Pourya Mansouri - Homework 4

tamrin 1

1- استفاده MRO برای پیدا کردن مسیر توابع در اثربری وقتی از چند کلاس والد استفاده میکنیم از تابع میکنیم.

In [ ]:

```
```

In [1]:

```
class A:
    def do_job(self):
        print('I am walking ...')


class Z:
    def do_job(self, n):
        print(f'I am counting from 1 to {n}:  {list(range(1, n + 1))}')


class B(A):
    def do_job(self, s):
        super().do_job()
        print(f'I am printing your string : "{s}"')


class C(A, Z):
    def do_job(self, n):
        super().do_job(n)
        print('I am jumping ...')


class D(B):
    def do_job(self, s):
        super().do_job(s)
        print('I am speaking ...')


class E(D, C):
    def do_job(self, s, n):
        super().do_job(s, n)
        print('I am laughing ...')


class F(Z, B):
    def do_job(self, s, n):
        super().do_job(s, n)
        print('I am playing ...')
```

In [3]:

```
E.mro()
```

Out[3]:

```
[__main__.E,
 __main__.D,
 __main__.B,
 __main__.C,
 __main__.A,
 __main__.Z,
 object]
```

همانطور که نتیجه تابع mro مشخص هست مسیر حرکت در مسیر زیر می باشد: E --> D --> B --> C -->

A --> Z

In [5]:
```
obja = A()
obja.do_job()

print()
objz = Z()
objz.do_job(3)

print()
obje = E()
obje.do_job('Python', 5)

print()
objf = F()
objf.do_job('Python', 6)
```

```
I am walking ...

I am counting from 1 to 3:  [1, 2, 3]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-e50209e3aa83> in <module>
      8 print()
      9 obje = E()
---> 10 obje.do_job('Python', 5)
     11
     12 print()

<ipython-input-4-10463e3962f1> in do_job(self, s, n)
     29 class E(D, C):
     30     def do_job(self, s, n):
---> 31         super().do_job(s, n)
     32         print('I am laughing ...')
     33

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

2- ارور TpyeError داریم که میگه تابع do_job() ورودی دارد. فقط ۲ ورودی میگیرد وبی ما به آن ۳ را میگیرد. ولی در کلاس n و s ورودی self بجز E در کلاس do_job() مشکل از اینجاست که با اینکه تابع صدا میزنیم E را در کلاس super یک ورودی میگیرد . وقتی ما تابعself این تابع بجز D والد آن یعنی کلاس می شود ولی تعداد ورودی داده شده به آن بیشتر از حد مجازD وارد کلاس mro می بینیم که با توجه به تابع است.

تمرین ۳ -

- Reference:
  - https://python-reference.readthedocs.io/

__call VS \init__

init is called when you are creating an instance of any class and initializing the instance variable also. And **call** is called when you call the object like any other function. **call** allows to return arbitrary values, while **init** being an constructor returns the instance of class implicitly.

In [5]:
```
class A:
    def __init__(self, y):
        print("inside __init__(), y: ",y)
        self.y = y
```

```
    def __call__(self):
        res = 0
        print("inside __call__()")
        print("adding 2 to the value of y")
        res = self.y + 2
        return res
```

In [6]:

```
# declaration of instance of class A
a = A(3)
print()

# calling __call__() for a
r = a()
print(r)
print()
# declaration of another instance
# of class A
b = A(10)
print()


# calling __call__() for b
r = b()
print(r)
```

```
inside __init__(), y:  3

inside __call__()
adding 2 to the value of y
5

inside __init__(), y:  10

inside __call__()
adding 2 to the value of y
12
```

- __del__

- The **__del__()** method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected

In [10]:

```
class Object:

        def __init__(self):
                print('Object created.')

        # Deleting (Calling destructor)
        def __del__(self):
                print('Destructor called, Object deleted.')

obj = Object()
del obj

obj
```

```
Object created.
Destructor called, Object deleted.
```

```
NameError                                 Traceback (most recent call last)
<ipython-input-10-7739dca20187> in <module>
     11 del obj
     12
---> 13 obj

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

- __getattr__

- Called when an attribute lookup has not found the attribute in the usual places (i.e. it is not an instance attribute nor is it found in the class tree for self).

In [25]:
```
class Object:
    def __init__(self, bamf):
        self.bamf = bamf
    def __getattr__(self, name):
        return 'Object does not have `{}` attribute.'.format(str(name))
```

In [26]:
```
f = Object("bamf")
print(f.bar)
print()
print(f.bamf)
```

```
Object does not have `bar` attribute.

bamf
```

In [37]:
```
class Object:
    def __init__(self, bamf):
        self.bamf = bamf
    def __getattr__(self, name):
#         setattr(self,name,str(name))
        return getattr(Object, name, f'{name} Not founded')
```

In [38]:
```
f = Object("bamf")
print(f.bar)
print()
print(f.bamf)
```

```
bar Not founded

bamf
```

- __setattr__

- This method is called instead of the normal mechanism (i.e. store the value in the instance dictionary).
- If __setattr__() wants to assign to an instance attribute, it should not simply execute self.name = value — this would cause a recursive call to itself.
- Instead, it should insert the value in the dictionary of instance attributes, e.g., self.__dict__[name] = value.

In [39]:
```
# this example uses __setattr__ to dynamically change attribute value to uppe
class Frob:
    def __setattr__(self, name, value):
```

```
        self.__dict__[name] = value.upper()

f = Frob()
f.bamf = "bamf"
f.bamf
```

Out[39]: 'BAMF'

__new__

- __**new**__ is the first step of instance creation. It's called first and is responsible for returning a new instance of your class. In contrast, __**init**__ doesn't return anything; it's only responsible for initializing the instance after it's been created

In [45]:
```
class Person:

    def __new__(cls):
        print("New")
        return object.__new__(cls)

    def __init__(self):
        self.instance_method()

    def instance_method(self):
        print('success!')

personObj = Person()
```

```
New
success!
```