Goal:
writing code like this:

including a parent class (here from `keras.models`, the `add` method from the `Sequential` class)

using `super` for accessing methods of a parent class (here `Sequential`) from within a child class (here `MyLeNet`)

```
class MyLeNet(Sequential):

    def __init__(self, input_shape, num_classes):
        super().__init__()

#building LeNet ##################################################################################################################
        #Note padding: string, either "valid" or "same" (case-insensitive). "valid" means no padding. "same"
        #more info: https://keras.io/api/layers/convolution_layers/convolution2d/

        self.add(Conv2D(6,   kernel_size = (5, 5),    strides = (1, 1), activation = 'tanh', input_shape = input_shape, padding = 'same'))
        self.add(AveragePooling2D(pool_size = (2, 2), strides = (2, 2),                                               padding = 'valid'))
        self.add(Conv2D(16,  kernel_size = (5, 5),    strides = (1, 1), activation = 'tanh',                           padding = 'valid'))
        self.add(AveragePooling2D(pool_size = (2, 2), strides = (2, 2),                                               padding = 'valid'))
        self.add(Conv2D(120, kernel_size = (5, 5),    strides = (3, 3), activation = 'tanh',                           padding = 'valid'))
        self.add(Flatten())
        self.add(Dense(84,                                              activation = 'tanh'))
        self.add(Dense(num_classes,                                     activation = 'softmax'))
        ##################################################################################################################
```

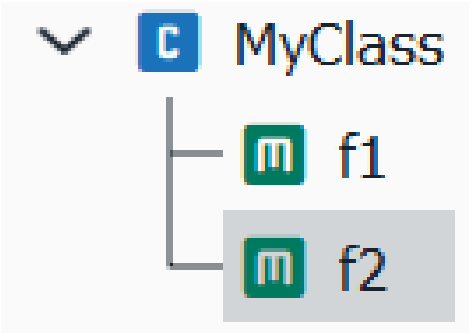
Goal:
writing code like this:

→ **repeating**

```
classes
methods
map, lambda
dicts, arrays and lists
```

Let's start from the beginning:

```
class MyClass():

    def f1(self, a, b):
        self.res = a + b

    def f2(self, c):

        return self.res*c
```



our codes become more complex

→ make it modular
→ avoid redundances
→ make it flexible
→ consider multiprocessing

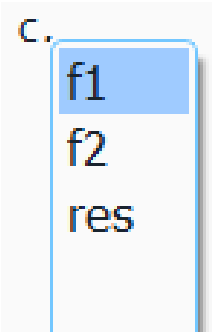
- ***f1*** and ***f2*** are methods of the class

- initialize an **instance** of a class:

```
C = MyClass()
```

- try `dir(C)`

- try `C.f1(3,4)`



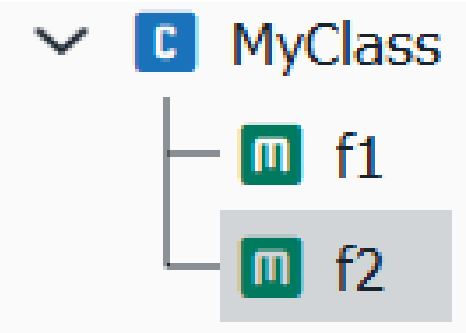
called **attributes** of a class
(can be functions, variables, etc)

try:

```
getattr(C, 'res')
getattr(C, 'f1')
my_fun = getattr(C, 'f1')
my_fun(3,4)
```

Let's start from the beginning:

```
class MyClass():

    def f1(self, a, b):
        self.res = a + b

    def f2(self, c):

        return self.res*c
```



our codes become more complex

→ make it modular
→ avoid redundances
→ make it flexible
→ consider multiprocessing

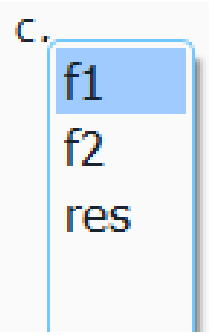
What is the purpose of *self*?

- ***f1*** and ***f2*** are methods of the class

- initialize an **instance** of a class:
  `C = MyClass()`

- try `dir(C)`

- try `C.f1(3,4)`



called **attributes** of a class
(can be functions, variables, etc)

try:

```
getattr(C, 'res')
getattr(C, 'f1')
my_fun = getattr(C, 'f1')
my_fun(3, 4)
```

We can also *inherit* attributes:

```
class C0():

        def f0(self, a, b):
                res = a * b
                return res
```

```
class C1():

        def f1(self, a, b):
                res = a + b
                return res
```

```
class C2(C1, C0):

        def f2(self, a, b, c, d):
                return a+b+c+d
```

run:

```
C = C2()
```

C2 ***inherited*** the attributes from its ***parents*** C0 and C1
A ***child*** class (here C2) can have any number of parents!

run:

```
C.f0(2, 3)
C.f1(2, 3)
C.f2(2, 3, 4, 5)
```

Since we need to initialize an instance of a class, it is often useful to add an __init__ method:

```
class Encoder():

    def __init__(self):

        NT    = 'ACGT'
        Code = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]

        Dict = {key: value for key, value in zip(NT,Code)}

        self.Enc = lambda Sequence: [Dict[s] for s in Sequence]


    def Encode(self, Sequence):

        print(self.Enc(Sequence))
```

Try do understand, what this code does and why the **__init__** is useful!

Often, we want to call a method of a parent class *within* a method of a child class

```
class C2(C1, C0):

        def f2(self, a, b, c, d):
                res = C1.f1(a,b)
                return res+c+d
```

```
C = C2()

C.f
```

f0
f1
f2

```
C.f2(2,3,5,7)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[137], line 1
----> 1 C.f2(2,3,5,7)

Cell In[132], line 4, in C2.f2(self, a, b, c, d)
      3 def f2(self, a, b, c, d):
----> 4     res = C1.f1(a,b)
      5     return res+c+d

TypeError: C1.f1() missing 1 required positional argument: 'b'
```

Error occurs because it didn't pass on *self*

Often, we want to call a method of a parent class *within* a method of a child class

```
class C2(C1, C0):

        def f2(self, a, b, c, d):
                res = super().f1(a,b)
                return res+c+d
```

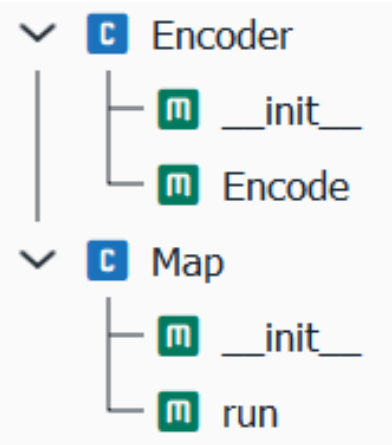
```
C.f2(2, 3, 5, 7)
```

```
C.f2(2,3,5,7)
17
```

Works with f2 too!

Often, we want to call a method of a parent class *within* a method of a child class

```
class Encoder():

    def __init__(self):

        NT    = 'ACGT'
        Code = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]
        Dict = {key: value for key, value in zip(NT,Code)}
        self.Enc = lambda Sequence: [Dict[s] for s in Sequence]

    def Encode(self, Sequence):

        print(self.Enc(Sequence))


    class Map(Encoder):

        def __init__(self):
            super().__init__()

        def run(self, S):
            return list(map(self.Enc, S))
```




```
run:
M = Map()
M.run(['AACCTG', 'TTGTG'])
```

Exercise:

```
class Encoder():

    def __init__(self):

        NT    = 'ACGT'
        Code = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]
        Dict = {key: value for key, value in zip(NT,Code)}
        self.Enc = lambda Sequence: [Dict[s] for s in Sequence]

    def Encode(self, Sequence):

        print(self.Enc(Sequence))


    class Map(Encoder):

        def __init__(self):
            super().__init__()

        def run(self, S):
            return list(map(self.Enc, S))
```

Change this line such that it returns an `np.array`. Make sure that the array is correctly oriented!

Change this line such that it calls the **Encode** method from the **Encoder**, via `super`

```
run:
M = Map()
Out = M.run(['AACCTG', 'TTGTG'])
```