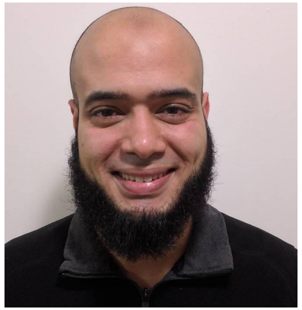
Python Programming

Multiple Inheritance with Super

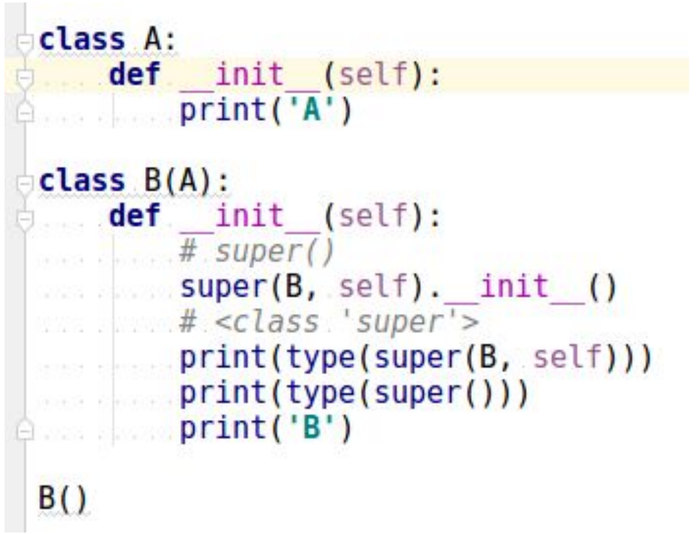
Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



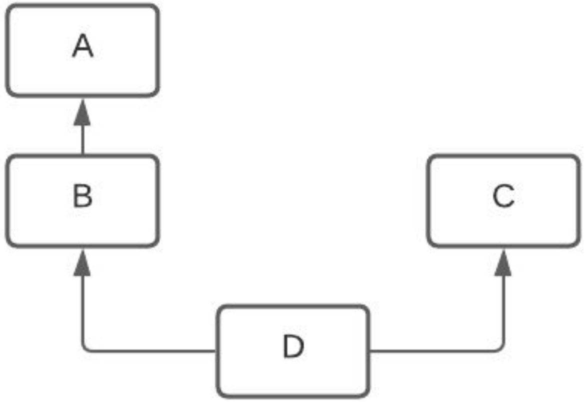
super() function

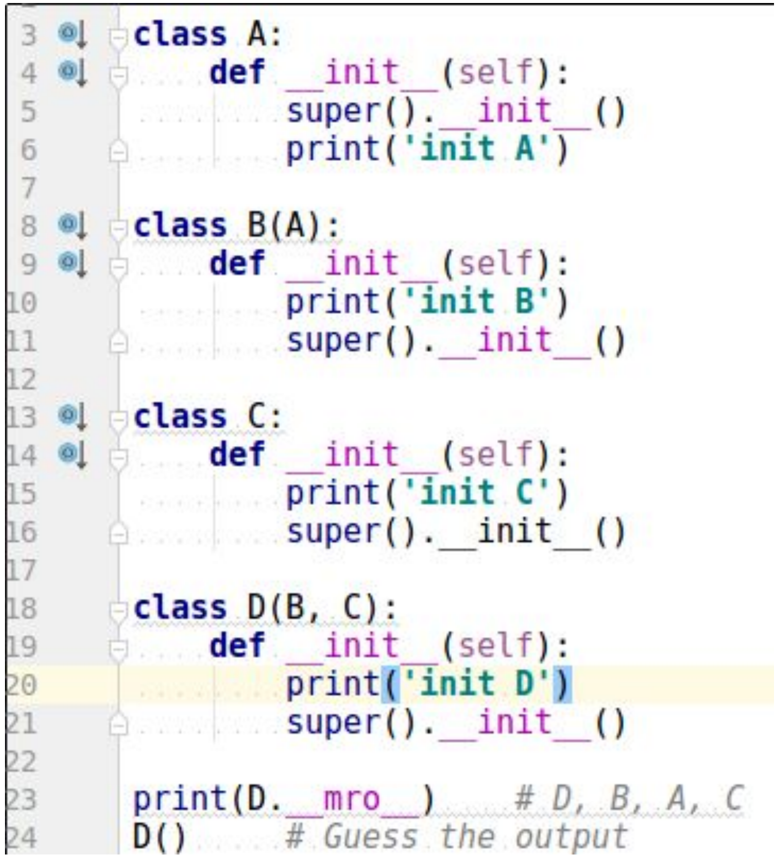
- *If you don’t get most of today or delay = no problem*
- We previously mentioned: super() returns an object of the parent superclass
    - This is not so accurate. It has 2 mistakes.
    - It returns a **proxy object** (think the wrapper we took for now). It will **delegate** the call to a specific class. This is *not an important* part.
    - The returned class is NOT necessarily your parent! This is a *critical* part
        - Yah super() is more complicated
- super() itself is an abbreviation for
    - super(class, self)



```
class A:
    def __init__(self):
        print('A')

class B(A):
    def __init__(self):
        # super()
        super(B, self).__init__()
        # <class 'super'>
        print(type(super(B, self)))
        print(type(super()))
        print('B')

B()
```

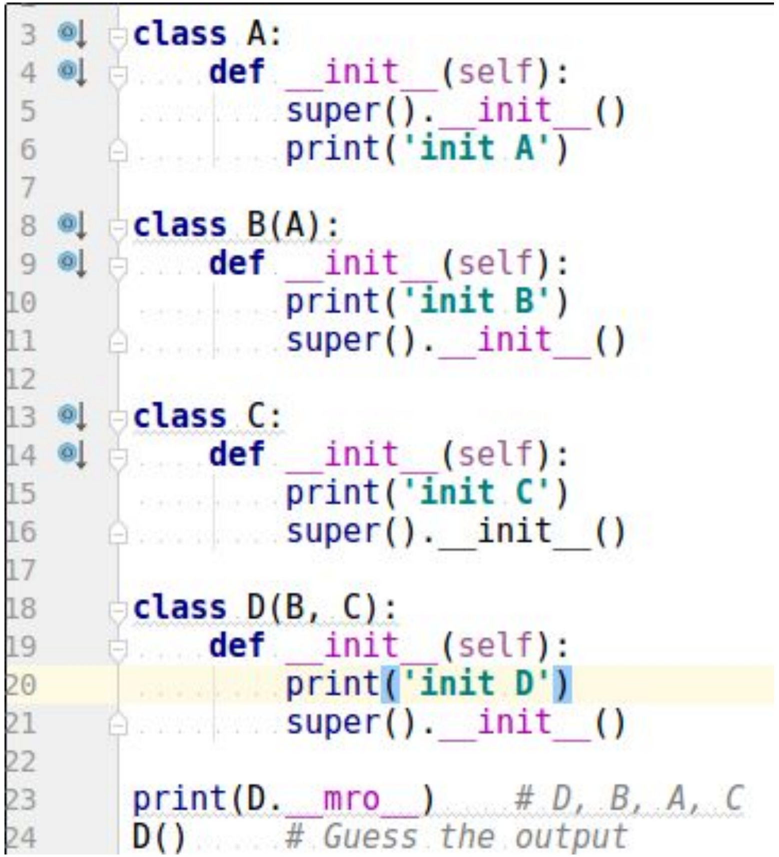
Guess the output




```
class A:
    def __init__(self):
        super().__init__()
        print('init A')

class B(A):
    def __init__(self):
        print('init B')
        super().__init__()

class C:
    def __init__(self):
        print('init C')
        super().__init__()

class D(B, C):
    def __init__(self):
        print('init D')
        super().__init__()

print(D.__mro__)    # D, B, A, C
D()     # Guess the output
```

Super and MRO

- Guessed init D, B, A?
  - Good trial, but wrong
  - Answer: init D, B, A, **C**
- The super() call finds the next method in the **MRO** at each step NOT necessarily one of your parents
  - At D, what is next? B. Super goes B.init
  - At B, what is next? A
  - At A, what is next? C
- Wait but A has NO parent?!
  - It is about MRO, not parents

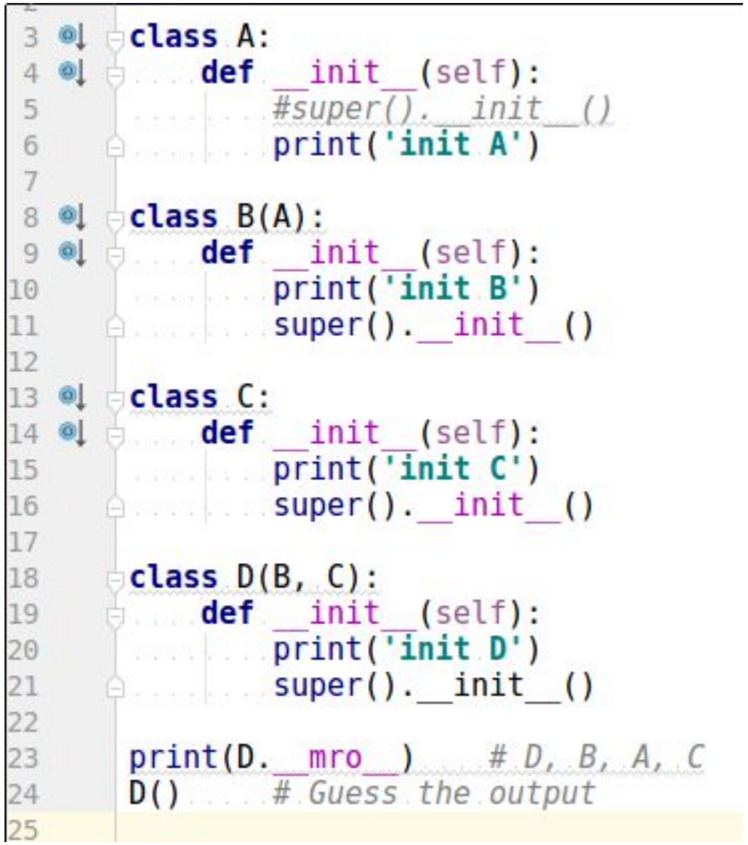


```python
class A:
    def __init__(self):
        super().__init__()
        print('init A')

class B(A):
    def __init__(self):
        print('init B')
        super().__init__()

class C:
    def __init__(self):
        print('init C')
        super().__init__()

class D(B, C):
    def __init__(self):
        print('init D')
        super().__init__()

print(D.__mro__)    # D, B, A, C
D()     # Guess the output
```

Super and MRO

- Let's comment line 5
- Guess the output?

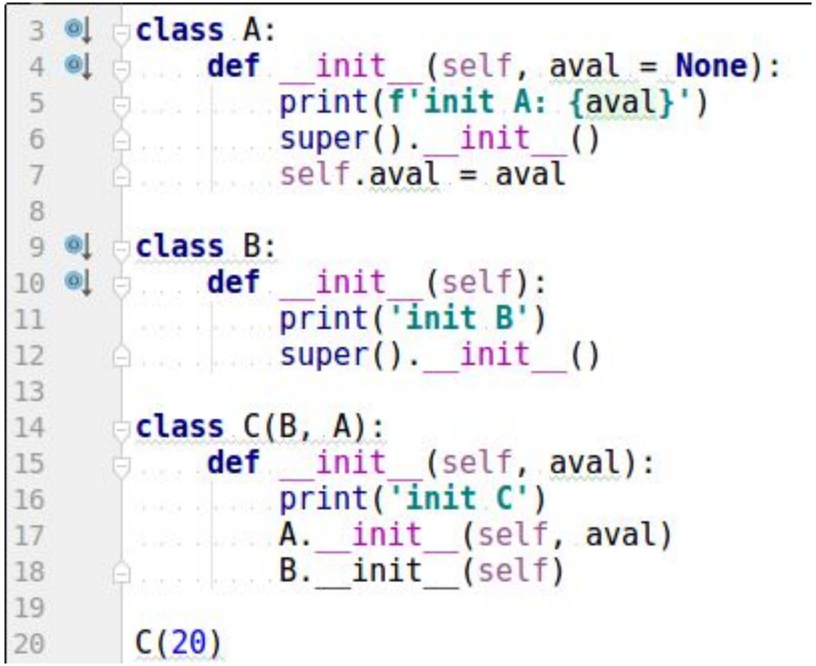
- Init D, B, A
- As A doesn't make call for super, we stopped at this point.
  - No one is calling C.
- Super() here is doing very interesting work, but also this could be so annoying!



```
 3  class A:
 4      def __init__(self):
 5          #super().__init__()
 6          print('init A')
 7
 8  class B(A):
 9      def __init__(self):
10          print('init B')
11          super().__init__()
12
13  class C:
14      def __init__(self):
15          print('init C')
16          super().__init__()
17
18  class D(B, C):
19      def __init__(self):
20          print('init D')
21          super().__init__()
22
23  print(D.__mro__)    # D, B, A, C
24  D()    # Guess the output
25
```

Guess the output

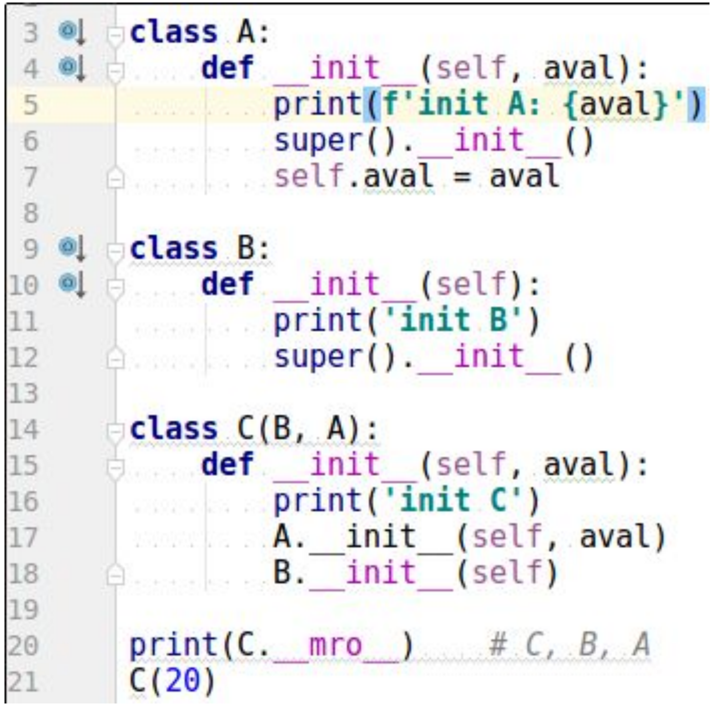
- C.MRO = C, B, A
- At C
  - **init C**
  - Call explicitly A with 20
    - At A
      - **init A: 20**
      - super() calls after A ⇒ object
  - Call explicitly B
    - **init B**
      - super() calls after B ⇒ A
        - At A
          - **init A: None**



```python
class A:
    def __init__(self, aval = None):
        print(f'init A: {aval}')
        super().__init__()
        self.aval = aval

class B:
    def __init__(self):
        print('init B')
        super().__init__()

class C(B, A):
    def __init__(self, aval):
        print('init C')
        A.__init__(self, aval)
        B.__init__(self)

C(20)
```

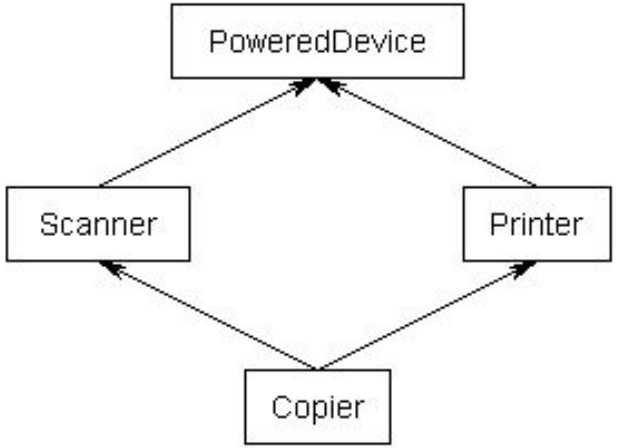
Guess the output

- init C
- init A: 20
- init B
- TypeError: __init__() missing 1 required positional argument: 'aval'
  - At line 12
- This will be shocking for some guys
  - B has no parent
  - super() init calls object init
  - Why do we need parameter?
  - We are actually calling A init NOT object init



```
 3  class A:
 4      def __init__(self, aval):
 5          print(f'init A: {aval}')
 6          super().__init__()
 7          self.aval = aval
 8
 9  class B:
10      def __init__(self):
11          print('init B')
12          super().__init__()
13
14  class C(B, A):
15      def __init__(self, aval):
16          print('init C')
17          A.__init__(self, aval)
18          B.__init__(self)
19
20  print(C.__mro__)    # C, B, A
21  C(20)
```

The Diamond Problem



Img Src

- We know object class is common to all
- Sometimes we build such diamonds
- There are 2 issues
  - Language issue
    - Some languages are harder to handle
    - Python is good with MRO
  - Development issue
    - We get confused about function calls and typically do errors
- Tip: Don’t do such style!

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."