CLASSES/SUBCLASSES REVIEW

Presented By: Tiffany, Riya, Cornelius, Ian

In This Presentation

Drawing Classes and Subclasses

- We're going to draw class folders and then diagram some code using call frames with classes and subclass

Creating Classes and Subclasses

- We're then going to create a class and two subclasses for practice!

DRAWING PRACTICE

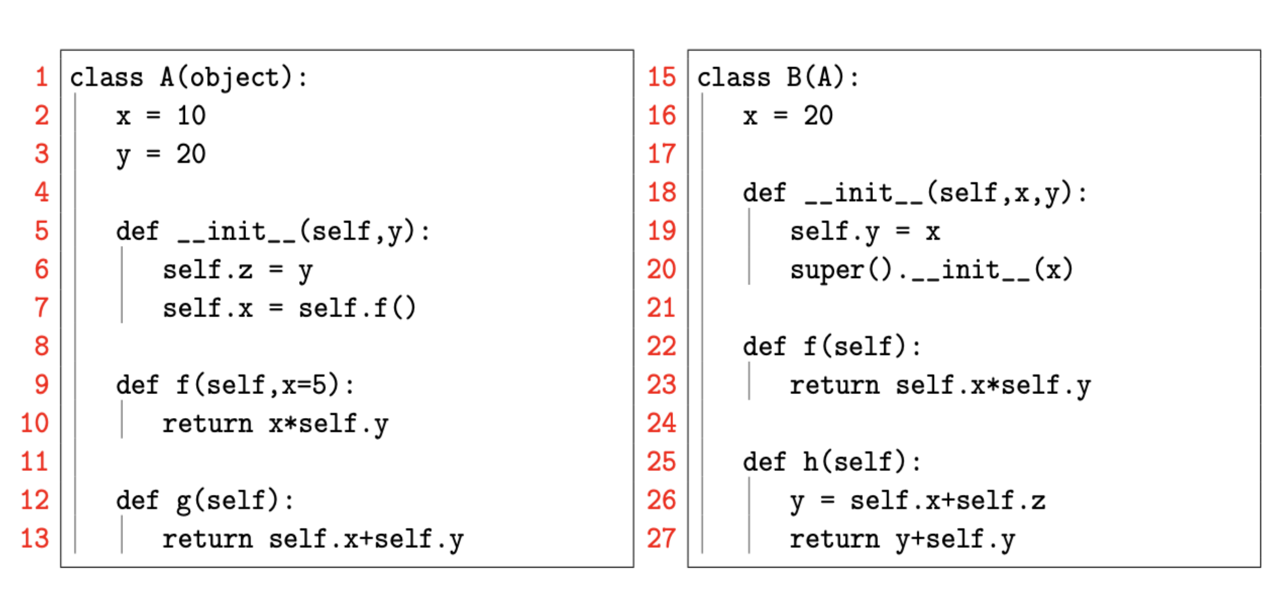
Let's draw folders and call frames together!

Let's Draw This Code

```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

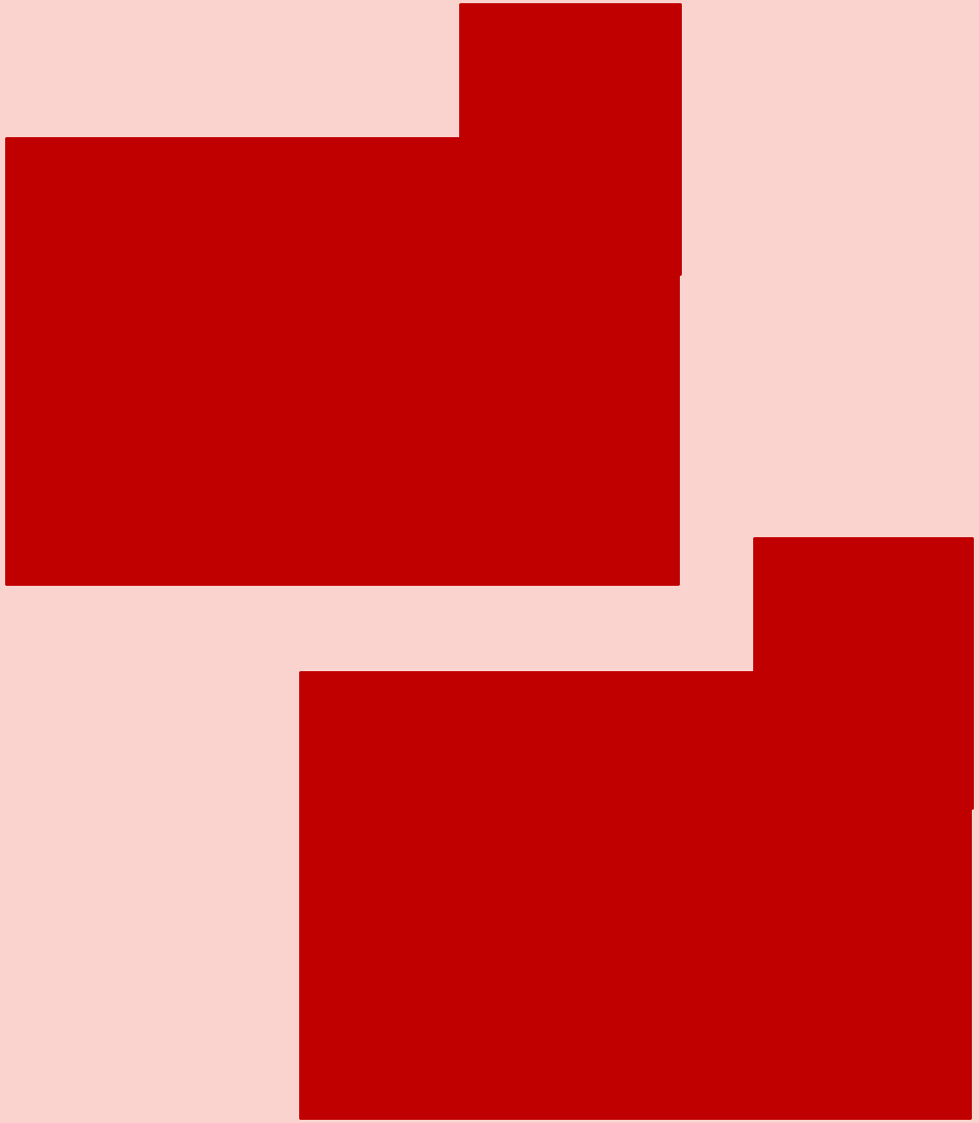
```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```

Let's Draw This Code



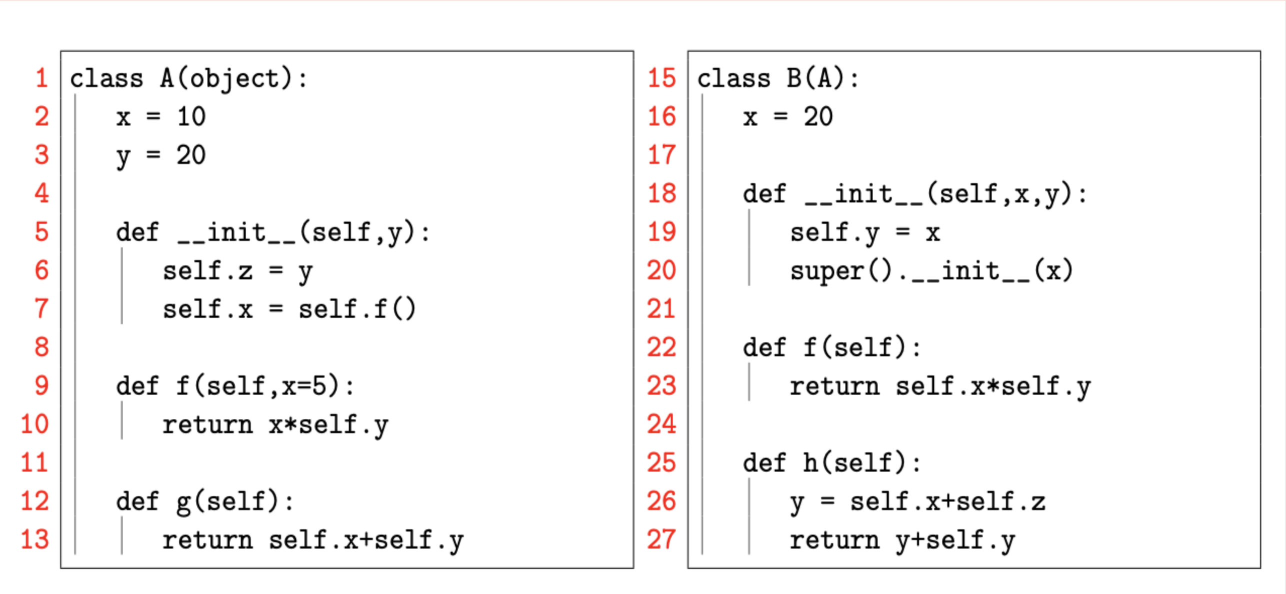
```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```



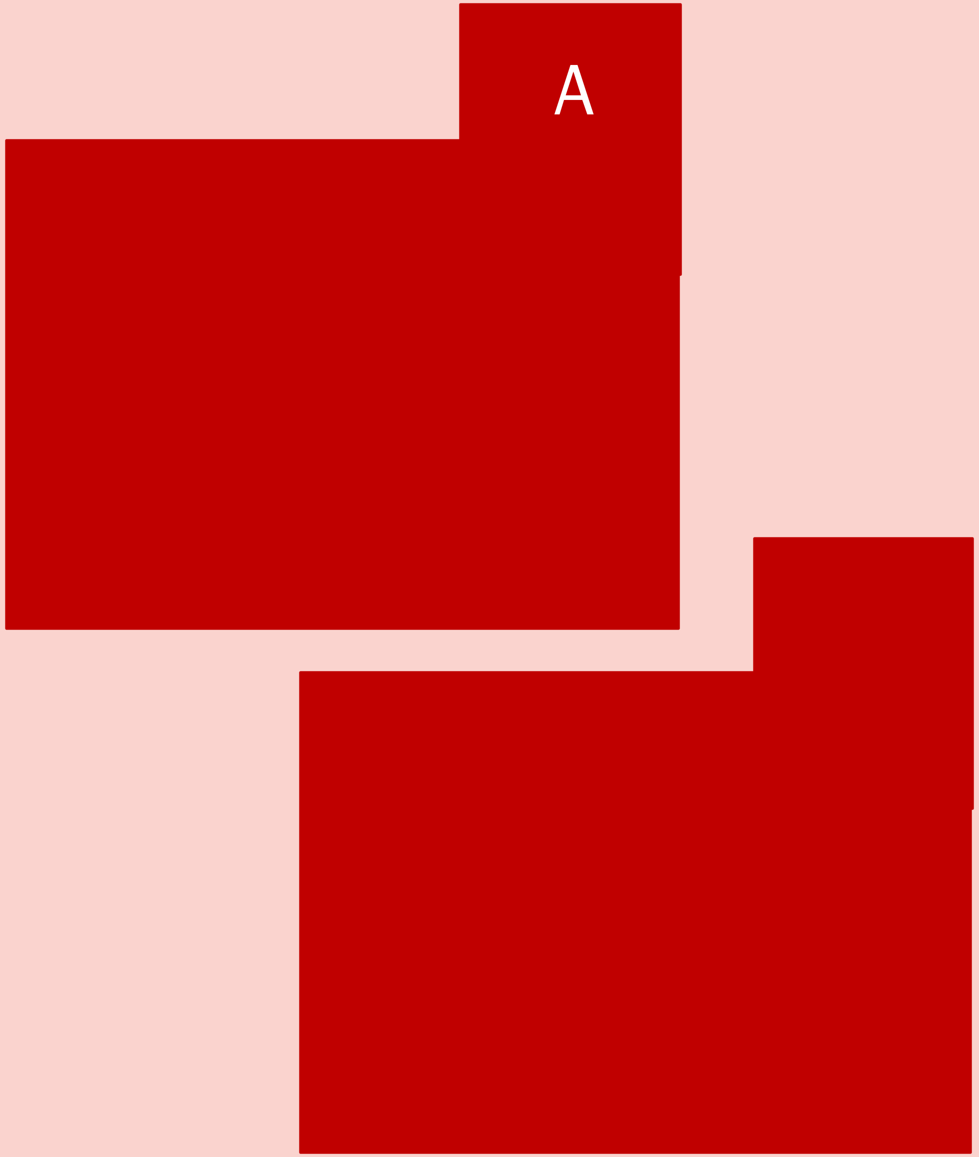
First, let's draw
the class folders!

Let's Draw This Code



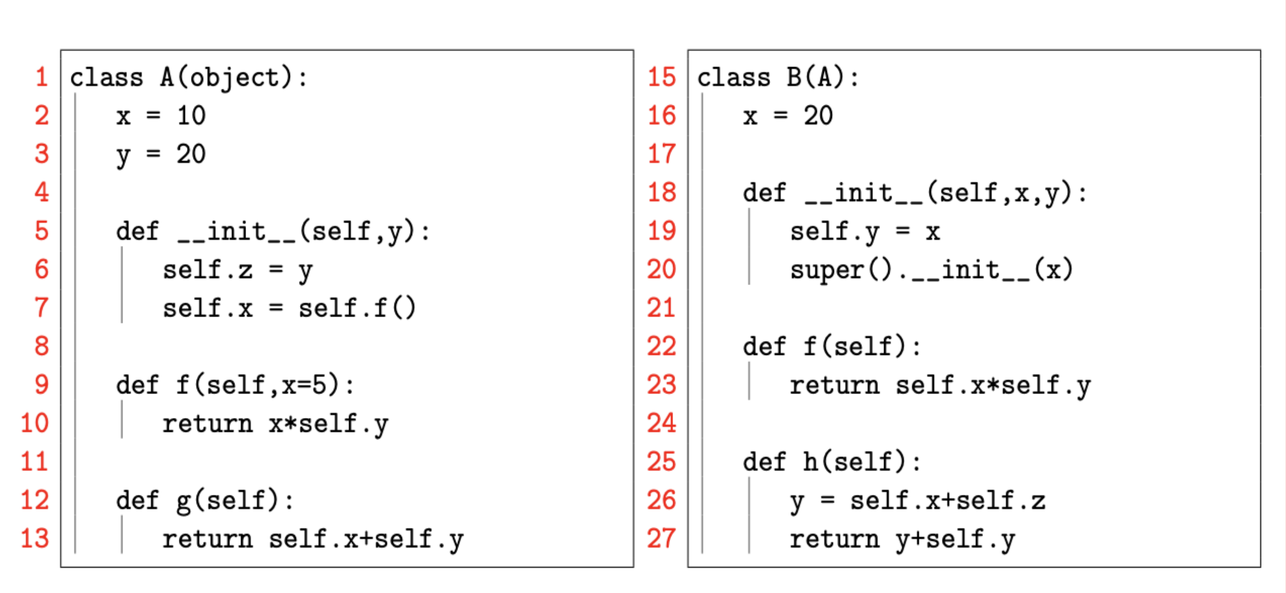
```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```



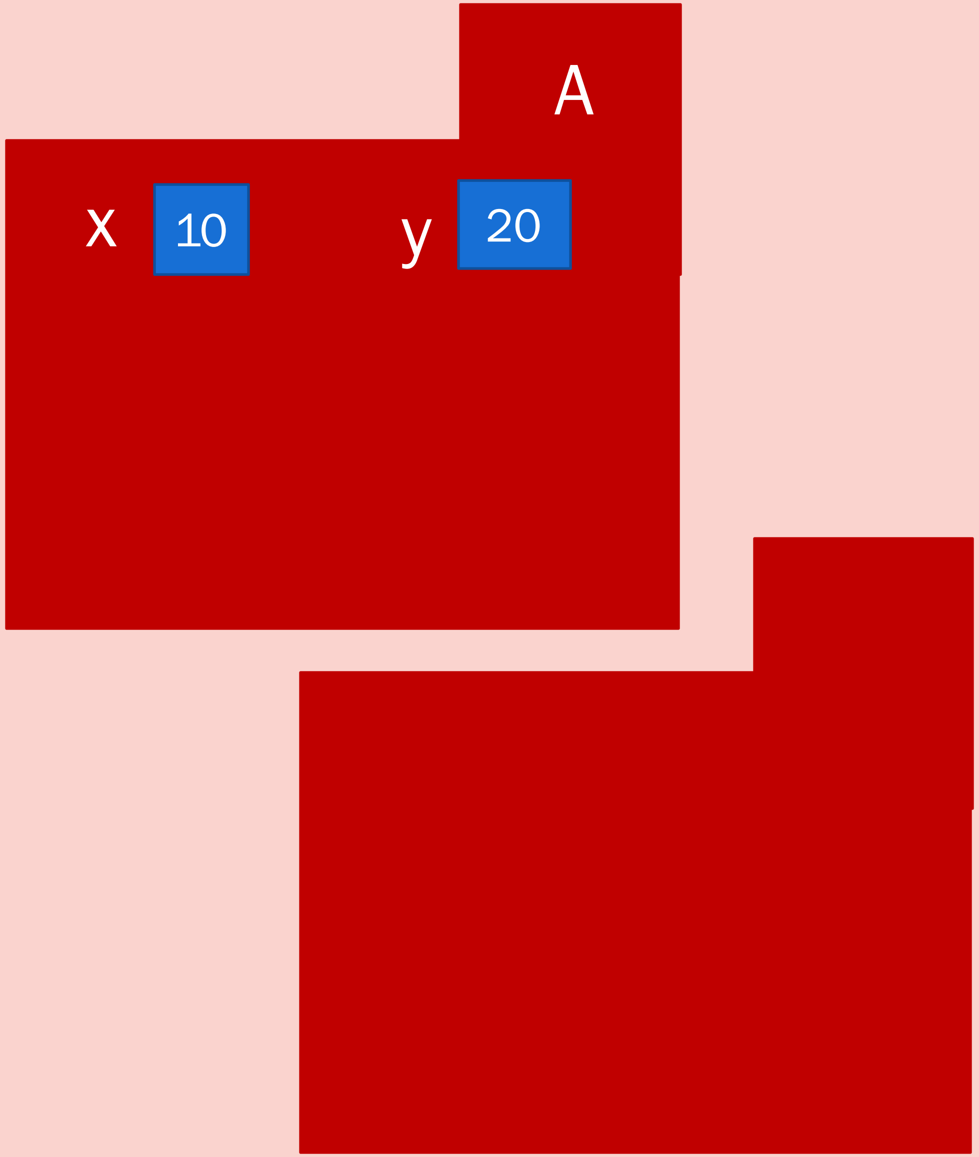
Always put class names in upper RIGHT tab!

Let's Draw This Code



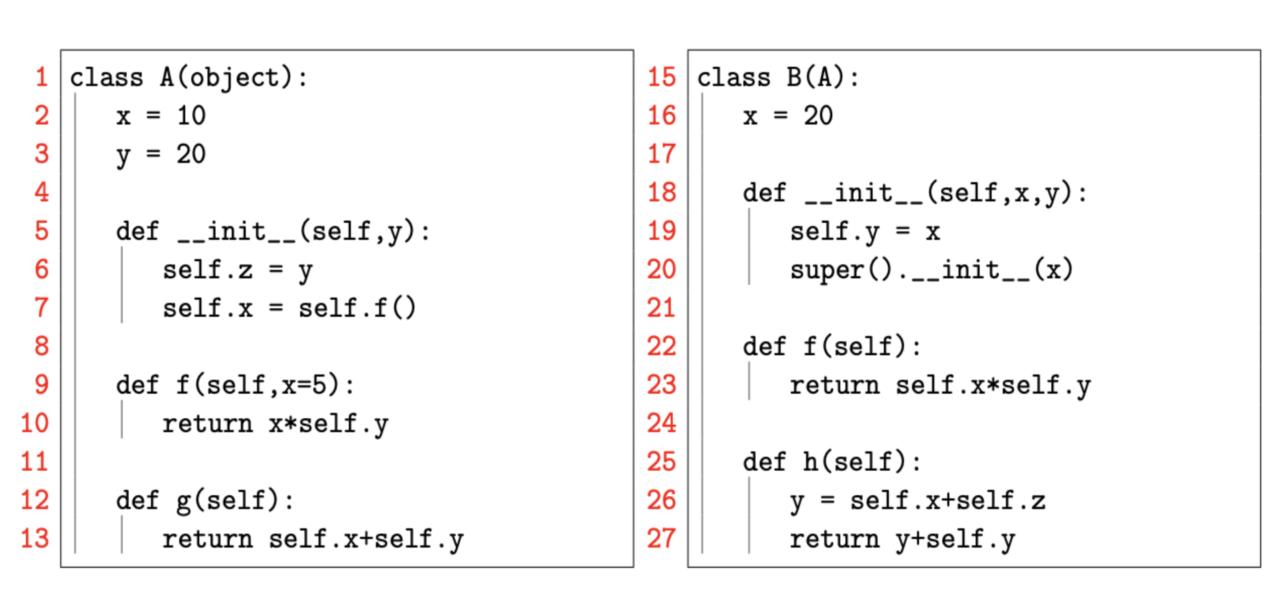
```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```



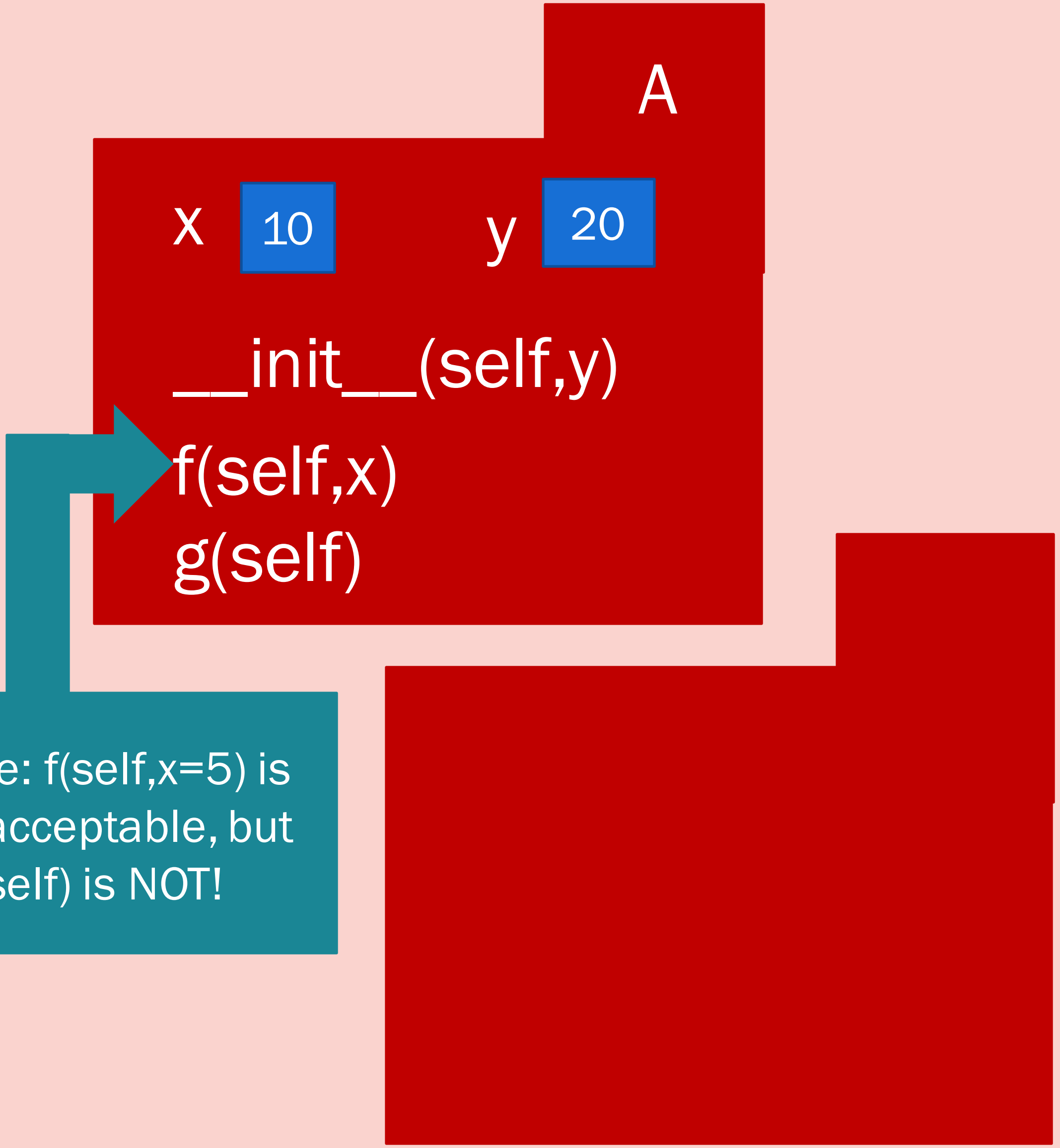
Then, add class attributes

Let's Draw This Code



```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```

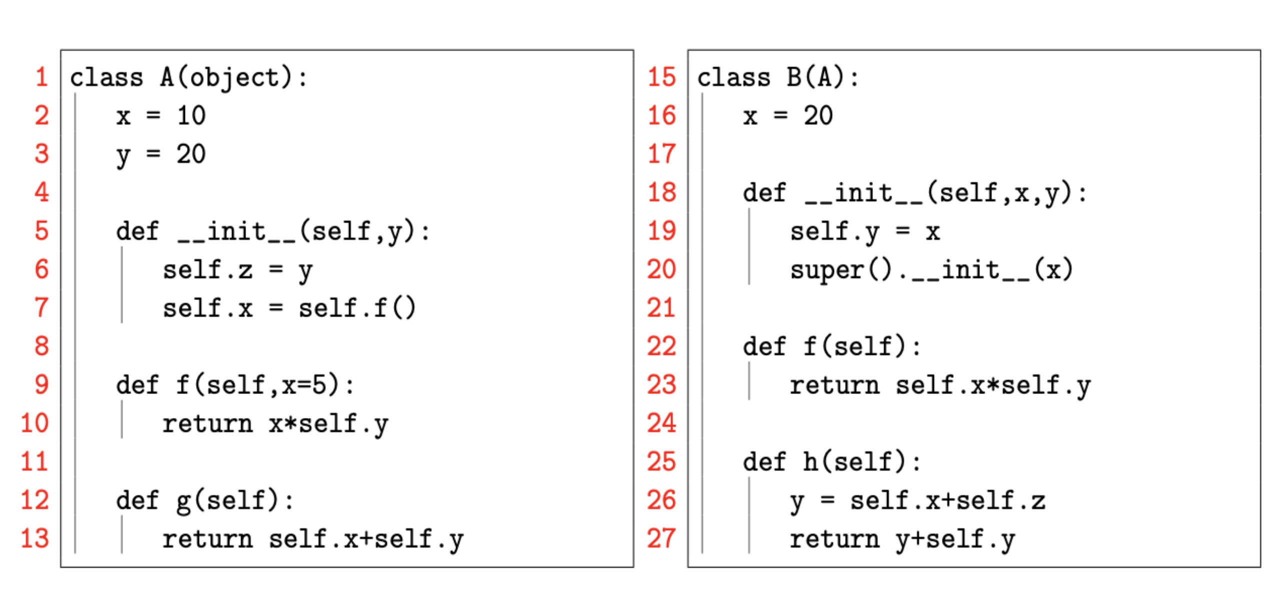


Notice: f(self,x=5) is also acceptable, but f(self) is NOT!

Next, add it methods

Then, the folder for A is done!

Let's Draw This Code

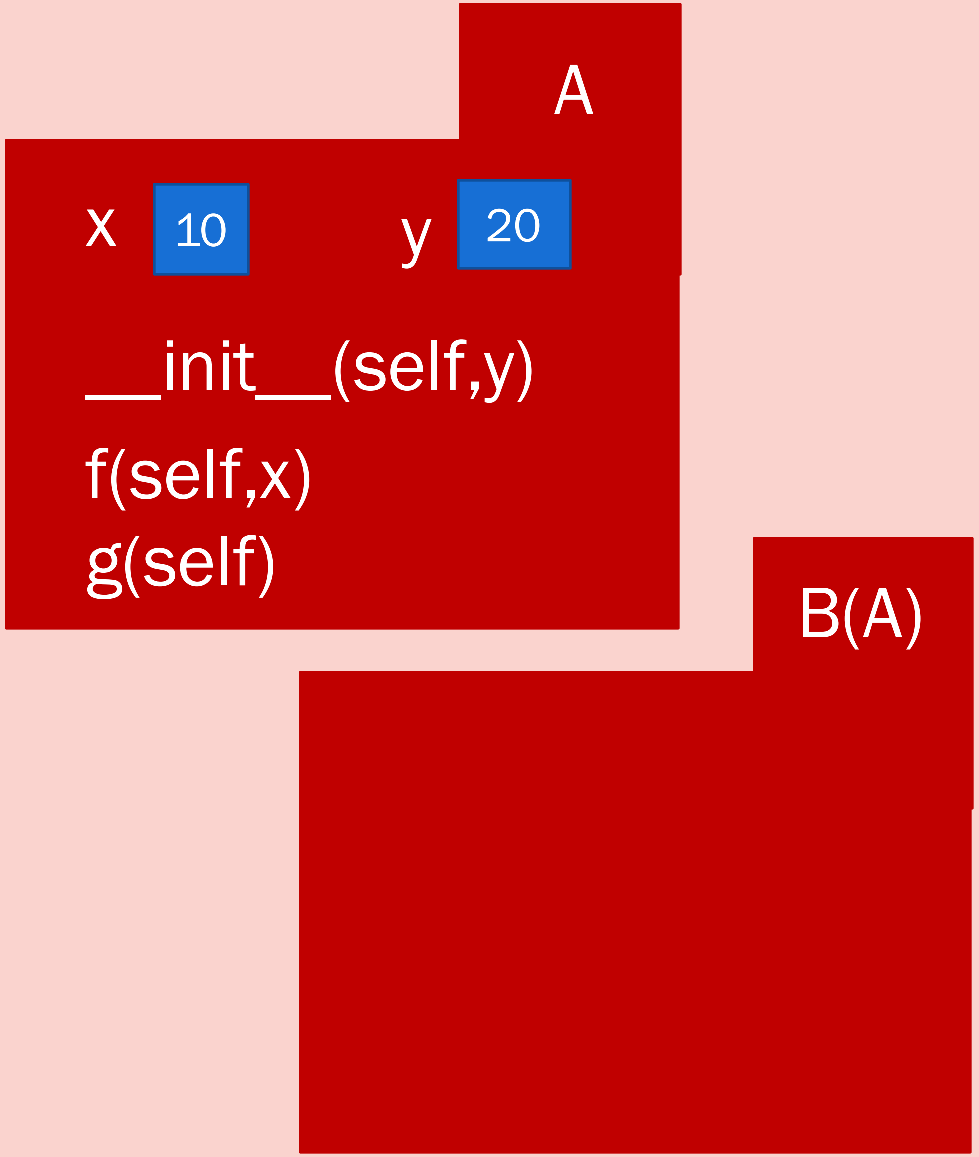


```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```

Let's do the same for B!

Add the name of the class to upper right tab.



A

x 10

y 20

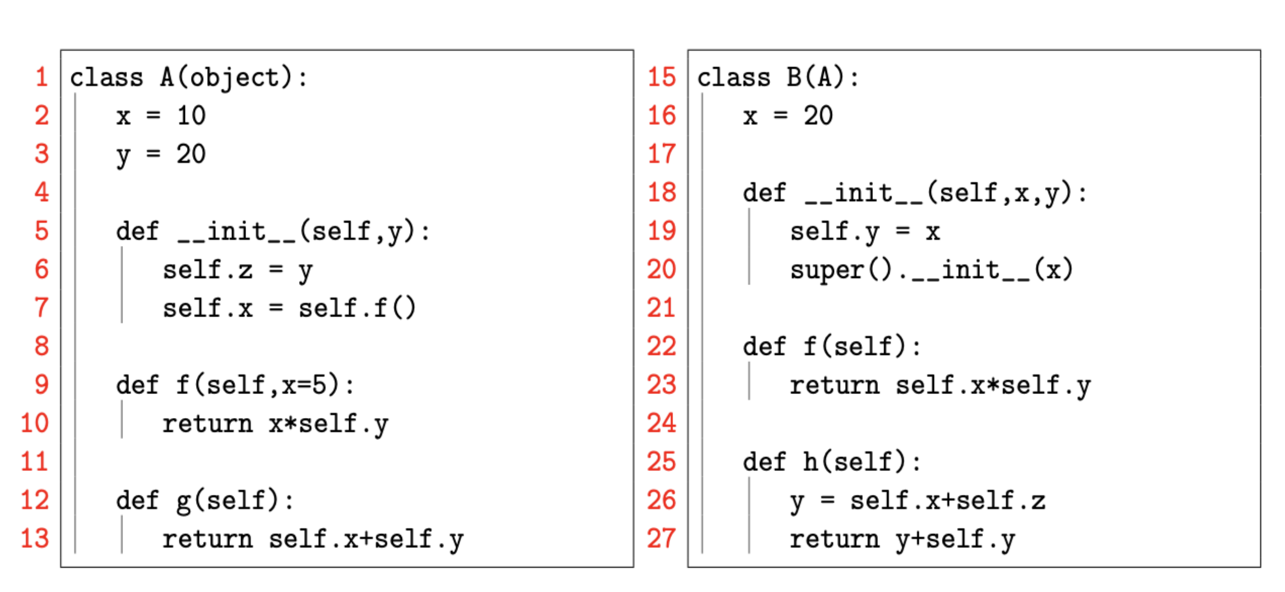
__init__(self,y)

f(self,x)

g(self)

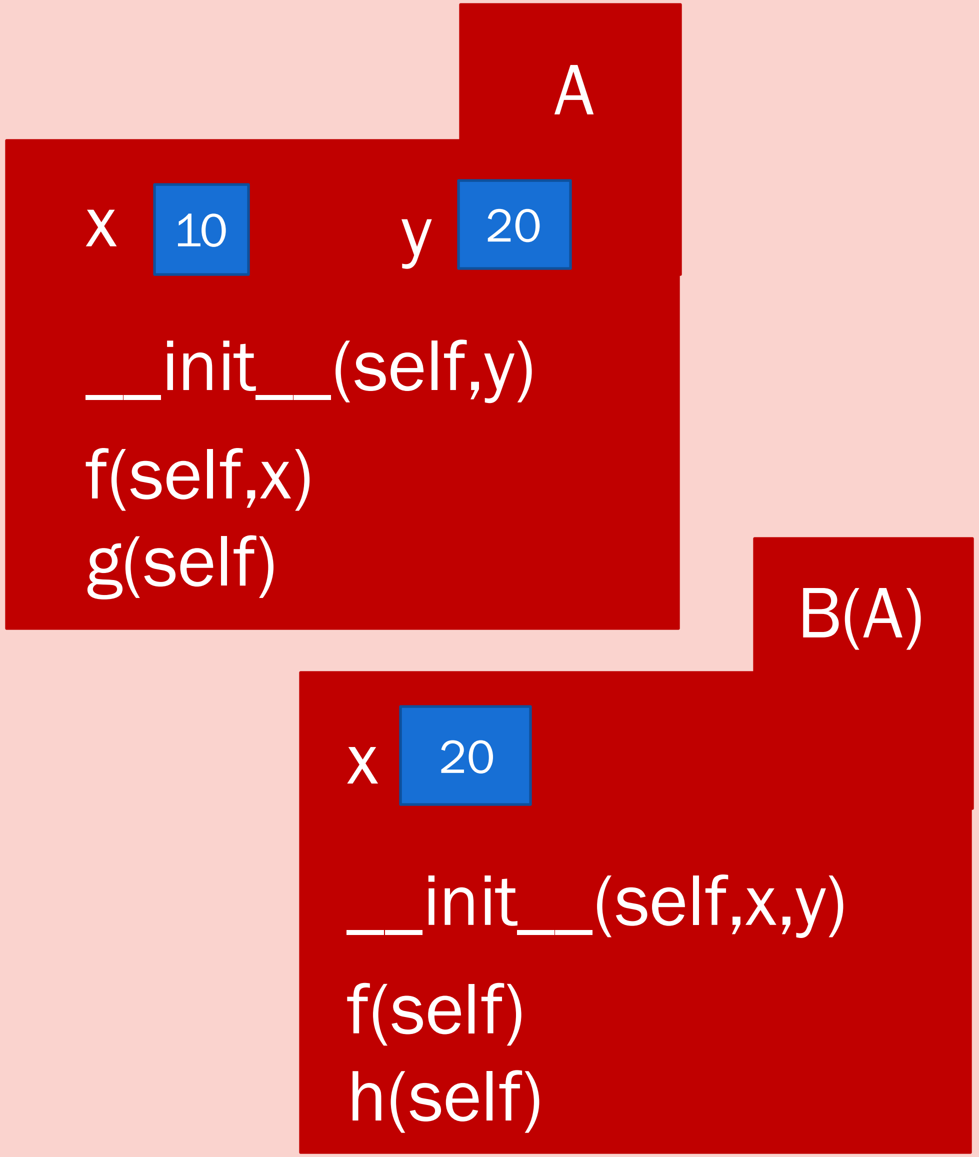
B(A)

Let's Draw This Code



```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```



A

x 10 y 20

__init__(self,y)

f(self,x)

g(self)

B(A)

x 20

__init__(self,x,y)

f(self)

h(self)

Then, add our methods and class variables!

Class vs. Object Folders

Class Folders

1. These folders are RED and have the tab in the upper RIGHT corner
2. These folders DO NOT have ID's (the name of the class goes in the tab)
3. These folders contain methods and variables.

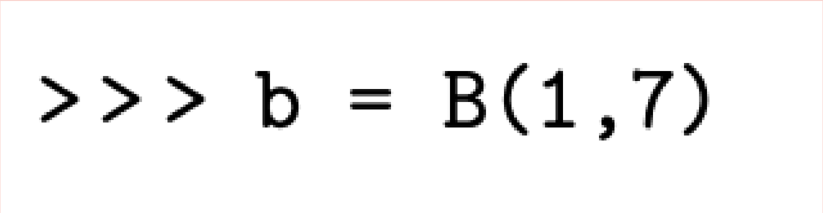
Object Folders

1. These folders are cream colored/yellow and have their tab in the upper LEFT corner
2. These folders DO have ID's and the ID goes in the tab
3. These folders only contain variables (not methods)

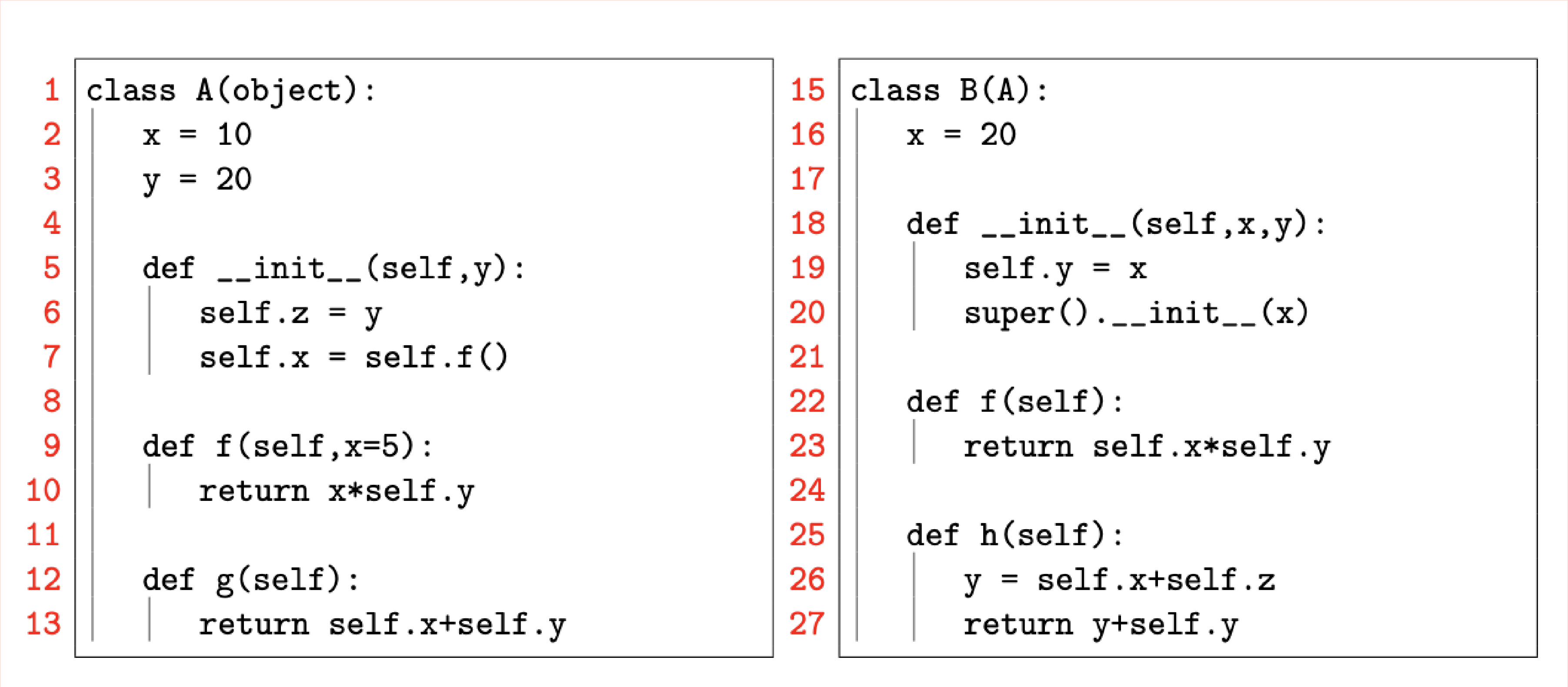
So…When Drawing a Class Folder

- First put the name of the class in the upper RIGHT corner
  - If the class has a super class, represent this by writing the name of the super class in parentheses next to the class's name
- Then, put class variable names and values in the folder.
- Finally, put method names in the folder.

Let's Draw This Call With Our Code



```
>>> b = B(1,7)
```



```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```

```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```

```
>>> b = B(1,7)
```

Global Space

Call Stack

Heap Space

Sorry the orientation is wacky; it's hard to fit this all on one slide!

```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

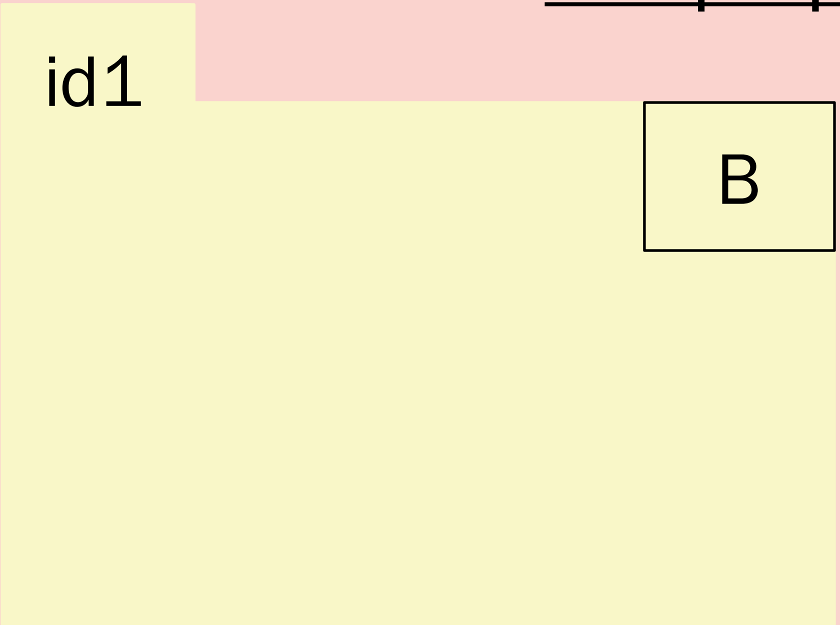
```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```

```
>>> b = B(1,7)
```

Global Space

Call Stack

Heap Space



This is a Constructor call

When Python sees one of these, it makes a folder for this object (gives it an ID in top left and puts the name of class in the right).

Then, it calls the __init__ method to populate the folder (next slide)

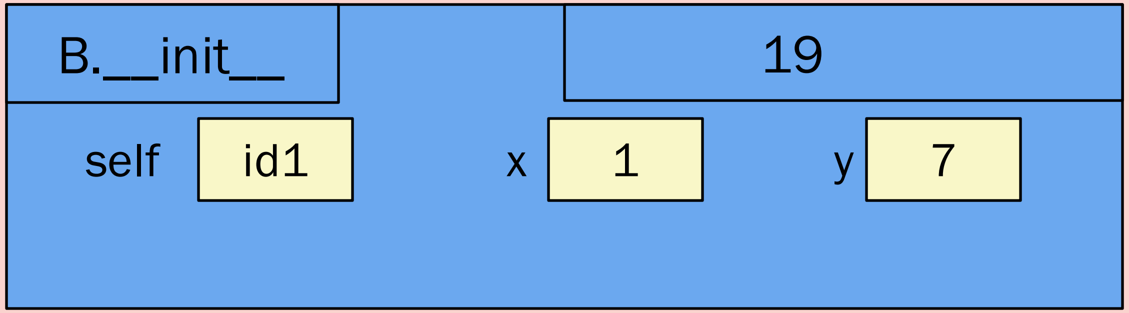
```
1  class A(object):
2      x = 10
3      y = 20
4
5      def __init__(self,y):
6          self.z = y
7          self.x = self.f()
8
9      def f(self,x=5):
10         return x*self.y
11
12     def g(self):
13         return self.x+self.y
```

```
15 class B(A):
16     x = 20
17
18     def __init__(self,x,y):
19         self.y = x
20         super().__init__(x)
21
22     def f(self):
23         return self.x*self.y
24
25     def h(self):
26         y = self.x+self.z
27         return y+self.y
```

```
>>> b = B(1,7)
```

Global Space

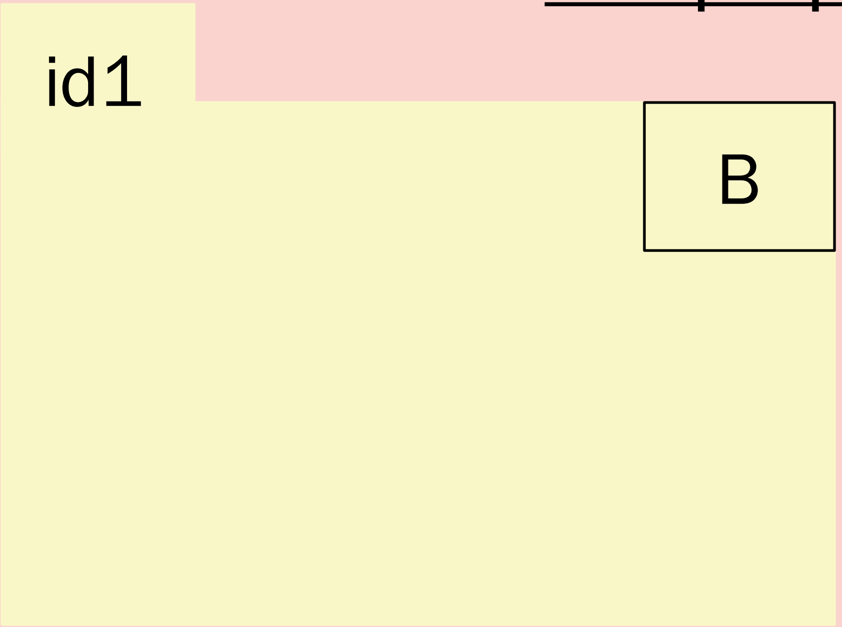
Call Stack



B.__init__ 19

self id1 x 1 y 7

Heap Space



id1

B

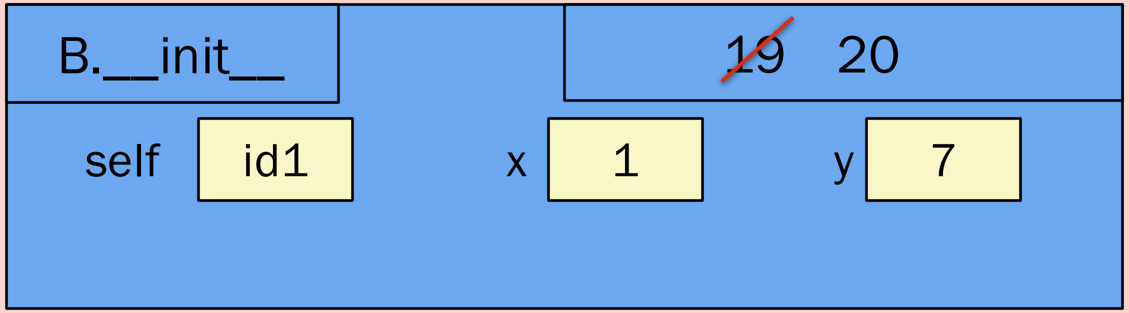
```
 1  class A(object):
 2      x = 10
 3      y = 20
 4
 5      def __init__(self,y):
 6          self.z = y
 7          self.x = self.f()
 8
 9      def f(self,x=5):
10          return x*self.y
11
12      def g(self):
13          return self.x+self.y
```

```
15  class B(A):
16      x = 20
17
18      def __init__(self,x,y):
19          self.y = x
20          super().__init__(x)
21
22      def f(self):
23          return self.x*self.y
24
25      def h(self):
26          y = self.x+self.z
27          return y+self.y
```

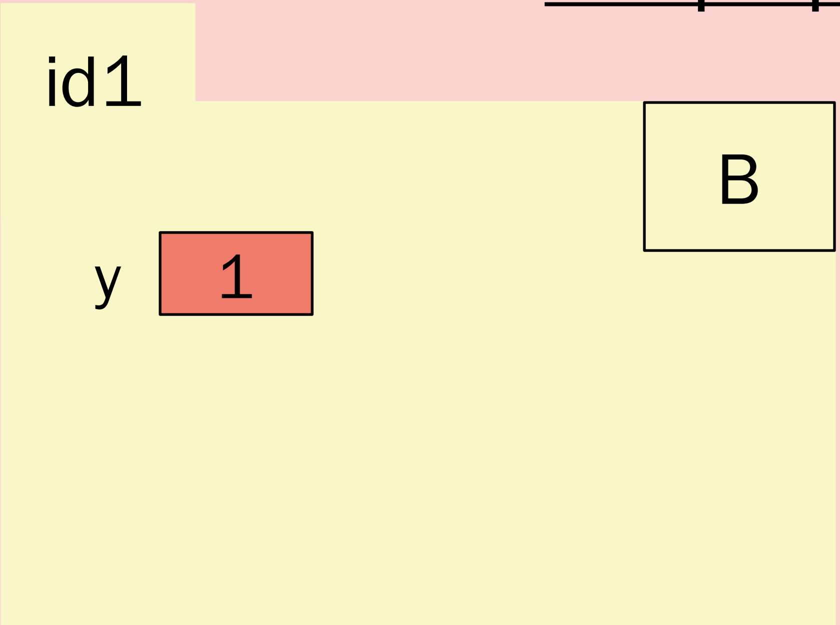
```
>>> b = B(1,7)
```

Global Space

Call Stack



Heap Space



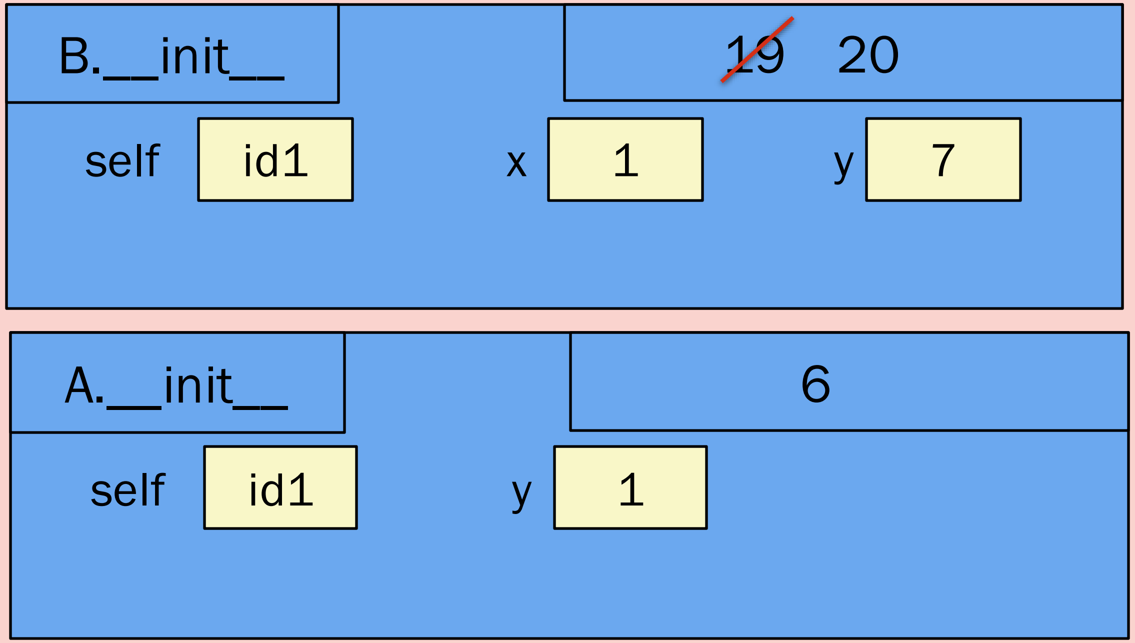
```
1   class A(object):
2       x = 10
3       y = 20
4
5       def __init__(self,y):
6           self.z = y
7           self.x = self.f()
8
9       def f(self,x=5):
10          return x*self.y
11
12      def g(self):
13          return self.x+self.y
```

```
15  class B(A):
16      x = 20
17
18      def __init__(self,x,y):
19          self.y = x
20          super().__init__(x)
21
22      def f(self):
23          return self.x*self.y
24
25      def h(self):
26          y = self.x+self.z
27          return y+self.y
```

```
>>> b = B(1,7)
```

Global Space

Call Stack



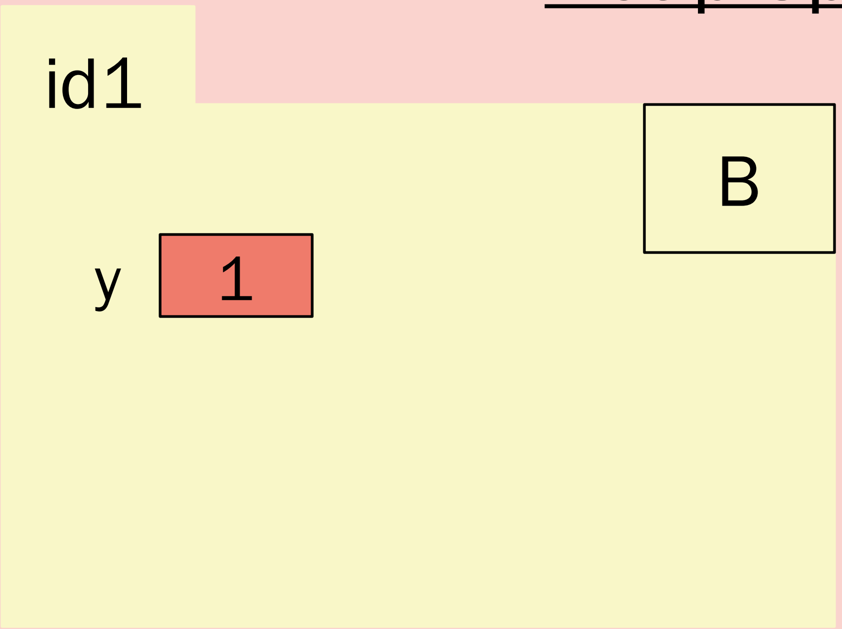
B.__init__ ~~19~~ 20

self id1 x 1 y 7

A.__init__ 6

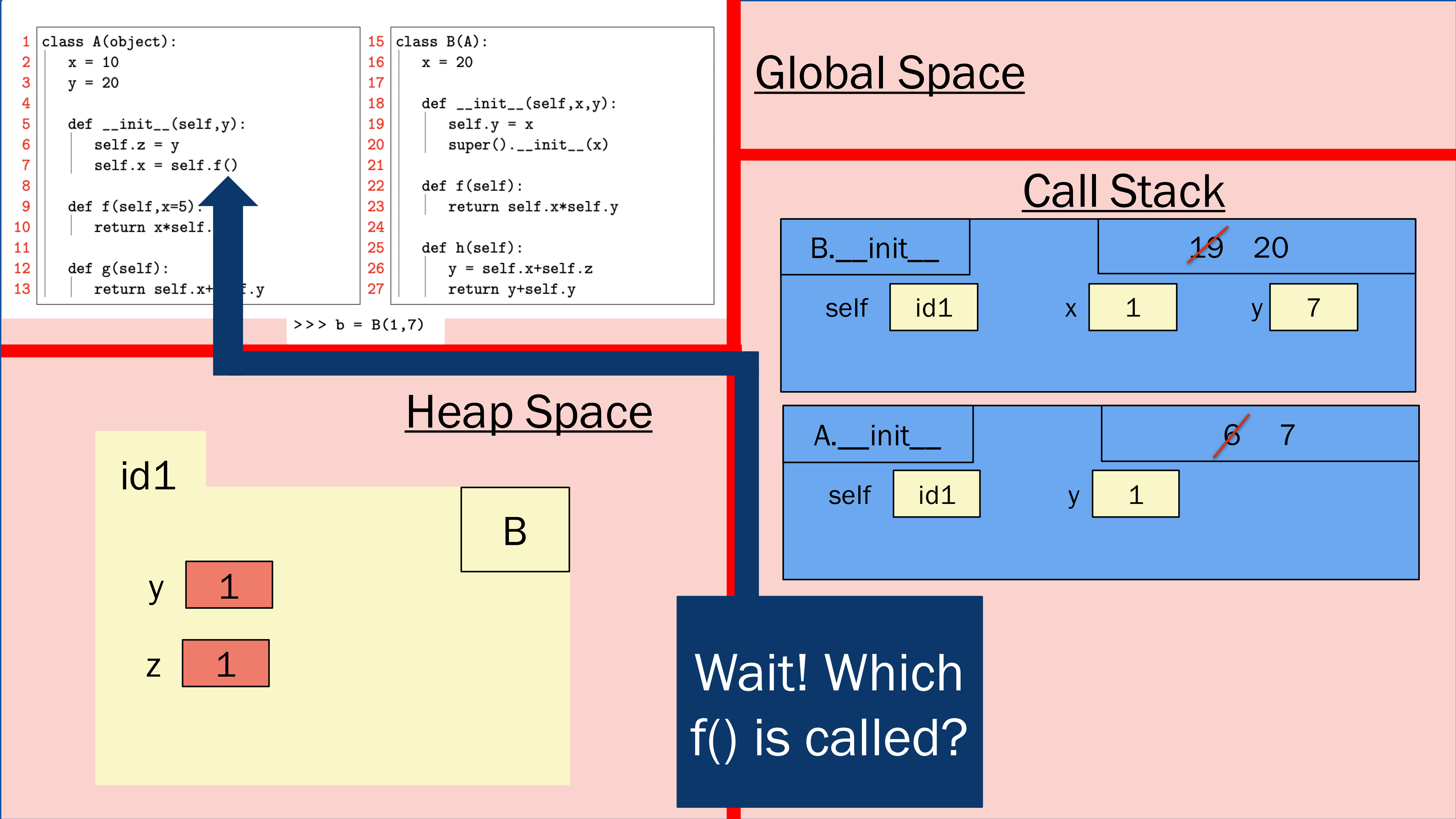
self id1 y 1

Heap Space



id1 B

y 1



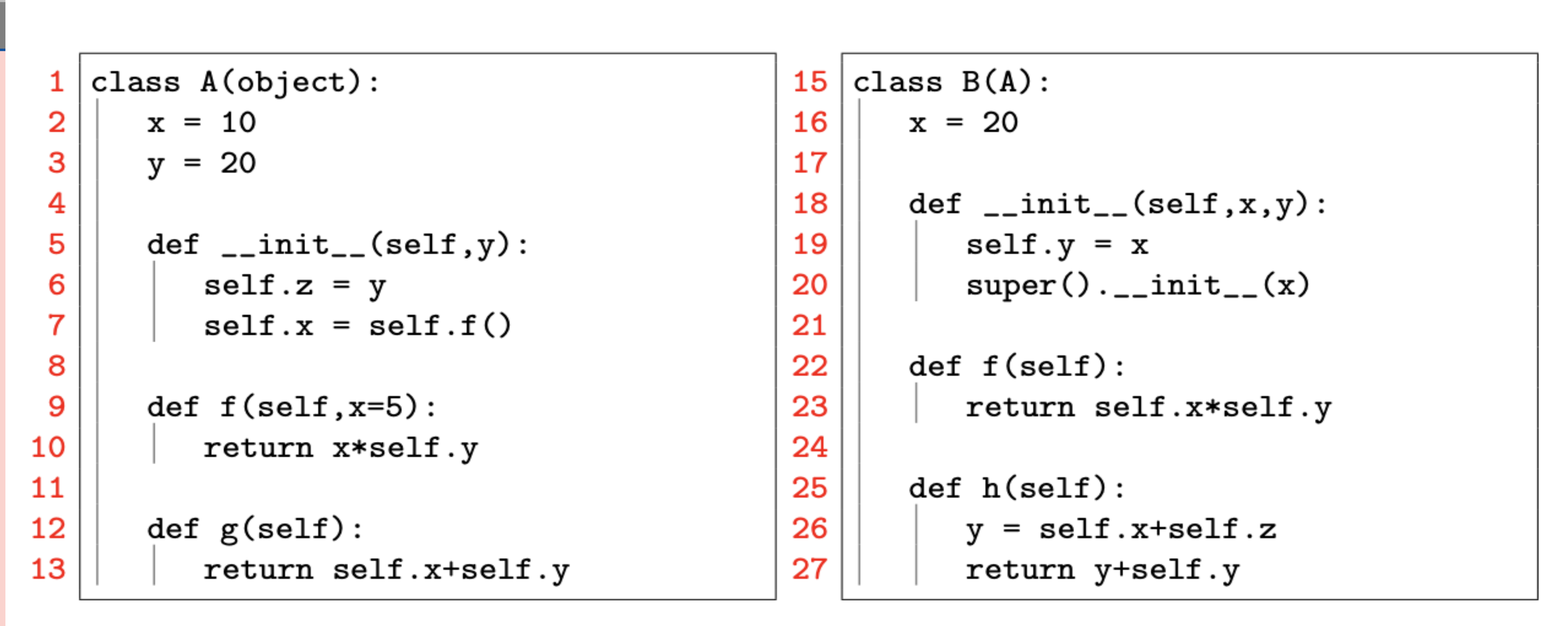
```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.

    def g(self):
        return self.x+    f.y
```

```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```

```
>>> b = B(1,7)
```

Wait! Which f() is called?

Overriding



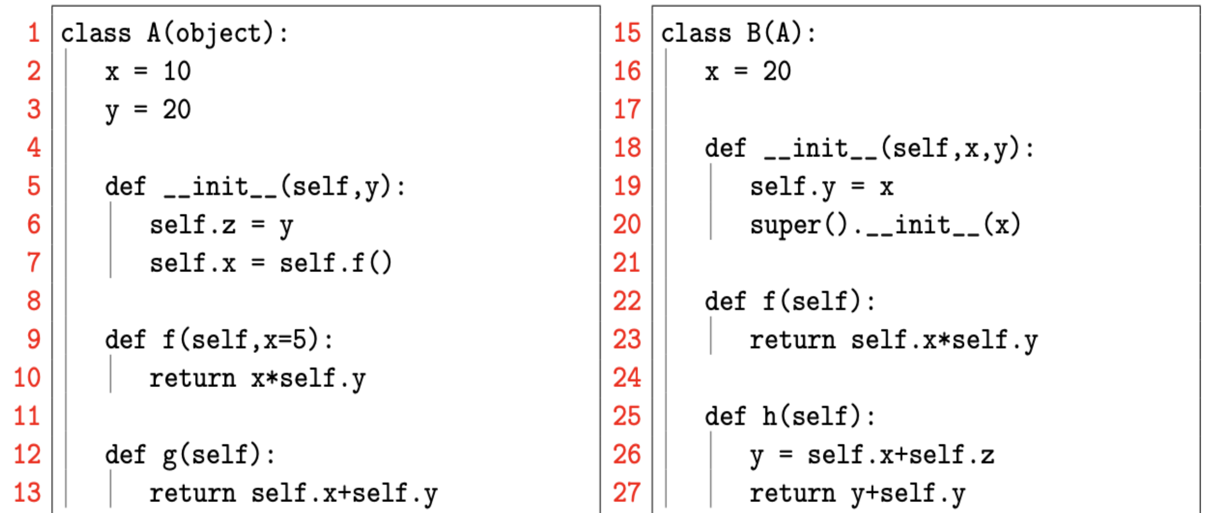
```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```

- Definition – when a method exists in a superclass and we define it in the subclass as well.
- So, f() is overridden in given code because it exists in A, and we define a different version in B
- Can be done with other methods, too (like __init__, __eq__, etc.)
- Which version is called though?

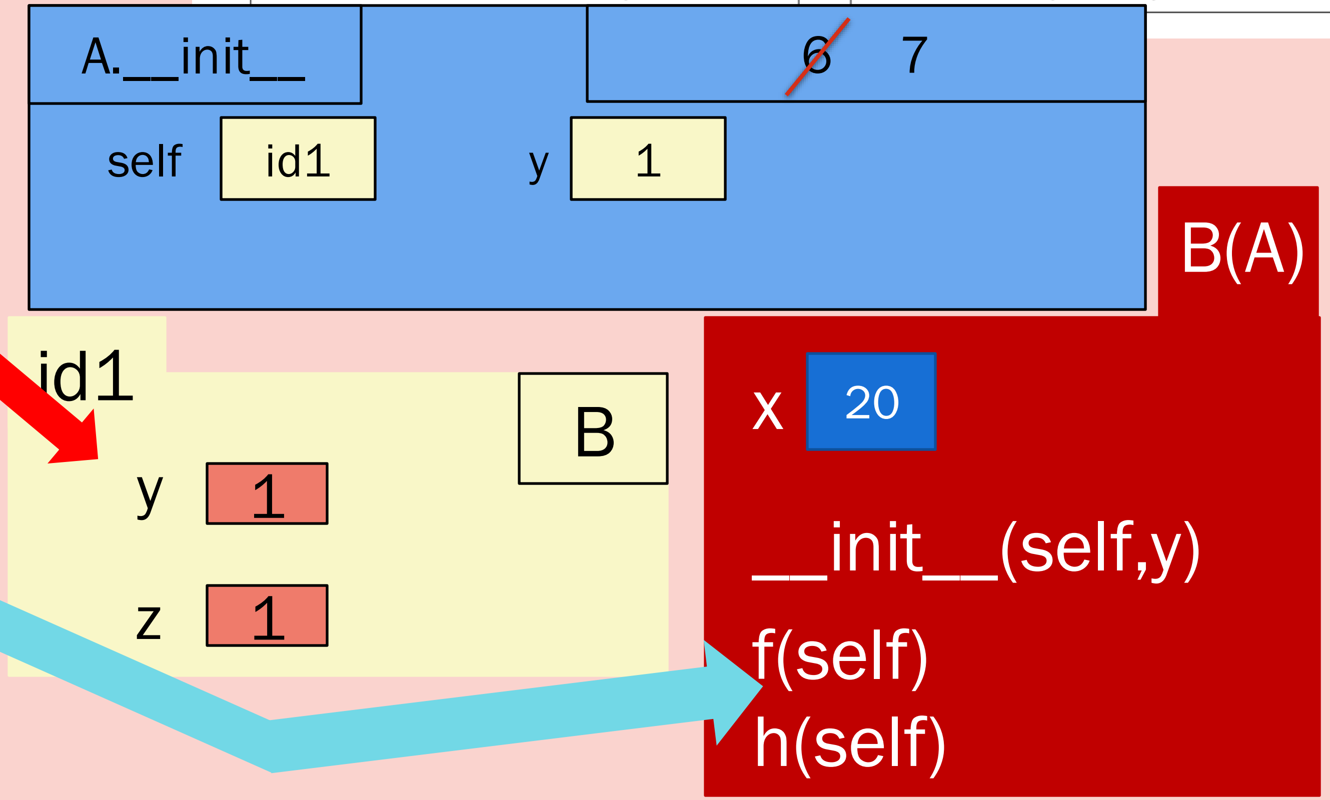
Overriding

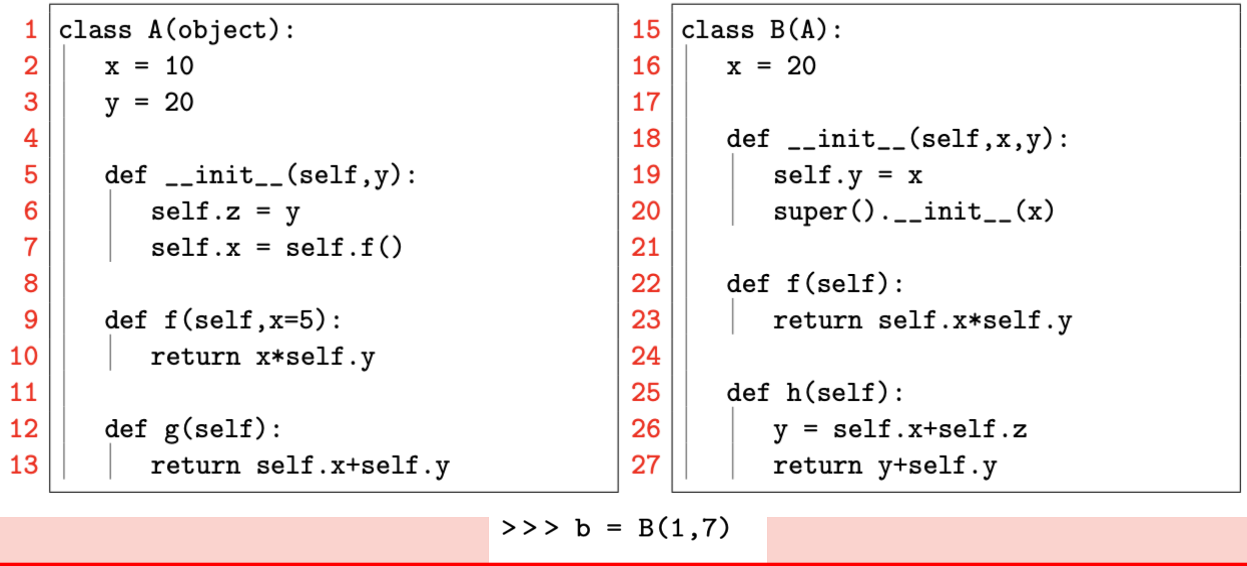
- The one in B!
- What Python does:
  - self is id1, so does id1 contain a method called f()?
    - No!
  - Next, check id1's class (B). Does it contain a method f()?
    - Yes! Call that one!
  - So, Python doesn't even care that it exists in A as well



```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```





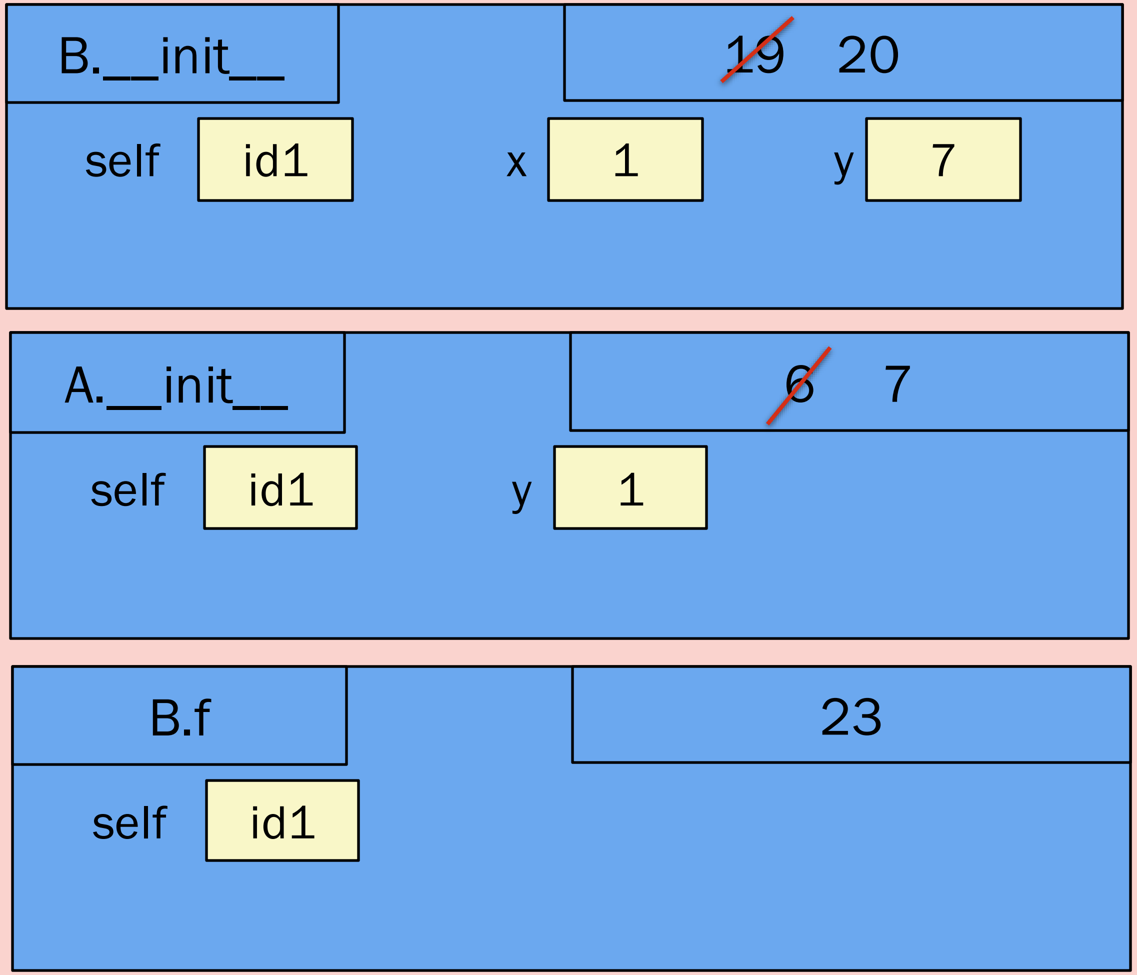
```
1  class A(object):
2      x = 10
3      y = 20
4
5      def __init__(self,y):
6          self.z = y
7          self.x = self.f()
8
9      def f(self,x=5):
10         return x*self.y
11
12     def g(self):
13         return self.x+self.y
```

```
15 class B(A):
16     x = 20
17
18     def __init__(self,x,y):
19         self.y = x
20         super().__init__(x)
21
22     def f(self):
23         return self.x*self.y
24
25     def h(self):
26         y = self.x+self.z
27         return y+self.y
```

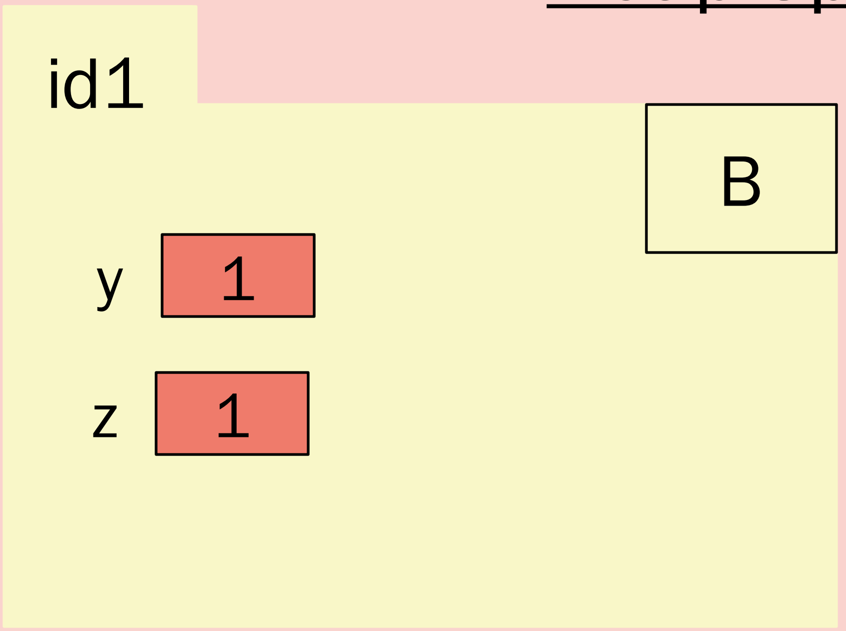
```
>>> b = B(1,7)
```

Global Space

Call Stack



Heap Space



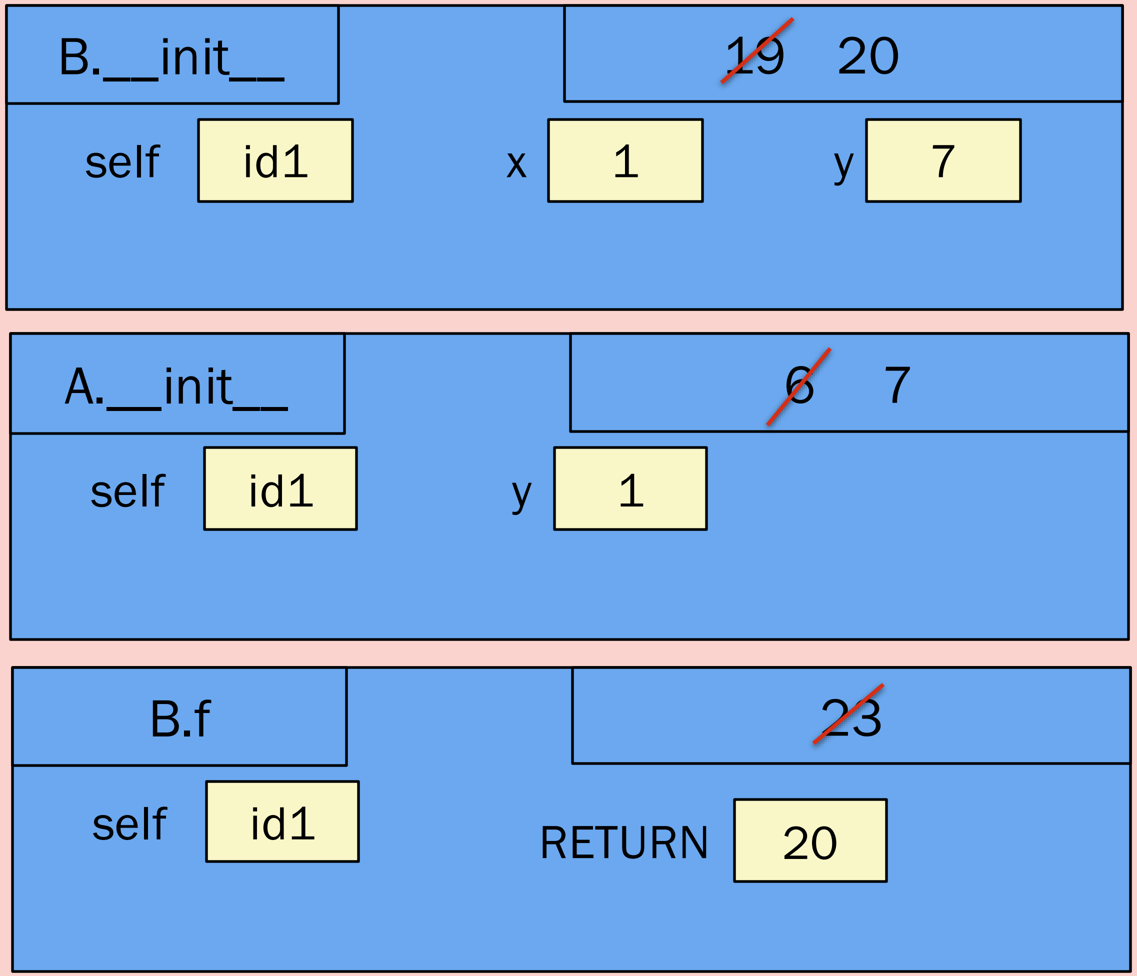
```
class A(object):
    x = 10
    y = 20

    def __init__(self,y):
        self.z = y
        self.x = self.f()

    def f(self,x=5):
        return x*self.y

    def g(self):
        return self.x+self.y
```

```
class B(A):
    x = 20

    def __init__(self,x,y):
        self.y = x
        super().__init__(x)

    def f(self):
        return self.x*self.y

    def h(self):
        y = self.x+self.z
        return y+self.y
```

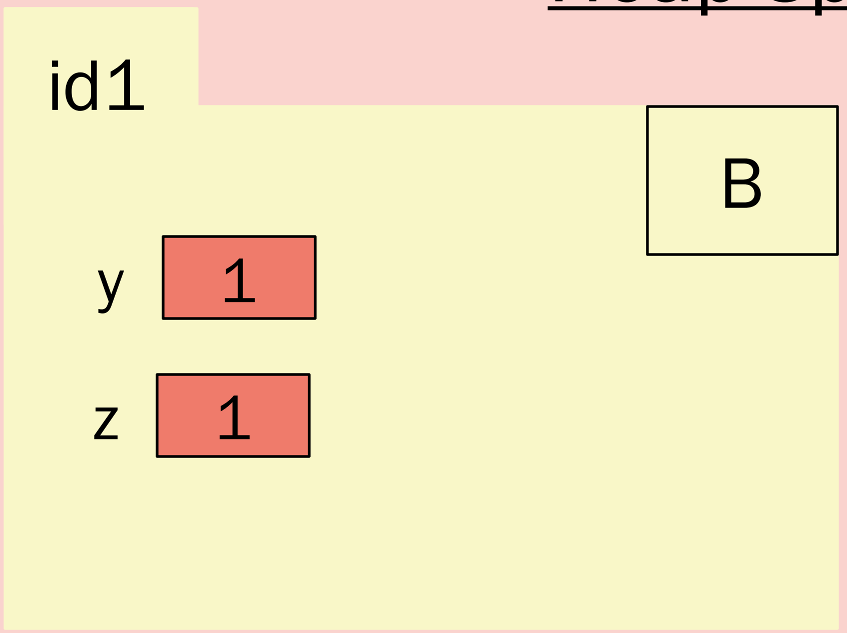
```
>>> b = B(1,7)
```

Global Space

Call Stack



Heap Space



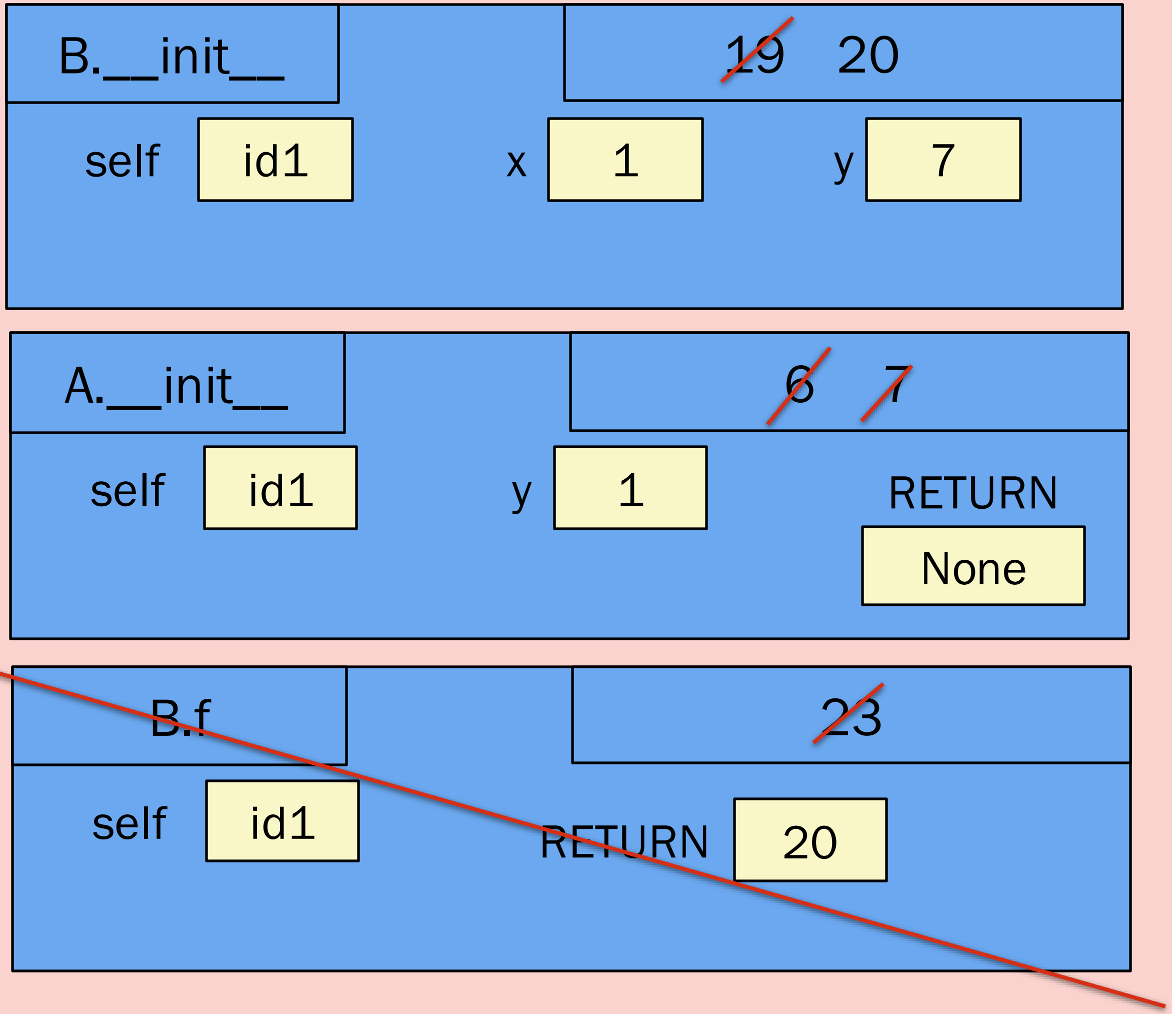
```
 1  class A(object):
 2      x = 10
 3      y = 20
 4
 5      def __init__(self,y):
 6          self.z = y
 7          self.x = self.f()
 8
 9      def f(self,x=5):
10          return x*self.y
11
12      def g(self):
13          return self.x+self.y
```

```
15  class B(A):
16      x = 20
17
18      def __init__(self,x,y):
19          self.y = x
20          super().__init__(x)
21
22      def f(self):
23          return self.x*self.y
24
25      def h(self):
26          y = self.x+self.z
27          return y+self.y
```

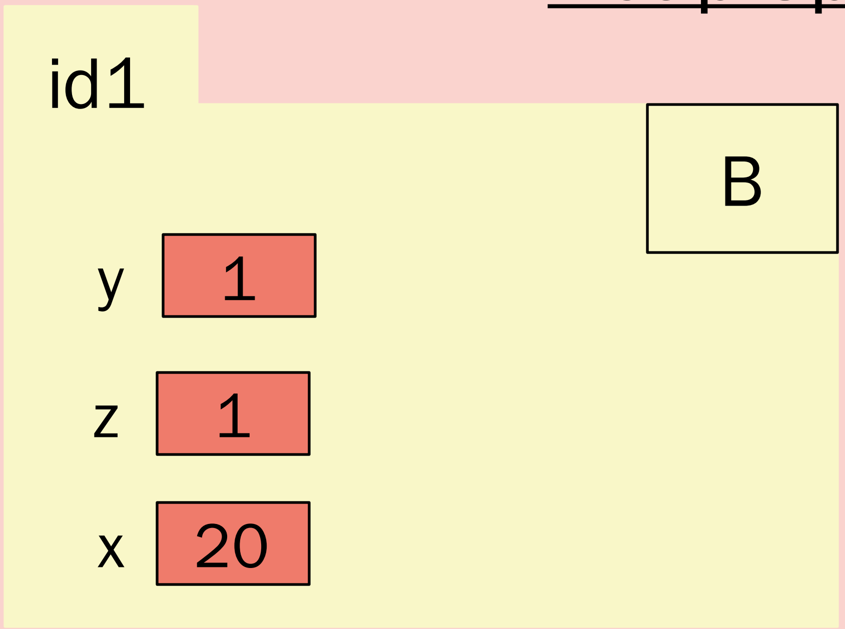
```
>>> b = B(1,7)
```

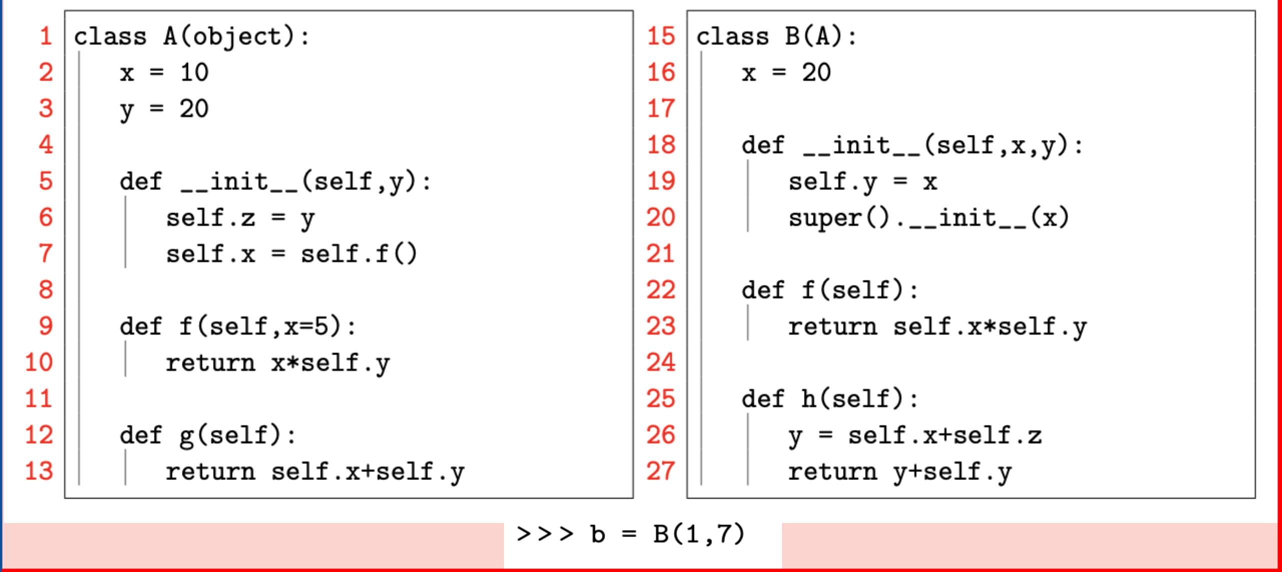
Global Space

Call Stack



Heap Space





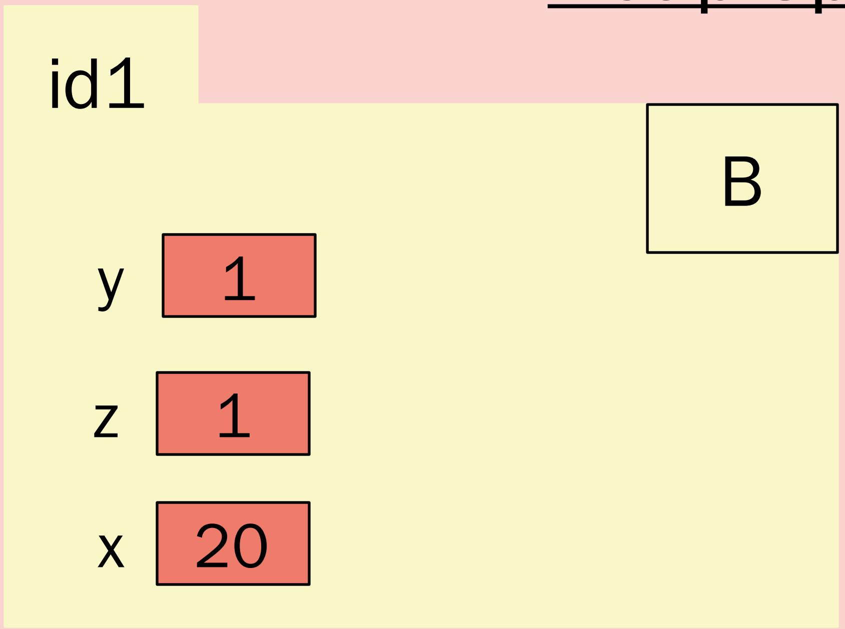
```
1  class A(object):
2      x = 10
3      y = 20
4
5      def __init__(self,y):
6          self.z = y
7          self.x = self.f()
8
9      def f(self,x=5):
10         return x*self.y
11
12     def g(self):
13         return self.x+self.y
```

```
15 class B(A):
16     x = 20
17
18     def __init__(self,x,y):
19         self.y = x
20         super().__init__(x)
21
22     def f(self):
23         return self.x*self.y
24
25     def h(self):
26         y = self.x+self.z
27         return y+self.y
```

```
>>> b = B(1,7)
```

Global Space

Heap Space



id1

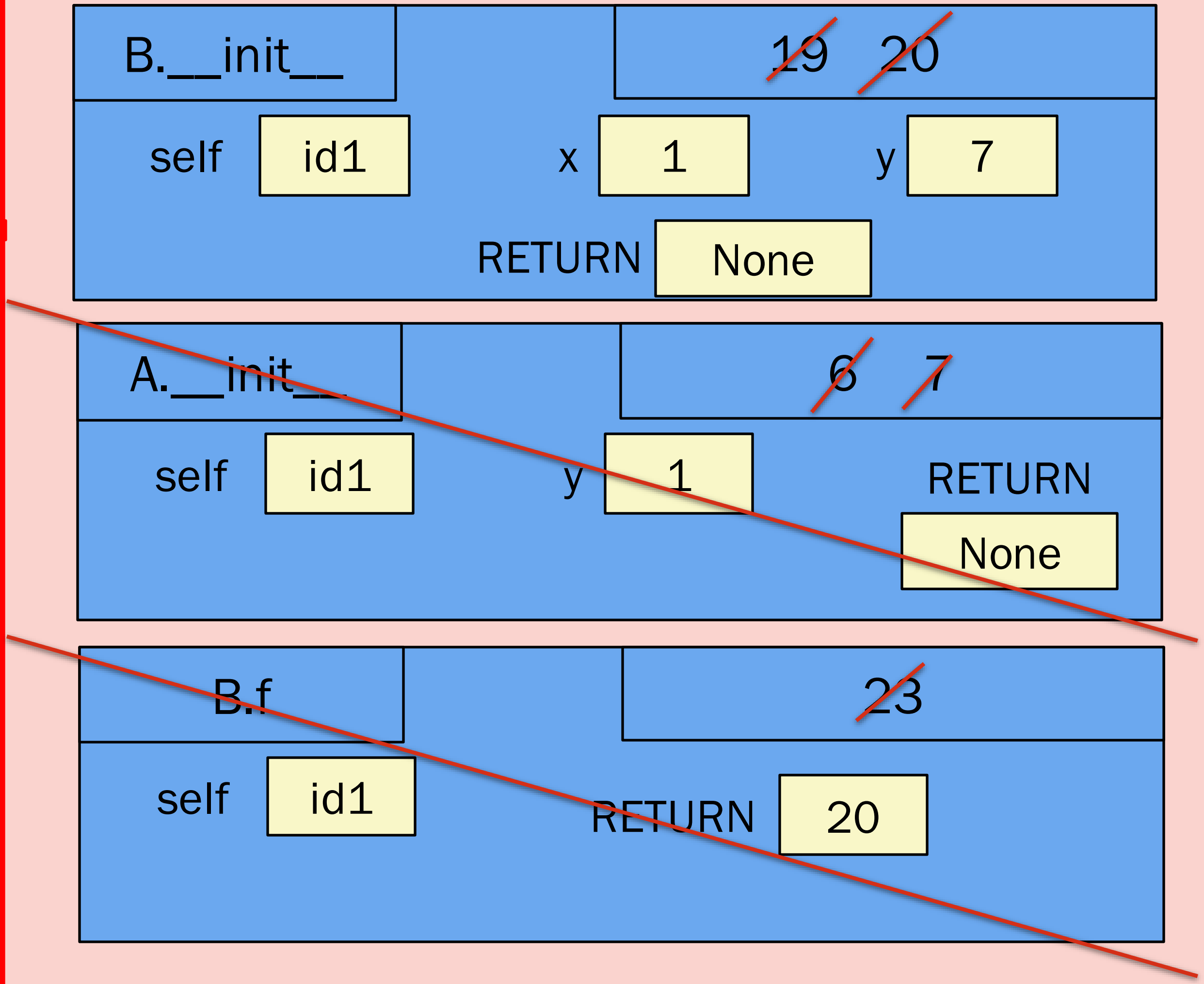
B

y 1

z 1

x 20

Call Stack



B.__init__ ~~19~~ ~~20~~

self id1 x 1 y 7

RETURN None

~~A.__init__~~ ~~6~~ ~~7~~

~~self id1 y 1 RETURN None~~

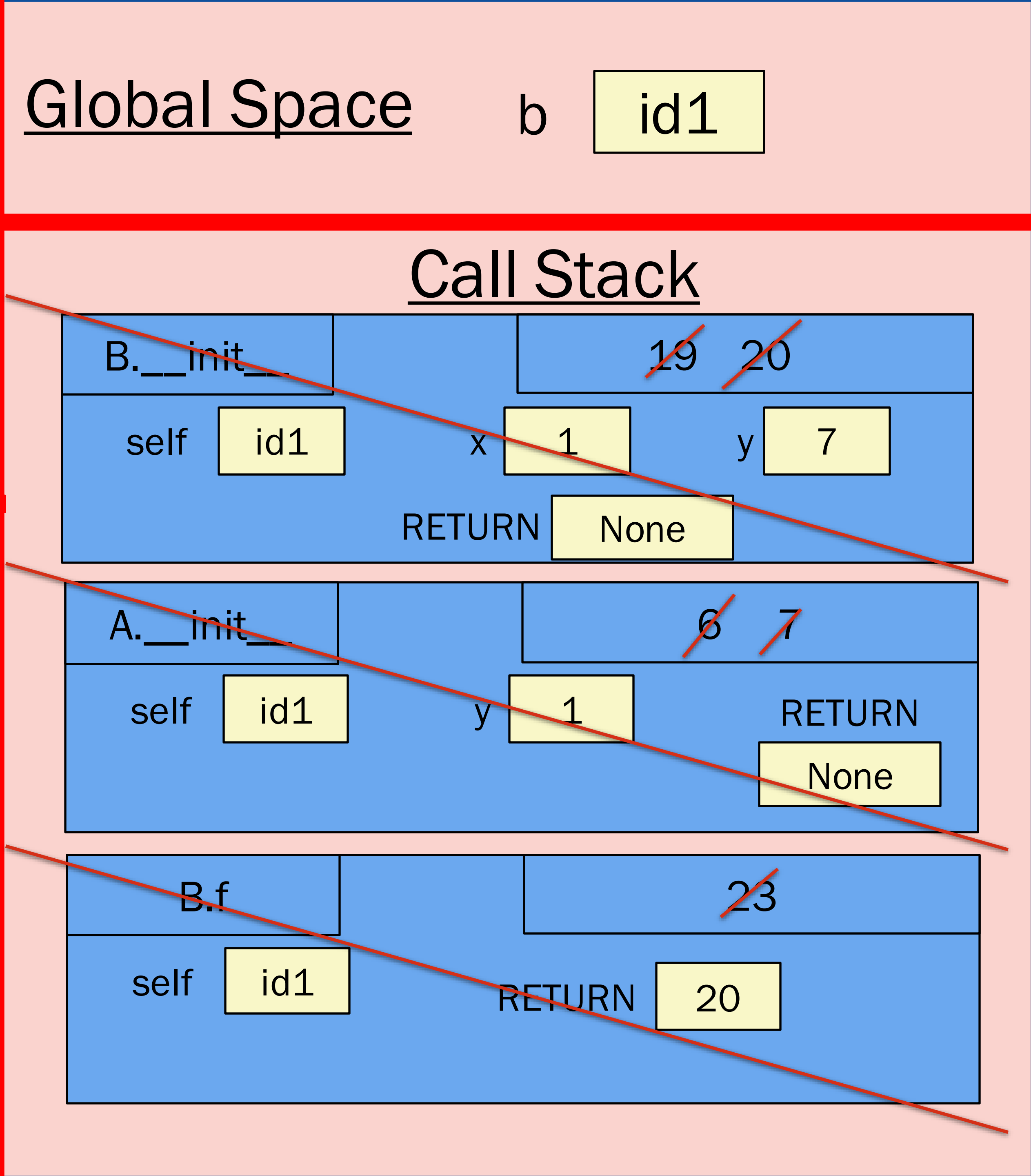
~~B.f~~ ~~23~~

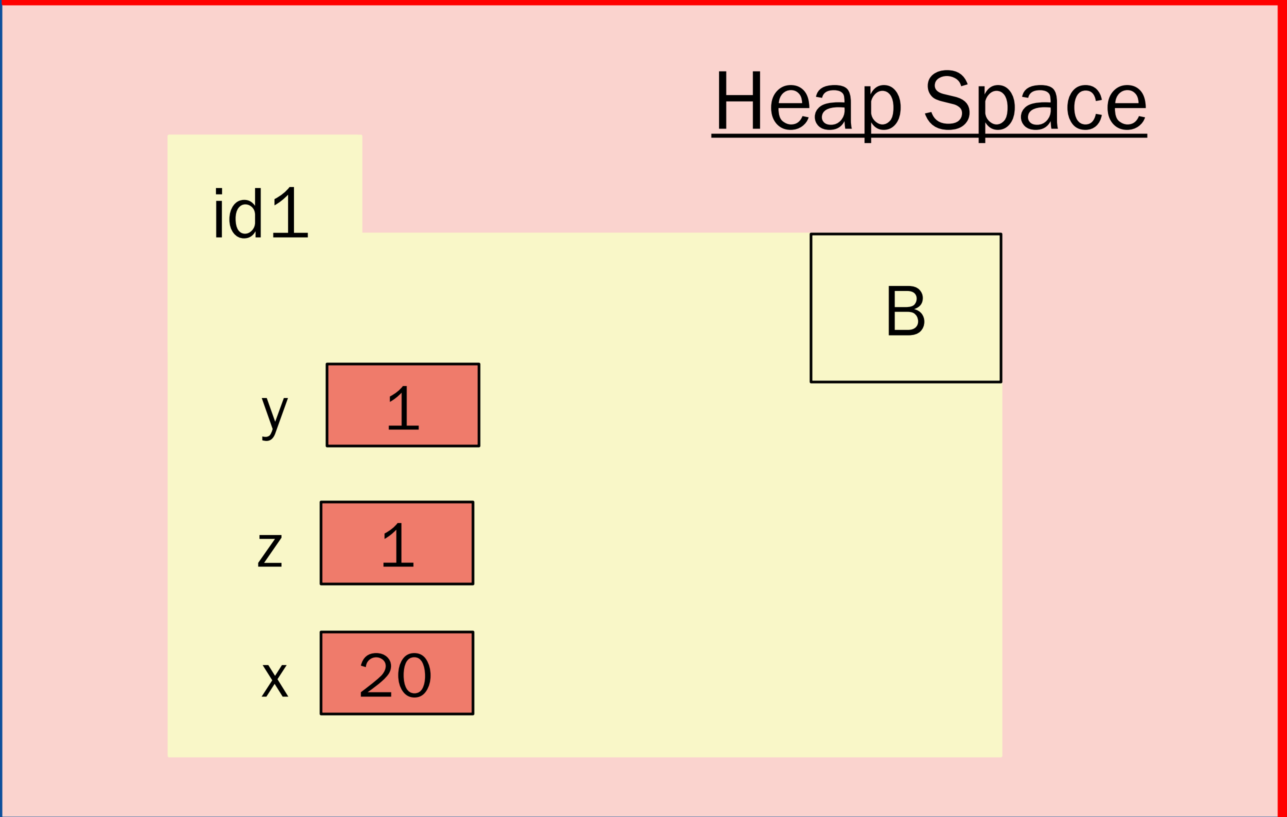
~~self id1 RETURN 20~~

```
1   class A(object):
2       x = 10
3       y = 20
4
5       def __init__(self,y):
6           self.z = y
7           self.x = self.f()
8
9       def f(self,x=5):
10          return x*self.y
11
12      def g(self):
13          return self.x+self.y
```

```
15  class B(A):
16      x = 20
17
18      def __init__(self,x,y):
19          self.y = x
20          super().__init__(x)
21
22      def f(self):
23          return self.x*self.y
24
25      def h(self):
26          y = self.x+self.z
27          return y+self.y
```

```
>>> b = B(1,7)
```





Summary: Constructors

- Constructor: B(1,7) from last slides
    - So, name of class followed by parentheses (and then arguments if necessary)
- How it works:
    - Create a folder and give it an ID (put name of class in right side of folder)
    - Call __init__ to populate folder, passing the ID of the folder as self
    - When __init__ finishes, "return" the value of the folder's id
        - Which is why id1 was stored in b at the end of our last slide

CODING PRACTICE

Let's code together!

Coding Scenario

- We're making a Cornell in Python!
- There are three classes we will code:
    - Cornellian
    - Student (which extend Cornellian)
    - Professor (which extends Cornellian)
- We're going to code the __init__ method for Cornellian, Student, and Professor
- We will also code the __eq__ method in Cornellian
- Finally, we will code a method that will allow students to choose their major!

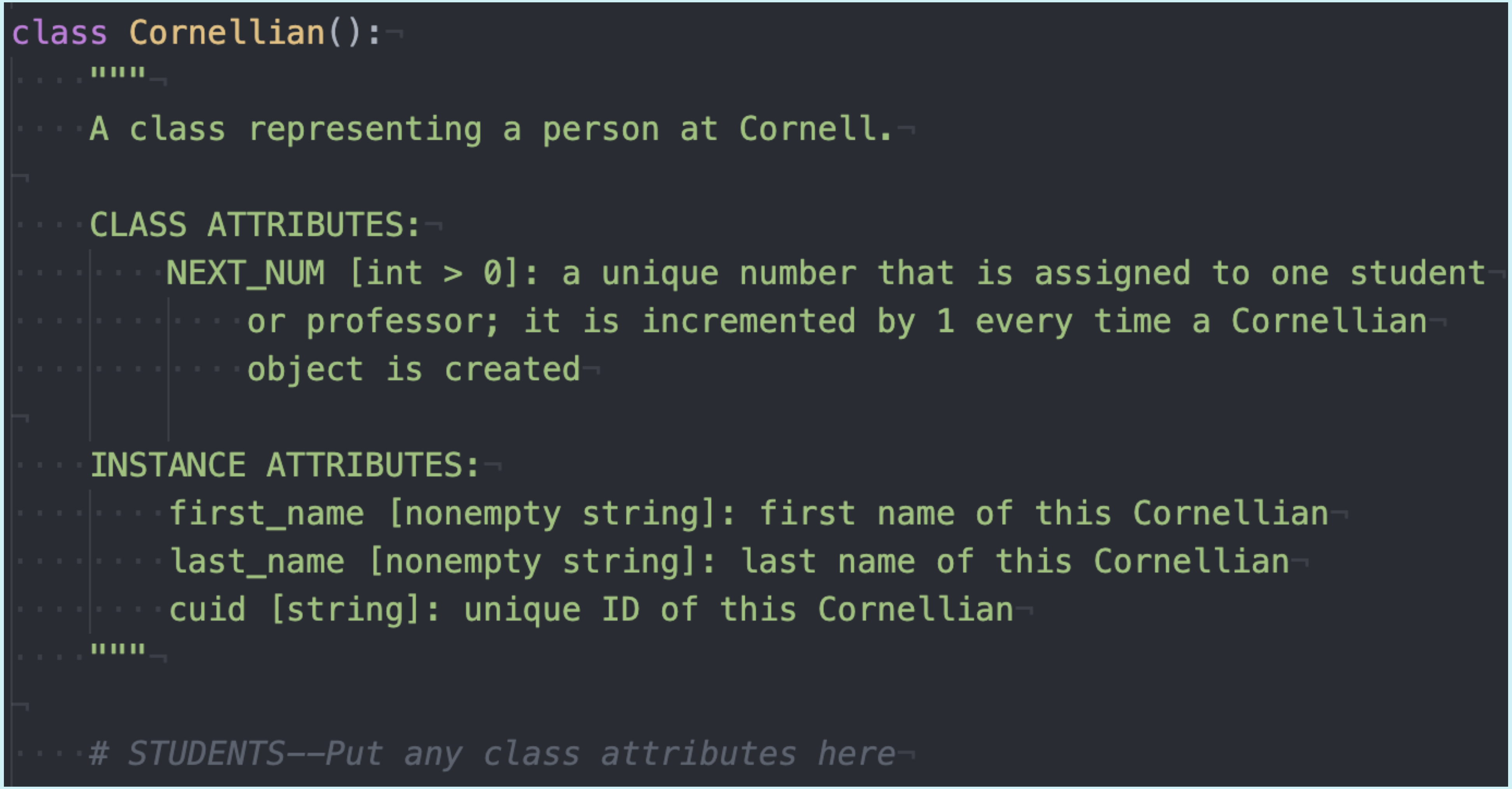
Coding Scenario

Cornellian	
Class Attributes	NEXT_NUM (int > 0)
Instance Attributes	first_name (nonempty string) last_name (nonempty string) cuid (nonempty string)
Methods	__init__ __eq__

Professor	
Class Attributes	departments (list of strings all_of_em (list of Professor objects
Instance Attributes	first_name (nonempty string) last_name (nonempty string) cuid (nonempty string) department (string) advisees (list of Student objecst)
Methods	__init__ chooseMajor

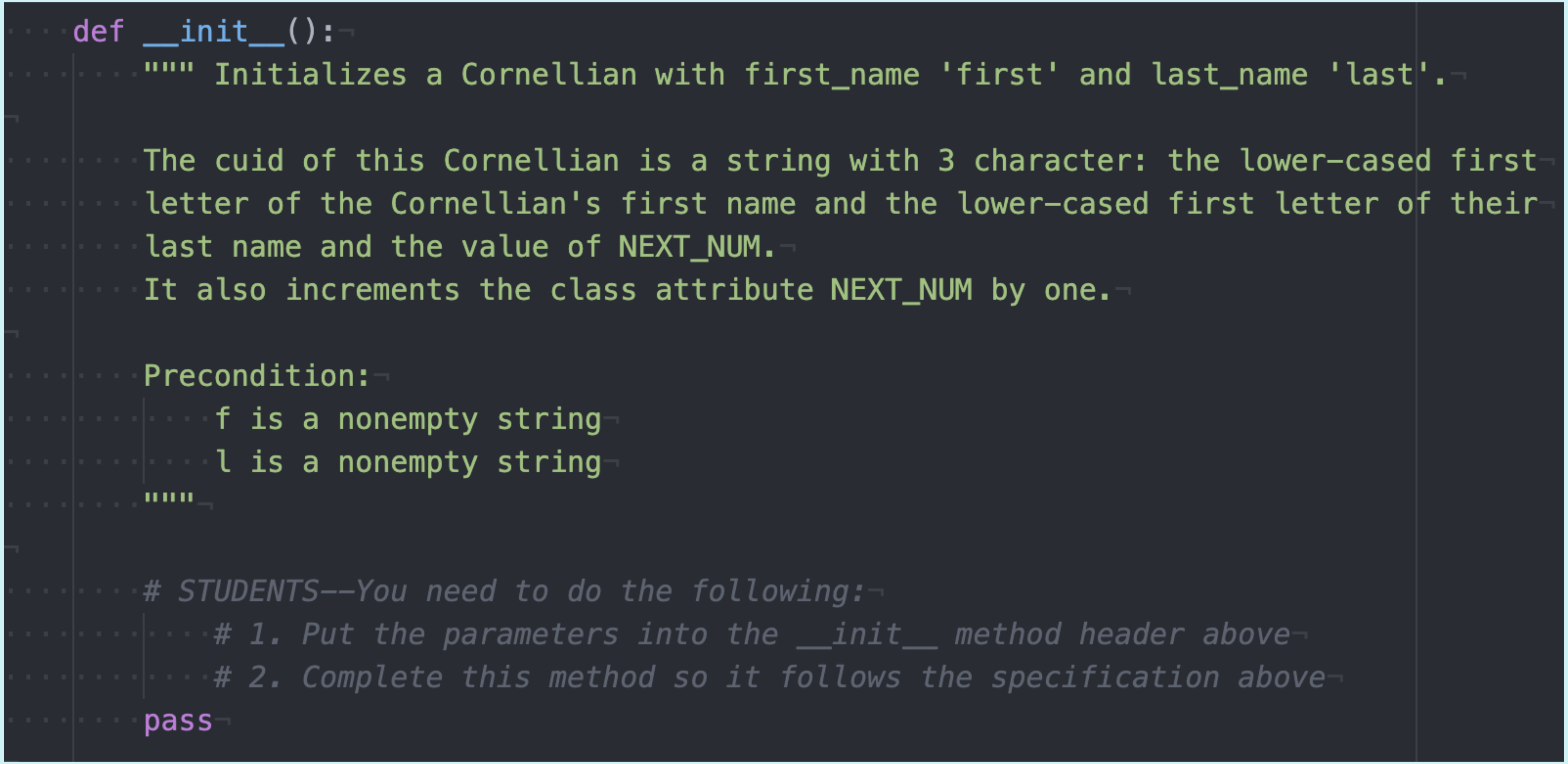
Student	
Class Attributes	n/a
Instance Attributes	first_name (nonempty string) last_name (nonempty string) cuid (nonempty string) college (string) major (string) advisor (Professor object)
Methods	__init__ chooseMajor

Class Attribute(s) for Cornellian



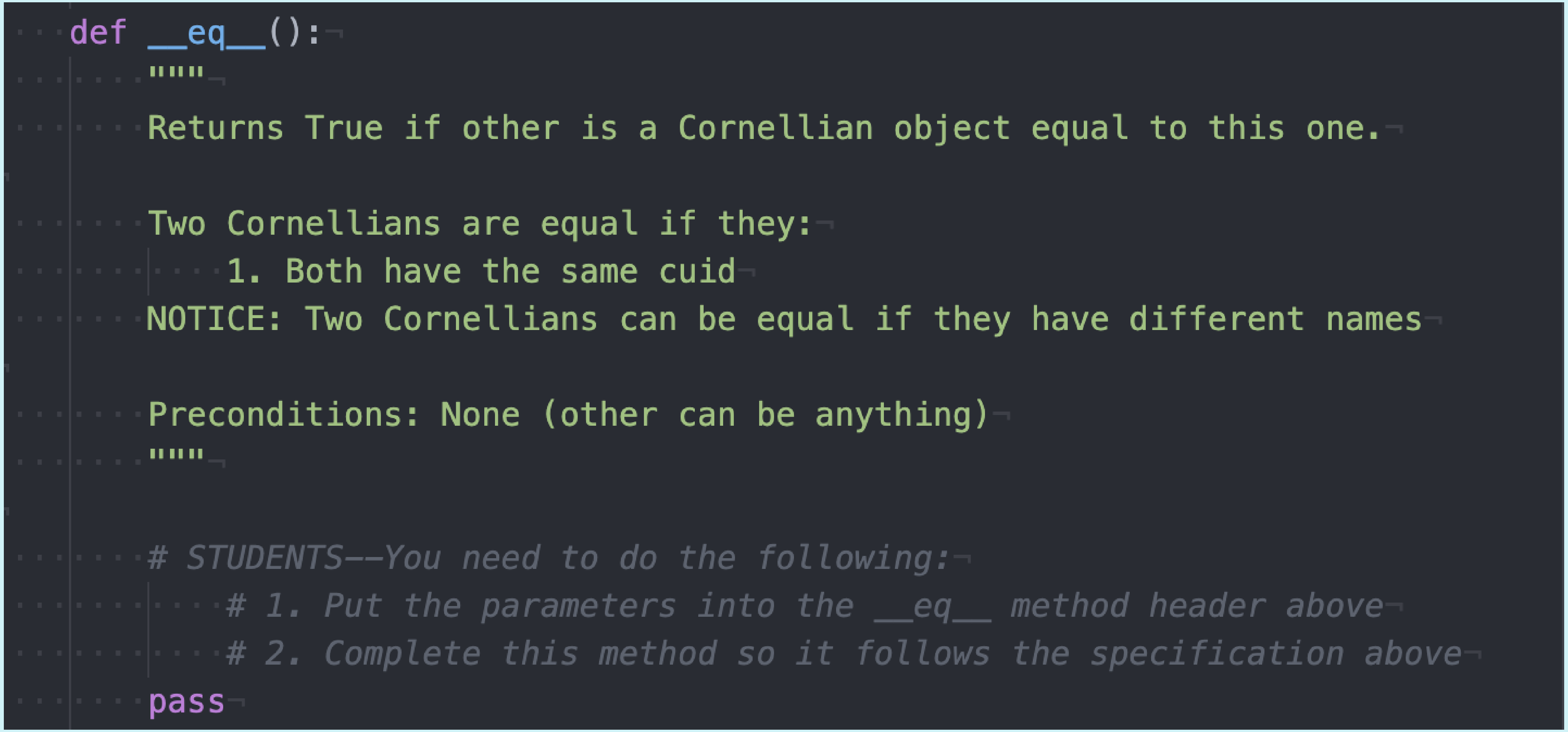
```
class Cornellian():
    """
    A class representing a person at Cornell.

    CLASS ATTRIBUTES:
        NEXT_NUM [int > 0]: a unique number that is assigned to one student
            or professor; it is incremented by 1 every time a Cornellian
            object is created

    INSTANCE ATTRIBUTES:
        first_name [nonempty string]: first name of this Cornellian
        last_name [nonempty string]: last name of this Cornellian
        cuid [string]: unique ID of this Cornellian
    """

    # STUDENTS--Put any class attributes here
```

__init__ for Cornellian



```python
    def __init__():
        """ Initializes a Cornellian with first_name 'first' and last_name 'last'.

        The cuid of this Cornellian is a string with 3 character: the lower-cased first
        letter of the Cornellian's first name and the lower-cased first letter of their
        last name and the value of NEXT_NUM.
        It also increments the class attribute NEXT_NUM by one.

        Precondition:
            f is a nonempty string
            l is a nonempty string
        """

        # STUDENTS--You need to do the following:
            # 1. Put the parameters into the __init__ method header above
            # 2. Complete this method so it follows the specification above
        pass
```

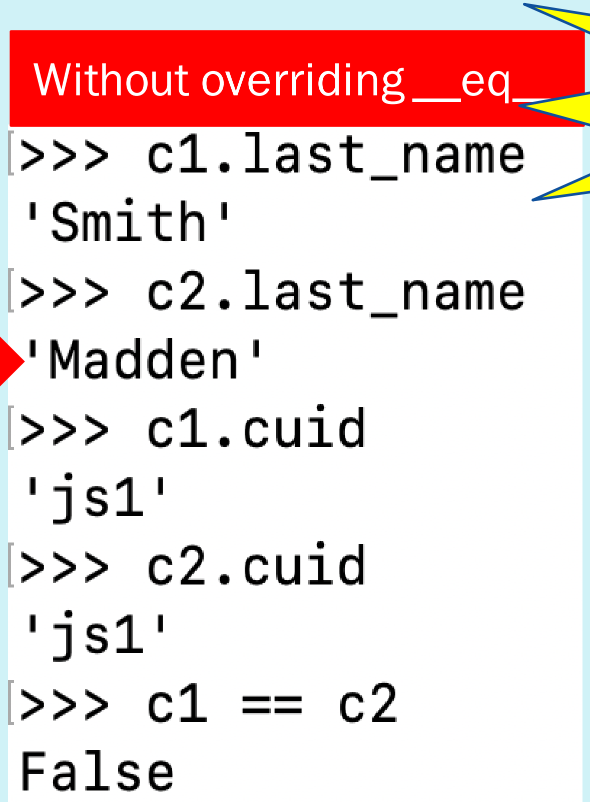
__eq__ for Cornellian



```python
    def __eq__():
        """
        Returns True if other is a Cornellian object equal to this one.

        Two Cornellians are equal if they:
            1. Both have the same cuid
        NOTICE: Two Cornellians can be equal if they have different names

        Preconditions: None (other can be anything)
        """


        # STUDENTS--You need to do the following:
            # 1. Put the parameters into the __eq__ method header above
            # 2. Complete this method so it follows the specification above
        pass
```

What does overriding __eq__ do?

- This changes how Python checks for equality.
- Say c1 and c2 were two Cornellian Objects.
- Without implementing __eq__, Python would simply check that the ids of the folders were the same when c1 == c2 was called.
- However, if we override __eq__, Python will now check something different when we call c1 == c2 (it will check that the cuid in each folder is the same), so we get the response we want.
- What determines which __eq__ is called?
  - We use the __eq__ associated with the first object listed
  - Because when Python sees c1 == c2, it translates this into c1.__eq__(c2)

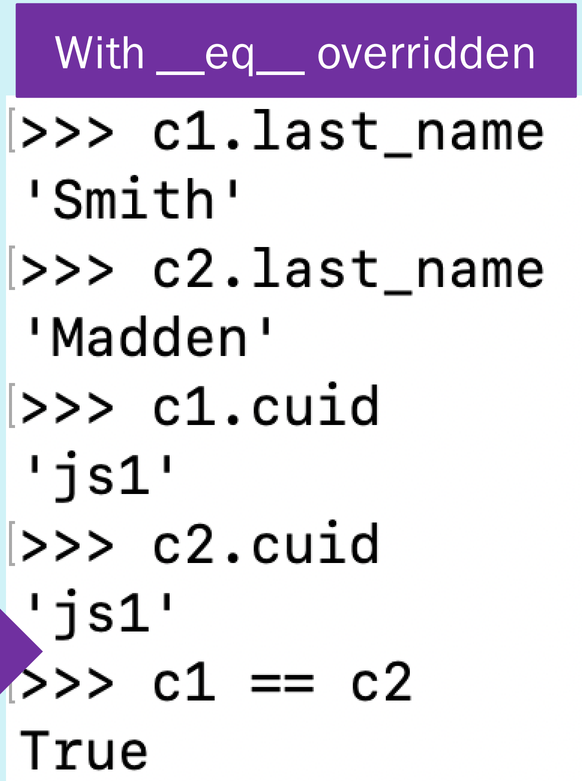


Without overriding __eq__

```
>>> c1.last_name
'Smith'
>>> c2.last_name
'Madden'
>>> c1.cuid
'js1'
>>> c2.cuid
'js1'
>>> c1 == c2
False
```

== is for equality

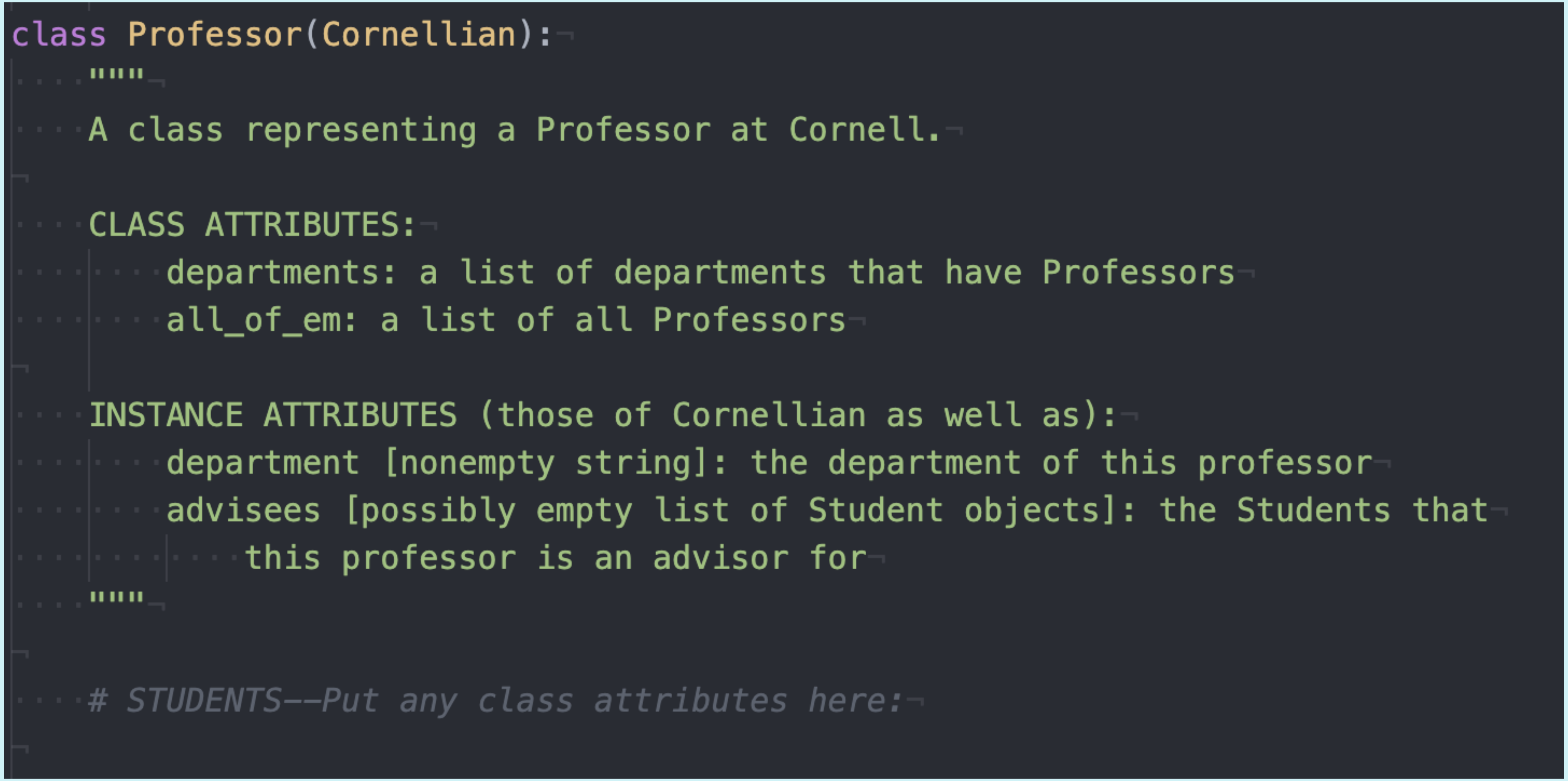
= is for assignment



With __eq__ overridden

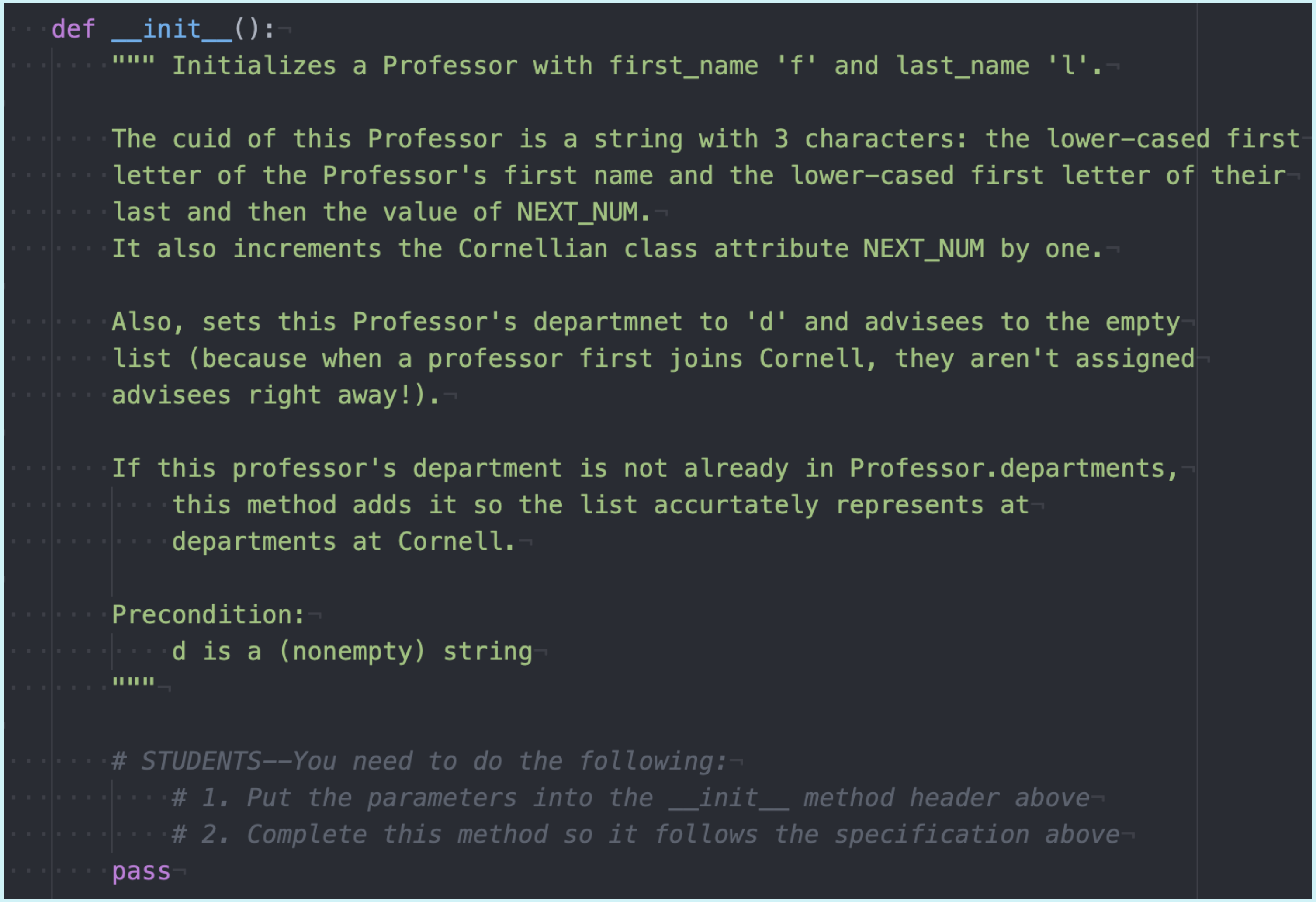
```
>>> c1.last_name
'Smith'
>>> c2.last_name
'Madden'
>>> c1.cuid
'js1'
>>> c2.cuid
'js1'
>>> c1 == c2
True
```

Class Attribute(s) for Professor



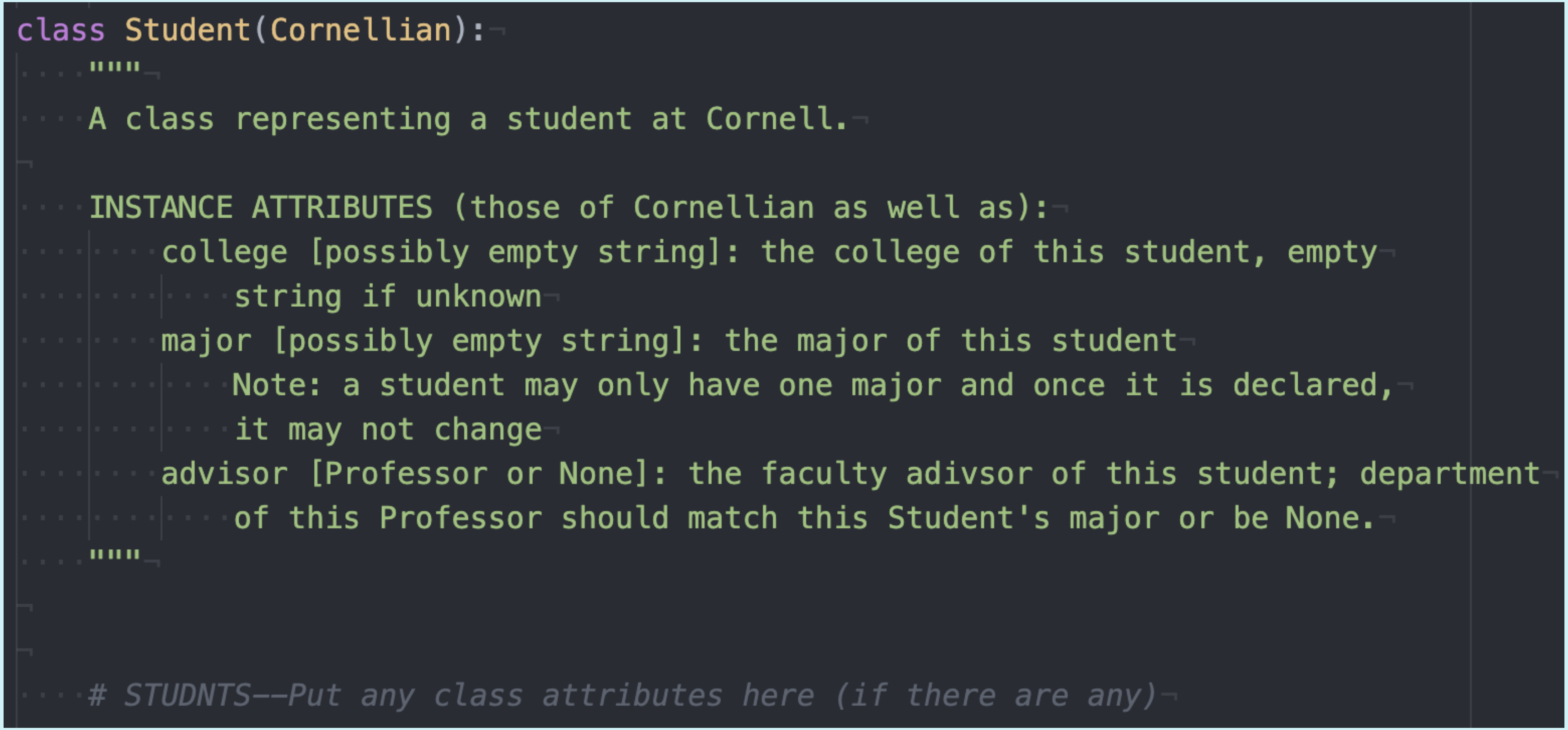
```python
class Professor(Cornellian):
    """
    A class representing a Professor at Cornell.

    CLASS ATTRIBUTES:
        departments: a list of departments that have Professors
        all_of_em: a list of all Professors

    INSTANCE ATTRIBUTES (those of Cornellian as well as):
        department [nonempty string]: the department of this professor
        advisees [possibly empty list of Student objects]: the Students that
            this professor is an advisor for
    """

    # STUDENTS--Put any class attributes here:

```

__init__ for Professor



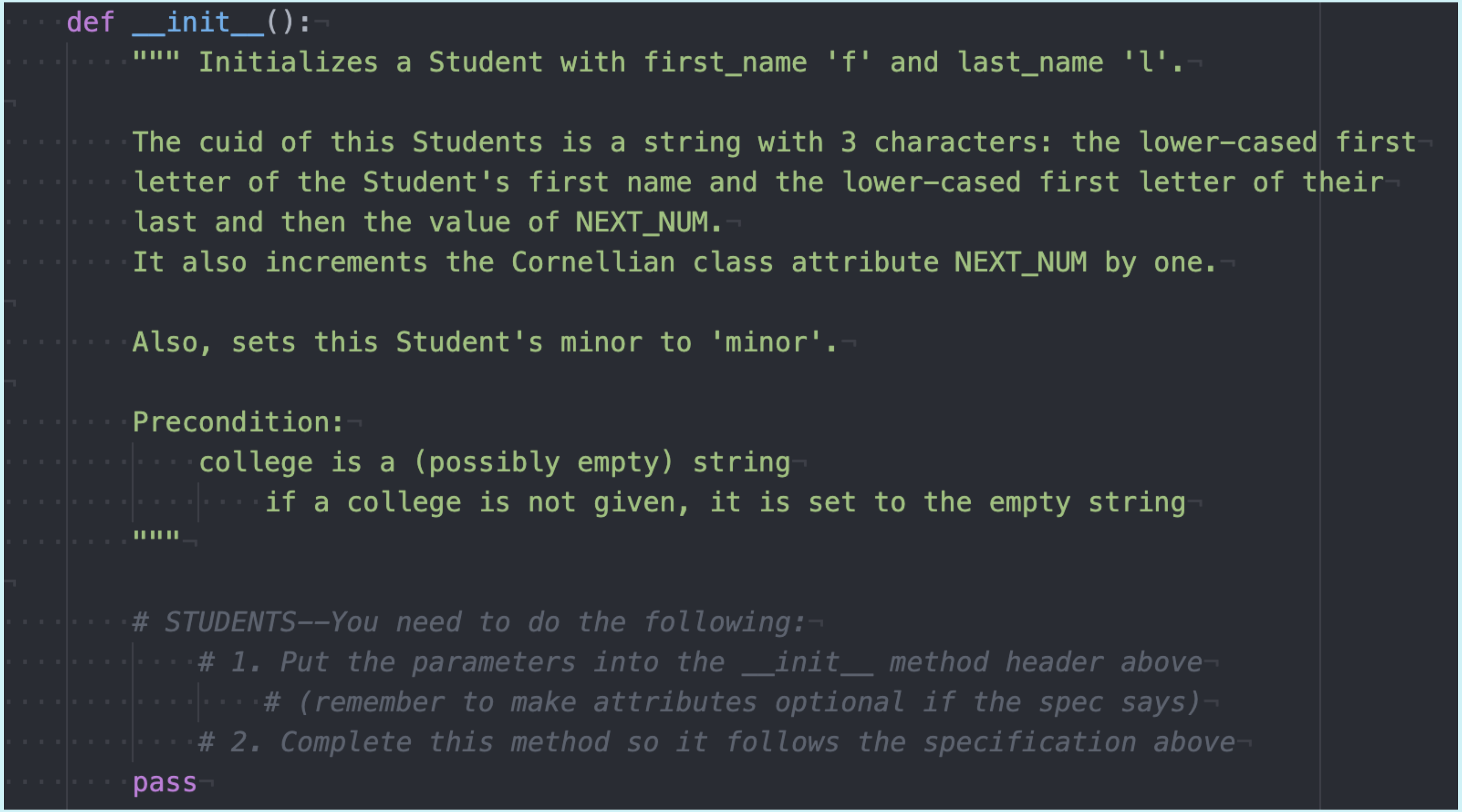
```
    def __init__():
        """ Initializes a Professor with first_name 'f' and last_name 'l'.

        The cuid of this Professor is a string with 3 characters: the lower-cased first
        letter of the Professor's first name and the lower-cased first letter of their
        last and then the value of NEXT_NUM.
        It also increments the Cornellian class attribute NEXT_NUM by one.

        Also, sets this Professor's departmnet to 'd' and advisees to the empty
        list (because when a professor first joins Cornell, they aren't assigned
        advisees right away!).

        If this professor's department is not already in Professor.departments,
            this method adds it so the list accurtately represents at
            departments at Cornell.

        Precondition:
            d is a (nonempty) string
        """

        # STUDENTS--You need to do the following:
            # 1. Put the parameters into the __init__ method header above
            # 2. Complete this method so it follows the specification above
        pass
```

Class Attribute(s) for Student?



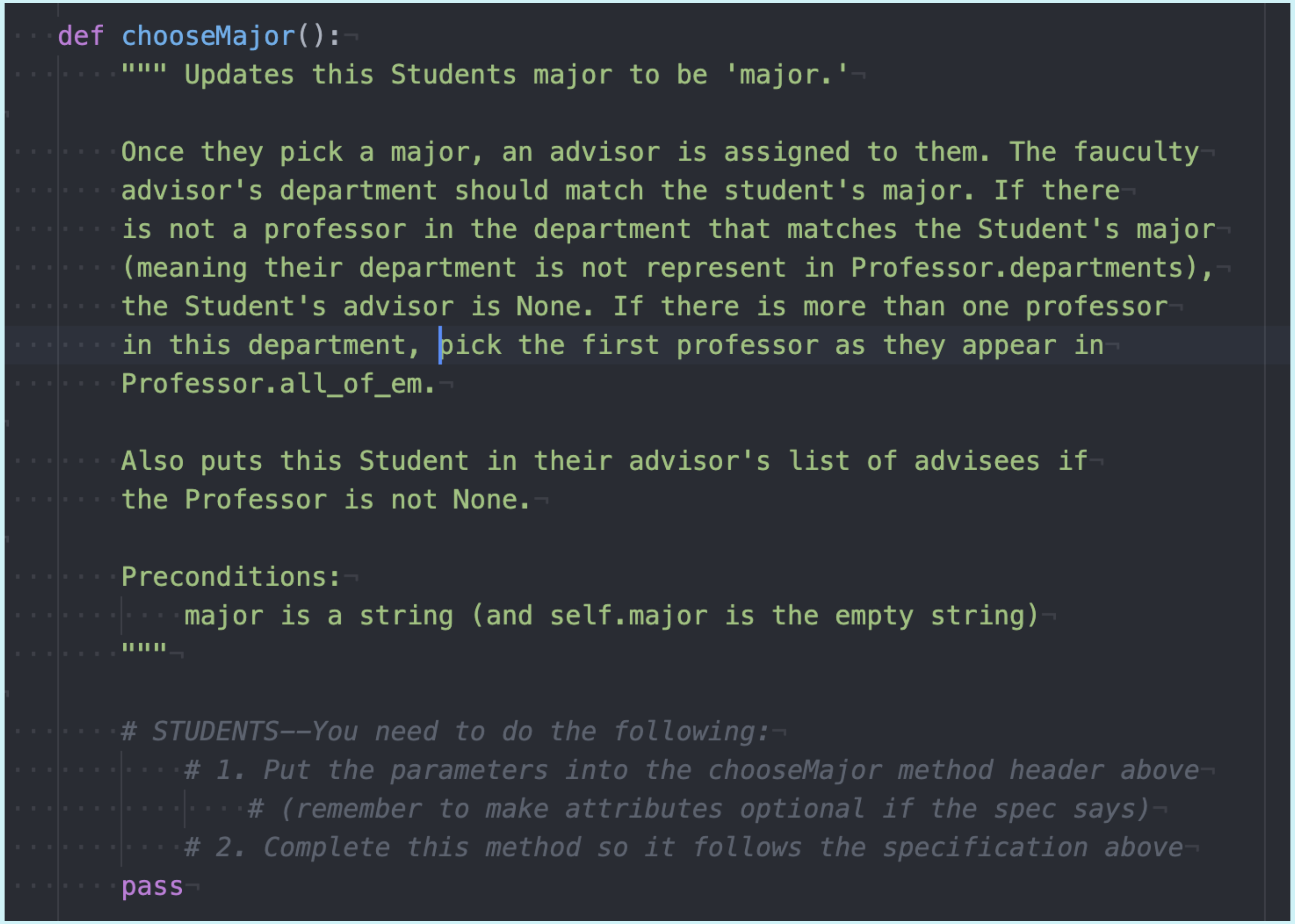
```
class Student(Cornellian):
    """
    A class representing a student at Cornell.

    INSTANCE ATTRIBUTES (those of Cornellian as well as):
        college [possibly empty string]: the college of this student, empty
            string if unknown
        major [possibly empty string]: the major of this student
            Note: a student may only have one major and once it is declared,
            it may not change
        advisor [Professor or None]: the faculty adivsor of this student; department
            of this Professor should match this Student's major or be None.
    """


    # STUDNTS--Put any class attributes here (if there are any)
```

__init__ for Student



```
    def __init__():
        """ Initializes a Student with first_name 'f' and last_name 'l'.

        The cuid of this Students is a string with 3 characters: the lower-cased first
        letter of the Student's first name and the lower-cased first letter of their
        last and then the value of NEXT_NUM.
        It also increments the Cornellian class attribute NEXT_NUM by one.

        Also, sets this Student's minor to 'minor'.

        Precondition:
            college is a (possibly empty) string
                if a college is not given, it is set to the empty string
        """

        # STUDENTS--You need to do the following:
            # 1. Put the parameters into the __init__ method header above
                # (remember to make attributes optional if the spec says)
            # 2. Complete this method so it follows the specification above
        pass
```

chooseMajor for Student



```python
    def chooseMajor():
        """ Updates this Students major to be 'major.'

        Once they pick a major, an advisor is assigned to them. The fauculty
        advisor's department should match the student's major. If there
        is not a professor in the department that matches the Student's major
        (meaning their department is not represent in Professor.departments),
        the Student's advisor is None. If there is more than one professor
        in this department, pick the first professor as they appear in
        Professor.all_of_em.

        Also puts this Student in their advisor's list of advisees if
        the Professor is not None.

        Preconditions:
            major is a string (and self.major is the empty string)
        """

        # STUDENTS--You need to do the following:
            # 1. Put the parameters into the chooseMajor method header above
                # (remember to make attributes optional if the spec says)
            # 2. Complete this method so it follows the specification above
        pass
```

THANK YOU FOR COMING

Any Questions?