for arg in argv:
        print(f"another arg through *argv : {arg}")

test_var_args('lucas','python','ML','test', '1', "2", "3")
```

```
first normal arg: lucas
('python', 'ML', 'test', '1', '2', '3')
another arg through *argv : python
another arg through *argv : ML
another arg through *argv : test
another arg through *argv : 1
another arg through *argv : 2
another arg through *argv : 3
```

```
def multiply(*argv):
    z = 1
    for num in argv:
        z *= num
    print(z)

multiply(4, 5)
multiply(10, 9)
multiply(2, 3, 4)
multiply(3, 5, 10, 6)
multiply(3, 5, 10, 6, 34, 23, 34)
```

```
20
90
24
900
23929200
```

```
def sequential_model(*layers):
  print("-----------")
  for layer in layers:
    print(layer)
  print("-----------")

sequential_model("Dense", "Flatten", "Dense")
sequential_model("Dense_relu")
sequential_model("Dense_relu", "Dense")
```

```
-----------
Dense
Flatten
Dense
```

```
_________________________________________________________________
_________________________________________________________________
Dense_relu
_________________________________________________________________
_________________________________________________________________
Dense_relu
Dense
_________________________________________________________________
```

Por otra parte **kwargs se usa para pasar parámetros de longitud variable con clave, esto puede usarse con múltiples fines, como por ejemplo para ejecutar instrucciones como una CLI.

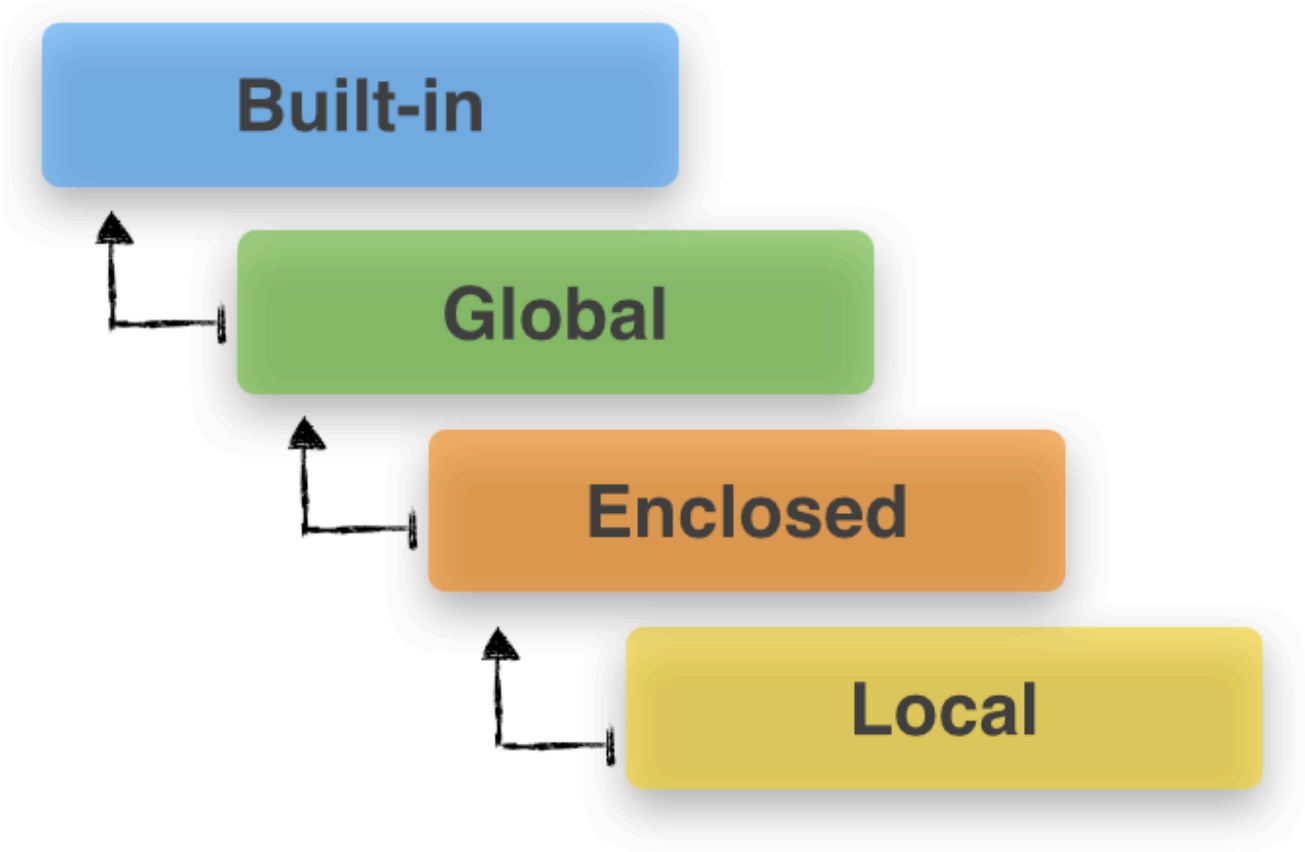
```
def greet_me(**kwargs):
    if kwargs is not None:
        print(kwargs)
        for key, value in kwargs.items():
            print(f"{key} --> {value}")

greet_me(name="lucas", language="python", area="ML", context="test", country="spa
print("=============")
greet_me(name="lucas", language="python", area="ML", context="test", country="spa
```

```
{'name': 'lucas', 'language': 'python', 'area': 'ML', 'context': 'test', 'cou
name --> lucas
language --> python
area --> ML
context --> test
country --> spain
asdfasdf --> asdfasdfa
=============
{'name': 'lucas', 'language': 'python', 'area': 'ML', 'context': 'test', 'cou
name --> lucas
language --> python
area --> ML
context --> test
country --> spain
month --> June
```

Variables globales vs Variable locales

Variables globales vs. Variables Locales

Todas las variables tienen un ámbito que puede depender de donde ha sido declarada una variable, así pueden existir variables globales, locales y contenidas. Esto se puede ver bien en el gráfico:



Sacando un ejemplo práctico podemos ver:

```
a_var = 'global value'
print(a_var)

def outer():
    a_var = 'local value'
    print('outer before:', a_var)
    def inner():
        nonlocal a_var
        a_var = 'inner value'
        print('in inner():', a_var)
    inner()
    print("outer after:", a_var)
outer()
print(a_var)
```

```
global value
outer before: local value
in inner(): inner value
outer after: inner value
global value
```

Funciones anónimas (Lambda)

Las funciones anónimas se llaman así ya que no se declaran con la palabra reservada **def**. No necesitan identación para declarar la funcionalidad y tienen una serie de características que las hacen muy especiales:

- Utilizan la palabra reservada **lambda** para declararse
- Pueden tener múltiples argumentos pero solo pueden devolver un valor por expresión
- Una función anónima no puede llamarse directamente, ya que **lambda** necesita una expresión.
- Las funciones lambda tienen su propio *namespace* y no pueden acceder variables otras que las pasadas por argumentos.

La sintaxis de una función lambda sería la siguiente:

```
lambda [arg1, arg2....]:expresion
```

Para comprender como funciona una función lambda primero vamos a ver una función sencilla:

```
def cube(x):
  return x*x*x

res = cube(7)
print(res)
```

```
343
```

Ahora vamos a representar esta misma función mediante **lambda**:

```
cub = lambda x: x*x*x
res = cub(7)
print(res)
```

```
343
```

Como podemos observar la diferencia principal es que en una función estándar necesitamos declarar un nombre de función al que tendremos que referenciar cada vez que queramos ejecutar una función. Con **lambda** tendremos una función anónima que podremos asignar a una variable para ejecutarla posteriormente.

Ahora vamos a ver la verdadera potencia de las funciones anónimas, cuando usamos métodos de listas como **filter o map** usaremos **lambda** para poder declarar las transformaciones de una forma muy elegante. Así, si queremos un programa que elimine los números impares de una lista pasaremos de esto:

```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
def filter_list(lis):
  new_list = []
  for element in lis:
    if(element%2 != 0):
      new_list.append(element)
  return new_list

li_filtered = filter_list(li)
print(li_filtered)
```

```
[5, 7, 97, 77, 23, 73, 61]
```

A una solución mucho más sencilla de leer:

```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
li_filtered = list(filter(lambda x: (x%2 != 0), li)) # return lazy iterator conve
print(li_filtered)
```

Esta serie de operaciones, que pueden involucrar transformación, reducción o filtrado de matrices son esenciales en *Data Science* además, introducen una nueva dimensionalidad de **sintactic sugar** que permite al programador hacer más escribiendo menos.

Para el uso de esta función lambda hemos recurrido a la función **filter**. Esta función permite transformar iterables en base a una función que pasaremos como argumento. Las funciones más comunes dentro del tratamiento de iterables son:

- **Map** -> Transforma todos los elementos de un iterable.
- **Filter** -> Filtra los elementos de un iterable en base a una condición.
- **Reduce** -> Reduce los elemntos de un iterable y devuelve un solo valor en base a una operación.

```
li = [5, 6, 22, 97]

# map
li_mapped = list(map(lambda x: x*2, li))
print(li_mapped)

# filter
li_filtered = list(filter(lambda x: (x%2 != 0), li)) # return lazy iterator conve
print(li_filtered)

# reduce
from functools import reduce
li_reduced = reduce((lambda x, y: x * y), li)
print(li_reduced)
```

```
[10, 12, 44, 194]
[5, 97]
64020
```

Python Orientado a Objetos

Entramos en uno de los conceptos más importantes dentro de la programación, el paradigma de la orientación a objetos. Como su nombre indica centra el desarrollo en una estructura llamada objetos.

Los conceptos de Programación Orientada a Objetos se remontan a principios de los 60, concretamente el lenguaje Simula 67 fue el primero en incorporarlo, aunque no fue hasta los años 90 que se empezó a popularizar.

Hasta ahora hemos estado viendo programación estructurada, que es aquella en que los datos y los procedimientos están separados y sin relación y animan a centrar el desarrollo en procedimientos y funciones. El problema puede ser que se acabe con mucho código repetido o también llamado código espagueti.

La **Programación Orientada a Objetos** se centra en el desarrollo de objetos. Un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases o sus propias clases, ya que puede tener diferentes valores en los **atributos**. Además tiene mecanísmos de interacción llamados **métodos** que favorece el cambio de estado o las acciones de los objetos.

```
wheels = 4
engine = 110
max_acc = 140
acc_seg = 80

def seg_to_mph(km):
  if km > 140:
    print("this car cannot")
    return
  return km/80

print(seg_to_mph(130))
```

```
1.625
```

```
wheels = 2
engine = 90
max_acc = 100
acc_seg = 40

def seg_to_mph(km):
  if km > 140:
    print("this car cannot")
    return
  return km/80

print(seg_to_mph(130))
```

Simplificando lo anterior, podemos decir que en la Orientación a Objetos estructuramos **variables** y **funciones** en unidades lógicas, lo que llamaremos objetos. Estas **variables** son lo que hemos llamado **atributos** y las **funciones** serán los **métodos**.

Un ejemplo de objeto sería un coche. Un coche tiene **atributos** importantes como el color, el modelo, el año y el tipo de gasolina. Además podemos interactuar con el objeto mediante sus **métodos** como arrancar, o acelerar.

Aunque sea sorprendente, ya hemos usado múltiples objetos en Python, por ejemplo las listas. Las listas tienen **métodos** como append(), clear() o copy() y se organizan en unidades lógicas con una función.

```
test = [4, 5, 6, 8]
test.clear()
print(test)
```

```
[]
```

La orientación a objetos está sustentada por cuatro pilares fundamentales:

- **Encapsulación** --> Agrupación de varios métodos y atributos en un objeto.
- **Abstracción** --> Capacidad de ocultar la complejidad de una implementación.
- **Herencia** --> Mecanismo para eliminar información redundante pudiendo compartir características.
- **Polimorfismo** --> Literalmente significa muchas formas, capacidad de que cambie el comportamiento de un objeto dependiendo del tipo de dato de entrada.

Sabiendo estos conceptos básicos podemos meternos de lleno en los diferentes elementos:

Clases

Son "prototipos" definidos por el programador que sirven como modelo para los atributos y los métodos de un objeto. Se podría hacer una analogía con las recetas de un plato. Con las clases podemos definir qué valores tendrá el objeto y qué podrá hacer.

Para crear una clase solo necesitamos declarar la palabra reservada *class* y definir la clase interna:

```
class NewClass(object):
    # Documentation
    class_body
```

Así podremos ver un ejemplo de clase completo con un modelo de ML:

```
class Model(object):
  'Common class for a ML Model'
  number_layers = 0

  def __init__(self, layers):
    self.layers = layers

  def get_layers(self):
    return self.layers

  def print_layers(self):
    for layer in self.layers:
      print(f"Layer {layer}")

  def compile(self, loss='mean_squared_error',
              optimizer='Adam'):
    print("Compiling..")

  def fit(self, features, labels, epoch=500, verbose=False):
    if verbose:
      print("verbose mode....")
    print("Training...")
```

Aquí podemos ver tres elementos importantes:

- La variable *number_layers* es una variable compartida por todas las instancias de la clase, puede acceder estáticamente a través de una instancia *ClassName.variable*, en este caso *Model.number_layers*
- El primer método *__init__()* es un método especial reservado a la inicialización del objeto, llamado **constructor**. Este es llamado cuando se instancia un objeto.
- Se declaran los atributos y los métodos, que son proipos de cada instancia. Para ello se usa la palabra reservada *self* tanto como primer argumento de un método como para referenciar a un atributo.

Objetos

Un objeto es una instancia de una estructura de datos definida en una clase. Un objeto se compone tanto de **atributos** de clase como de métodos.

Para crear una instancia de una clase, solo hay que asignar a una variable la llamada al constructor de una clase y pasar los argumentos que acepta el constructor *__init__*

En Python la instanciación de un objeto sería de la siguiente forma:

```
inst_object = NewClass(attribute)
```

```
dense_model = Model(["l0"])

dense_model_sequential = Model(["l0", "l1", "l2"])

classification_model = Model(["lDense", "lFlatten"])
```

Atributos

Una vez instanciado el objeto, podremos acceder a sus atributos y métodos, para ello usaremos la notación de python (que ya conocemos bastante bien) con la llamada a la variable, seguida de un punto y el nombre del atribtuo o método.

```
isnt_object.attribute
inst_object.method()
```

Así vemos como acceder a los atributos de nuestros objetos ya instanciados:

```
dense_model_sequential.print_layers()
print(dense_model_sequential.layers)
dense_model_sequential.compile()
features= [1, 3, 4, 5, 6, 8]
layers= [2, 8, 14, 25, 36, 18]
dense_model_sequential.fit(features, layers)
```

```
Layer l0
Layer l1
Layer l2
['l0', 'l1', 'l2']
Compiling..
Training...
```

En las llamadas a métodos, pese a que tenga como primer argumento *self* no es necesario declararlo, ya que como comentamos es una palabra reservada de Python para declarar una función como un método de clase.

Por otro lado, podremos modificar atributos de clase normalmente:

```
dense_model_sequential.layers = ["lDense"]
dense_model_sequential.print_layers()
```

```
Layer lDense
```

Hay una serie de funciones de Python que permiten consultar ciertas características de los atributos en Python:

- **getattr(obj, name)** -> Accede al atributo del objeto
- **hasattr(obj, name)** -> Comprueba si el atributo existe
- **setattr(obj, name)** -> Añade un atributo dinámicamente a un objeto
- **delattr(obj, name)** -> Elimina dinámicamente un atributo

```
print(hasattr(dense_model, 'layers'))    # returns true if layer exists
print(getattr(dense_model_sequential, 'layers'))    # returns value of 'layer' at
setattr(dense_model, 'layers', ["lDense"]) # Set attribute 'layer' at lDense
dense_model.print_layers()
delattr(dense_model, 'layers')    # deletes attribute 'color'
print(hasattr(dense_model, 'layers'))
```

```
True
['lDense']
Layer lDense
False
```

Por

```
print(f"Model.__doc__: {Model.__doc__}")
print(f"Model.__name__: {Model.__name__}")
print(f"Model.__module__: {Model.__module__}")
print(f"Model.__bases__: {Model.__bases__}")
print(f"Model.__dict__: {Model.__dict__}")
print(f"Model.__str__: {Model.__str__}")
```

Herencia

Para evitar la repetición de código de clases que puedan ser similares o compartan características en común, podemos derivar una clase de otra, haciendo que compartan tanto

atributos como *métodos*.

La clase hija "hereda" toda esto y lo puede utilizar según su conveniencia, adaptando valores para el propósito de la clase.

La sintaxis de la herencia en Python es la siguiente:

```
class ParentClass(object):
    object definition
    def __init__(self):
        constructor...

class ChildClass(ParentClass):
     objedct definition
     def __init__(self):
        super().__init__() #optional
```

Vamos a adelantarnos un poco a la lección, pero en Tensorflow utilizaremos la herencia para declarar nuestros propios modelos:

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Flatten, Conv2D
from tensorflow.keras import Model

class MyModel(Model):
  def __init__(self):
    super(MyModel, self).__init__()
    self.conv1 = Conv2D(32, 3, activation='relu')
    self.flatten = Flatten()
    self.d1 = Dense(128, activation='relu')
    self.d2 = Dense(10)

  def call(self, x):
    x = self.conv1(x)
    x = self.flatten(x)
    x = self.d1(x)
    return self.d2(x)



# Create an instance of the model
model = MyModel()
model.call()
# model.compile(...)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-34-a259c40fc704> in <cell line: 3>()
      1 # Create an instance of the model
      2 model = MyModel()
----> 3 custom_model.compile(optimizer='adam',
      4               loss='sparse_categorical_crossentropy',
      5               metrics=['accuracy'])

NameError: name 'custom_model' is not defined
```

Decoradores

En Python tenemos un elemento de sintáxis muy poderoso llamado **decorador**. Básicamente son funciones o clases que envuelven otras funciones o clases y las modifican.

Es muy útil para **separar efectos** y reutilizar lógica entre aplicaciones.

Básicamente aprovecha varias funcionalidades de Python para poder modificar su ejecución.

```
def is_weekend(day):
  print("What a week, huh?")

  def is_weekend():
    print("Sure it was!")

  def is_not_weekend():
    print(f"Captain, is {day}")

  if day in ["sunday", "saturday"]:
    return is_weekend()
  else:
    return is_not_weekend()

is_weekend("wednesday")

print("==============")

def week_decorator(func):
  def wrapper():
    print("What a week, huh?")
    func()
    print("....")

  return wrapper

def reply():
  print("Captain, is wednesday")

dec = week_decorator(reply)
dec()

print("==============")

@week_decorator
def reply_dec():
  print("Captain, is friday")

reply_dec()

print("==============")
```

```
# Funciones dentro de funciones
def is_weekend(weekend):
  print("What a week, huh?")

  def is_weekend():
    print("Yeah, sure it was")

  def is_not_weekend():
    print("Captain, is Wednesday")

  if weekend:
    return is_weekend()
  else:
    return is_not_weekend()

is_weekend(False)


# Ejemplo de decorador
def my_decorator(func):
  def wrapper():
    print("First is called this")
    func()
    print("Finally we call this")
  return wrapper

def reply():
  print("Captain, is Wednesday")

decor = my_decorator(reply)

decor()
```

```
First is called this
Captain, is Wednesday
Finally we call this
```

```
# Decorator with sintactic sugar
def my_decorator(func):
  def wrapper():
    print("First is called this")
    func()
    print("Finally we call this")
  return wrapper

@my_decorator
def reply():
  print("Captain, is Wednesday")

reply()
```

```
First is called this
Captain, is Wednesday
Finally we call this
```

```
# Decorador que acepta funciones
def my_decorator(func):
  def wrapper(*args, **kwargs):
    print("First is called this")
    func(*args, **kwargs)
    print("Finally we call this")
  return wrapper

@my_decorator
def reply(custom_message):
  print(custom_message)

reply("Captain, is Wednesday")
```

```
# Decorador que devuelve valores
def my_decorator(func):
  def wrapper(*args, **kwargs):
    print("First is called this")
    func(*args, **kwargs)
    print("Finally we call this")
    return func(*args, **kwargs)
  return wrapper

@my_decorator
def reply(custom_message):
  print(custom_message)
  return f"{custom_message}!!!"

response = reply("Captain, is Wednesday")
print(response)
```

```
# Preservar la identidad de la función
import functools

def my_decorator(func):
  @functools.wraps(func)
  def wrapper(*args, **kwargs):
    print("First is called this")
    func(*args, **kwargs)
    print("Finally we call this")
    return func(*args, **kwargs)
  return wrapper

@my_decorator
def reply(custom_message):
  print(custom_message)
  return f"{custom_message}!!!"

response = reply("Captain, is Wednesday")
print(response)
```

```
# Decoradores con argumentos

def custom_message(message):
  def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
      print(message)
      func(*args, **kwargs)
      print("Finally we call this")
      return func(*args, **kwargs)
    return wrapper
  return my_decorator

@custom_message("This is the first message")
def reply(custom_message):
  print(custom_message)
  return f"{custom_message}!!!"

response = reply("Captain, is Wednesday")
print(response)


# Using a trigger to execute funtion

def custom_message(message, trigger):
  def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
      print(message)
      if trigger:
        func(*args, **kwargs)
        print("Finally we call this")
        return func(*args, **kwargs)
      else:
        return []
    return wrapper
  return my_decorator

@custom_message("This is the first message test", True)
def reply(custom_message):
  print(custom_message)
  return f"{custom_message}!!!"

response = reply("Captain, is Wednesday")
print(response)
```

Administrador de contextos

Los Administradores de Contexto o **context managers** permiten asignar y liberar recursos cuando lo necesites.

Esta función se controla en Python con el elemento `with` .

Es con esto que se pueden hacer lecturas de documentos sin necesidad de liberar después la lectura o hacer operaciones sin necesidad de gestionar los recursos.

```
# Context Manager

with open('addresses.csv', 'w') as opened_file:
    opened_file.write('Hola!')


# Allocating resources

file = open('some_file', 'w')
try:
    file.write('Hola!')
finally:
    file.close()
```

Pytorch

PyTorch es un framework de aprendizaje profundo desarrollado por Facebook's AI Research lab. Se ha ganado una enorme popularidad en la comunidad científica y de IA gracias a su flexibilidad, eficiencia y capacidad para realizar investigaciones de vanguardia. PyTorch ofrece un ecosistema completo para la investigación y el desarrollo, desde la manipulación de tensores hasta la construcción y entrenamiento de modelos complejos de IA.

Arquitectura

PyTorch se basa en el concepto de tensores, similares a los arrays multidimensionales en NumPy, pero optimizados para cálculos en GPUs, lo que permite aceleraciones significativas en los tiempos de procesamiento. PyTorch también ofrece una interfaz intuitiva y amigable, haciendo que la construcción de modelos sea tan simple como definir una clase en Python.

Uno de los aspectos más destacados de PyTorch es su capacidad para calcular gradientes automáticamente. Esto es posible gracias a su paquete autograd, que rastrea todas las operaciones realizadas en los tensores y construye un gráfico computacional para calcular los gradientes de manera eficiente.

Construcción de modelos con Pytorch

En el caso de Pytorch, introduce ciertos conceptos que veremos en profundidad en la siguiente sesión, de momento nos vamos a concentrar en el uso de las construcciones de Python que hemos visto hasta ahora. A diferencia de Keras, Pytorch tiene una forma más estandarizada de realizar entrenamientos de redes neuronales.

```
import torch
import torch.nn as nn
```