# Classes and subclasses

In this notebook, I will show you the basics of classes and subclasses in Python. As you've seen in the lectures from this week, `Trax` uses layer classes as building blocks for deep learning models, so it is important to understand how classes and subclasses behave in order to be able to build custom layers when needed.

By completing this notebook, you will:

- Be able to define classes and subclasses in Python
- Understand how inheritance works in subclasses
- Be able to work with instances

## Part 1: Parameters, methods and instances

First, let's define a class `My_Class` .

In [1]:
```
class My_Class: #Definition of My_class
    x = None
```

`My_Class` has one parameter `x` without any value. You can think of parameters as the variables that every object assigned to a class will have. So, at this point, any object of class `My_Class` would have a variable `x` equal to `None` . To check this, I'll create two instances of that class and get the value of `x` for both of them.

In [2]:
```
instance_a= My_Class() #To create an instance from class "My_Class" you have to call "My_Class"
instance_b= My_Class()
print('Parameter x of instance_a: ' + str(instance_a.x)) #To get a parameter 'x' from an instance 'a', write 'a.x'
print('Parameter x of instance_b: ' + str(instance_b.x))
```

```
Parameter x of instance_a: None
Parameter x of instance_b: None
```

For an existing instance you can assign new values for any of its parameters. In the next cell, assign a value of `5` to the parameter `x` of `instance_a` .

In [3]:
```
### START CODE HERE (1 line) ###
instance_a.x = 5
### END CODE HERE ###
print('Parameter x of instance_a: ' + str(instance_a.x))
```

```
Parameter x of instance_a: 5
```

## 1.1 The `__init__` method

When you want to assign values to the parameters of your class when an instance is created, it is necessary to define a special method: `__init__` . The `__init__` method is called when you create an instance of a class. It can have multiple arguments to initialize the paramenters of your instance. In the next cell I will define `My_Class` with an `__init__` method that takes the instance ( `self` ) and an argument `y` as inputs.

In [4]:
```
class My_Class:
    def __init__(self, y): # The __init__ method takes as input the 
instance to be initialized and a variable y
        self.x = y          # Sets parameter x to be equal to y
```

In this case, the parameter `x` of an instance from `My_Class` would take the value of an argument `y` . The argument `self` is used to pass information from the instance being created to the method `__init__` . In the next cell, create an instance `instance_c` , with `x` equal to `10` .

In [5]:
```
### START CODE HERE (1 line) ###
instance_c = My_Class(10)
### END CODE HERE ###
print('Parameter x of instance_c: ' + str(instance_c.x))
```

```
Parameter x of instance_c: 10
```

Note that in this case, you had to pass the argument `y` from the `__init__` method to create an instance of `My_Class` .

## 1.2 The `__call__` method

Another important method is the `__call__` method. It is performed whenever you call an initialized instance of a class. It can have multiple arguments and you can define it to do whatever you want like

- Change a parameter,
- Print a message,
- Create new variables, etc.

In the next cell, I'll define `My_Class` with the same `__init__` method as before and with a `__call__` method that adds `z` to parameter `x` and prints the result.

In [6]:
```
class My_Class: 
    def __init__(self, y): # The __init__ method takes as input the 
instance to be initialized and a variable y
        self.x = y         # Sets parameter x to be equal to y
    def __call__(self, z): # __call__ method with self and z as arg
uments
        self.x += z        # Adds z to parameter x when called 
        print(self.x)
```

Let's create `instance_d` with `x` equal to 5.

In [7]:
```
instance_d = My_Class(5)
```

And now, see what happens when `instance_d` is called with argument `10` .

In [8]:
```
instance_d(10)
```

```
15
```

Now, you are ready to complete the following cell so any instance from `My_Class` :

- Is initialized taking two arguments `y` and `z` and assigns them to `x_1` and `x_2` , respectively. And,
- When called, takes the values of the parameters `x_1` and `x_2` , sums them, prints and returns the result.

In [9]:
```
class My_Class:
    def __init__(self, y, z): #Initialization of x_1 and x_2 with a
rguments y and z
        ### START CODE HERE (2 lines) ###
        self.x_1 = y
        self.x_2 = z
        ### END CODE HERE ###
    def __call__(self):       #When called, adds the values of para
meters x_1 and x_2, prints and returns the result
        ### START CODE HERE (1 line) ###
        result = self.x_1 + self.x_2
        ### END CODE HERE ###
        print("Addition of {} and {} is {}".format(self.x_1,self.x_
2,result))
        return result
```

Run the next cell to check your implementation. If everything is correct, you shouldn't get any errors.

In [10]:
```
instance_e = My_Class(10,15)
def test_class_definition():

    assert instance_e.x_1 == 10, "Check the value assigned to x_1"
    assert instance_e.x_2 == 15, "Check the value assigned to x_2"
    assert instance_e() == 25, "Check the __call__ method"

    print("\033[92mAll tests passed!")

test_class_definition()
```

```
Addition of 10 and 15 is 25
All tests passed!
```

## 1.3 Custom methods

In addition to the `__init__` and `__call__` methods, your classes can have custom-built methods to do whatever you want when called. To define a custom method, you have to indicate its input arguments, the instructions that you want it to perform and the values to return (if any). In the next cell, `My_Class` is defined with `my_method` that multiplies the values of `x_1` and `x_2`, sums that product with an input `w`, and returns the result.

In [11]:
```
class My_Class:
    def __init__(self, y, z): #Initialization of x_1 and x_2 with a
rguments y and z
        self.x_1 = y
        self.x_2 = z
    def __call__(self):       #Performs an operation with x_1 and x
_2, and returns the result
        a = self.x_1 - 2*self.x_2
        return a
    def my_method(self, w):   #Multiplies x_1 and x_2, adds argumen
t w and returns the result
        result = self.x_1*self.x_2 + w
        return result
```

Create an instance `instance_f` of `My_Class` with any integer values that you want for `x_1` and `x_2` . For that instance, see the result of calling `My_method` , with an argument `w` equal to `16` .

In [12]:
```
### START CODE HERE (1 line) ###
instance_f = My_Class(1,10)
### END CODE HERE ###
print("Output of my_method:",instance_f.my_method(16))
```

```
Output of my_method: 26
```

As you can corroborate in the previous cell, to call a custom method `m` , with arguments `args` , for an instance `i` you must write `i.m(args)` . With that in mind, methods can call others within a class. In the following cell, try to define `new_method` which calls `my_method` with `v` as input argument. Try to do this on your own in the cell given below.

In [14]:
```
class My_Class:
    def __init__(self, y, z):        #Initialization of x_1 and x_
2 with arguments y and z
        self.x_1 = y
        self.x_2 = z
    def __call__(self):              #Performs an operation with x
_1 and x_2, and returns the result
        a = self.x_1 - 2*self.x_2
        return a
    def my_method(self, w):          #Multiplies x_1 and x_2, adds
argument w and returns the result
        b = self.x_1*self.x_2 + w
        return b
    def new_method(self, v):         #Calls My_method with argumen
t v
        ### START CODE HERE (1 line) ###
        result = self.my_method(v)
        ### END CODE HERE ###
        return result
```

**SPOILER ALERT** Solution:

```
In [15]: # hidden-cell
         class My_Class:
             def __init__(self, y, z):      #Initialization of x_1 and x_2 w
         ith arguments y and z
                 self.x_1 = y
                 self.x_2 = z
             def __call__(self):            #Performs an operation with x_1
         and x_2, and returns the result
                 a = self.x_1 - 2*self.x_2
                 return a
             def my_method(self, w):        #Multiplies x_1 and x_2, adds ar
         gument w and returns the result
                 b = self.x_1*self.x_2 + w
                 return b
             def new_method(self, v):       #Calls My_method with argument v
                 result = self.my_method(v)
                 return result
```

```
In [16]: instance_g = My_Class(1,10)
         print("Output of my_method:",instance_g.my_method(16))
         print("Output of new_method:",instance_g.new_method(16))
```

```
Output of my_method: 26
Output of new_method: 26
```

## Part 2: Subclasses and Inheritance

`Trax` uses classes and subclasses to define layers. The base class in `Trax` is `layer` , which means that every layer from a deep learning model is defined as a subclass of the `layer` class. In this part of the notebook, you are going to see how subclasses work. To define a subclass `sub` from class `super` , you have to write `class sub(super):` and define any method and parameter that you want for your subclass. In the next cell, I define `sub_c` as a subclass of `My_Class` with only one method ( `additional_method` ).

```
In [17]: class sub_c(My_Class):           #Subclass sub_c from My_class
             def additional_method(self): #Prints the value of parameter x_1
                 print(self.x_1)
```

### 2.1 Inheritance

When you define a subclass `sub` , every method and parameter is inherited from `super` class, including the `__init__` and `__call__` methods. This means that any instance from `sub` can use the methods defined in `super` . Run the following cell and see for yourself.

In [18]:
```
instance_sub_a = sub_c(1,10)
print('Parameter x_1 of instance_sub_a: ' + str(instance_sub_a.x_1))
print('Parameter x_2 of instance_sub_a: ' + str(instance_sub_a.x_2))
print("Output of my_method of instance_sub_a:",instance_sub_a.my_method(16))
```

```
Parameter x_1 of instance_sub_a: 1
Parameter x_2 of instance_sub_a: 10
Output of my_method of instance_sub_a: 26
```

As you can see, `sub_c` does not have an initialization method `__init__` , it is inherited from `My_class` . However, you can overwrite any method you want by defining it again in the subclass. For instance, in the next cell define a class `sub_c` with a redefined `my_Method` that multiplies `x_1` and `x_2` but does not add any additional argument.

In [19]:
```
class sub_c(My_Class):                #Subclass sub_c from My_class
    def my_method(self):              #Multiplies x_1 and x_2 and returns the result
        ### START CODE HERE (1 line) ###
        b = self.x_1*self.x_2
        ### END CODE HERE ###
        return b
```

To check your implementation run the following cell.

In [20]:
```
test = sub_c(3,10)
assert test.my_method() == 30, "The method my_method should return the product between x_1 and x_2"

print("Output of overridden my_method of test:",test.my_method()) # notice we didn't pass any parameter to call my_method
#print("Output of overridden my_method of test:",test.my_method(16)) #try to see what happens if you call it with 1 argument
```

```
Output of overridden my_method of test: 30
```

In the next cell, two instances are created, one of `My_Class` and another one of `sub_c` . The instances are initialized with equal `x_1` and `x_2` parameters.

In [21]:
```
y,z= 1,10
instance_sub_a = sub_c(y,z)
instance_a = My_Class(y,z)
print('My_method for an instance of sub_c returns: ' + str(instance_sub_a.my_method()))
print('My_method for an instance of My_Class returns: ' + str(instance_a.my_method(10)))
```

```
My_method for an instance of sub_c returns: 10
My_method for an instance of My_Class returns: 20
```

As you can see, even though `sub_c` is a subclass from `My_Class` and both instances are initialized with the same values, `My_method` returns different results for each instance because you overwrote `My_method` for `sub_c` .

**Congratulations!** You just reviewed the basics behind classes and subclasses. Now you can define your own classes and subclasses, work with instances and overwrite inherited methods. The concepts within this notebook are more than enough to understand how layers in `Trax` work.