bianca.phelan@tudublin.ie

O Object
O Oriented
P Programming

S1 = Python with Bianca

S2 = Java with Bryan

DT228(TU856)/DT282(TU858) - 2

OLLSCOIL TEICNEOLAÍOCHTA BHAILE ÁTHA CLIATH

DUBLIN

TECHNOLOGICAL UNIVERSITY DUBLIN

COMPUTER SCIENCE

Inheritance and Friends

Objectives

- Discuss different variable scopes in Python
- Revise the principle of inheritance
- Discuss the principle of composition and aggregation
- Analyse the principle of abstract classes
- Program Polymorphism in Python

Object Basic Principle

- Look at the real world:
  - Your dog
  - Your desk lamp
  - Your tv
- All have a **state** and a **behavior**
  - Example dog:
  - State: breed, size, colour, name
  - Behaviour: bark, play fetch, go walkies

- Some objects are more complex than other objects
- Some objects contain other objects.

[1]

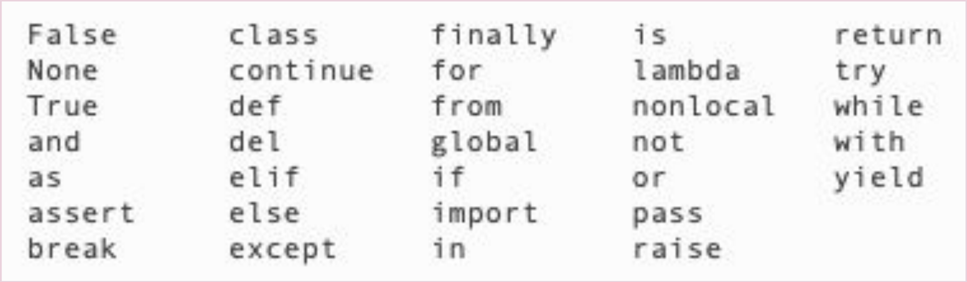
Scope of a Variable

Scope is the region a variable is created in.

- **Not all variables can be accessed from anywhere within the program**
  - The part of a program where a variable is accessible is its scope
  - LEGB rule: **L**ocal -> **E**nclosing -> **G**lobal -> **B**uilt-in
- **Local** Scope = variable created inside a function/method
  - The variable can only be seen and used within this function/method
  - And by inner functions, that a function inside a function (enclosing scope, only works one-way)
- **Gobal** Scope = variable created inside main body of Python code
  - Available everyone
- **Built-in**: keywords that are available from everywhere

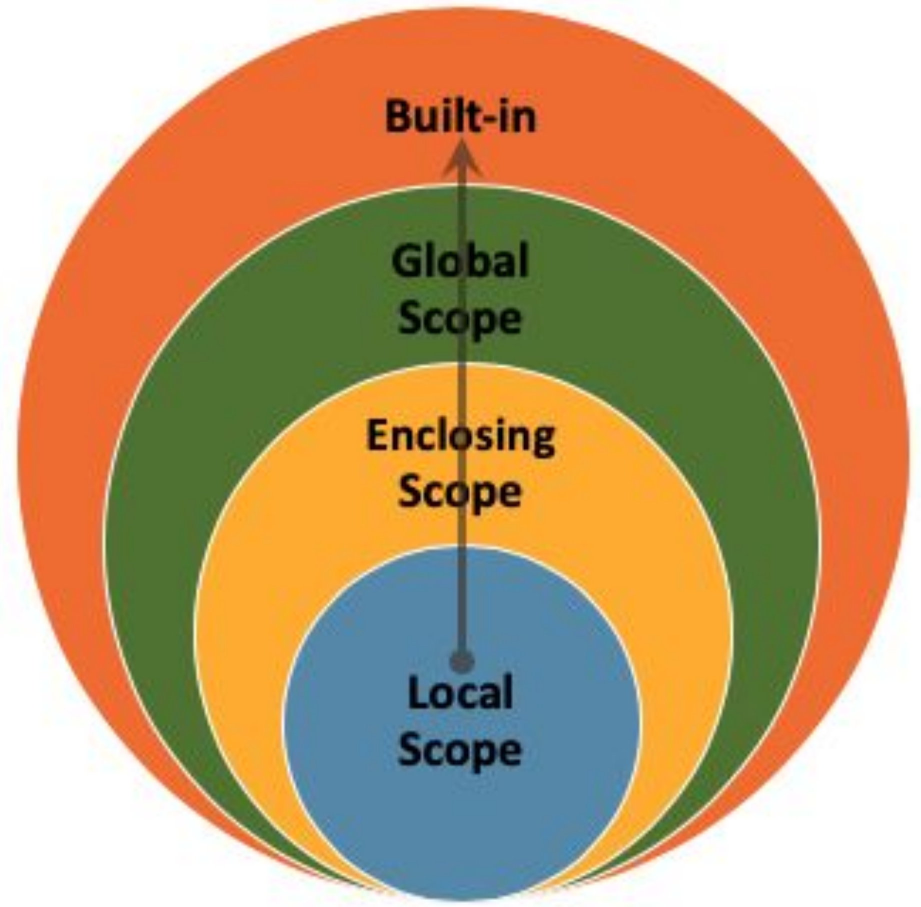
[12]

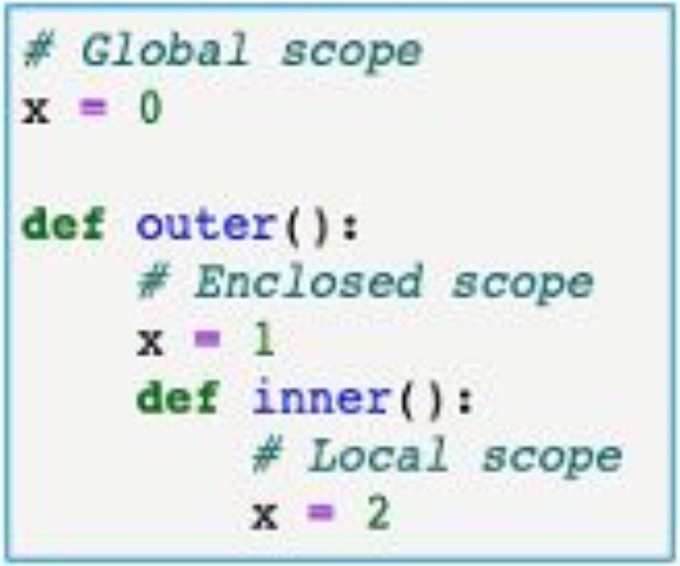
Example Built-in Scope:



```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

[12]



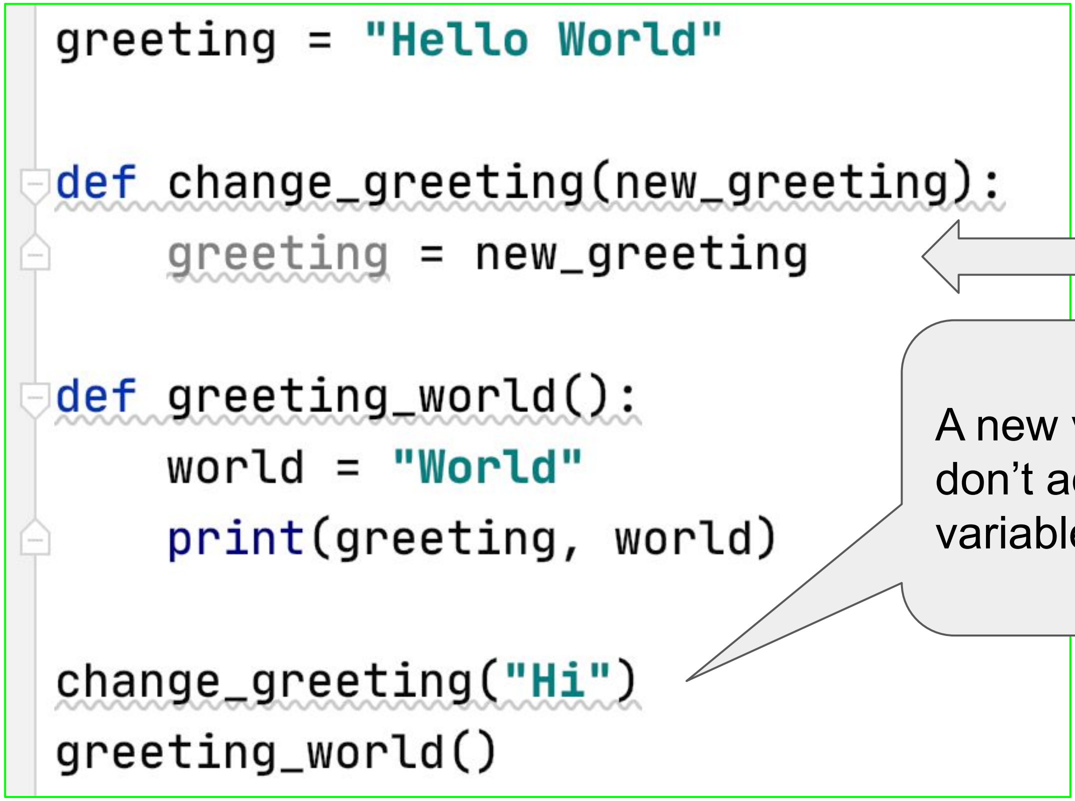


```
# Global scope
x = 0

def outer():
    # Enclosed scope
    x = 1
    def inner():
        # Local scope
        x = 2
```

Global Scope

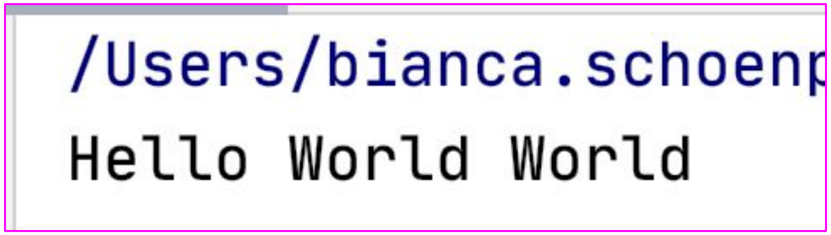
Scope Example 1.1

[12]



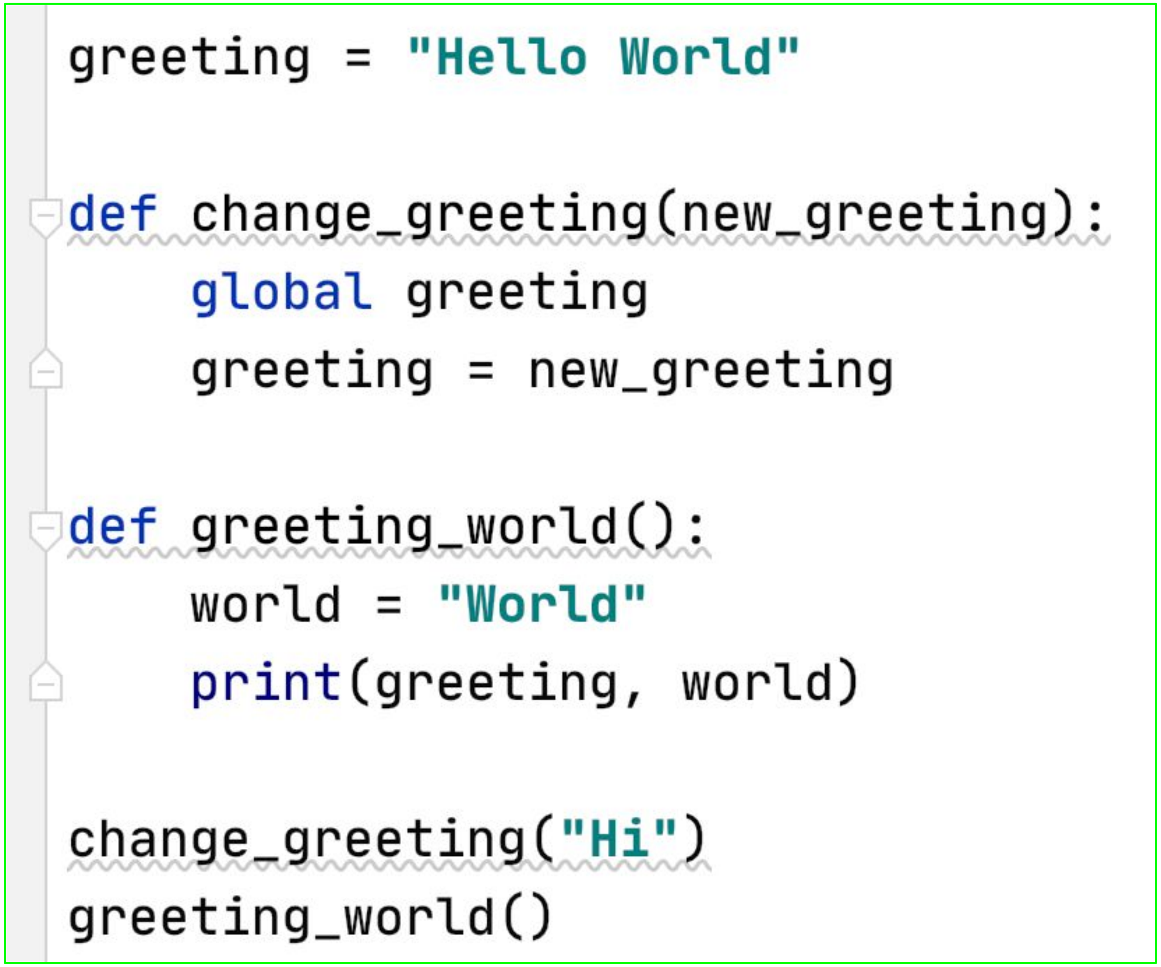
```
greeting = "Hello World"

def change_greeting(new_greeting):
    greeting = new_greeting

def greeting_world():
    world = "World"
    print(greeting, world)

change_greeting("Hi")
greeting_world()
```

A new variable is created. We don’t actually change the global variable that we meant to change.



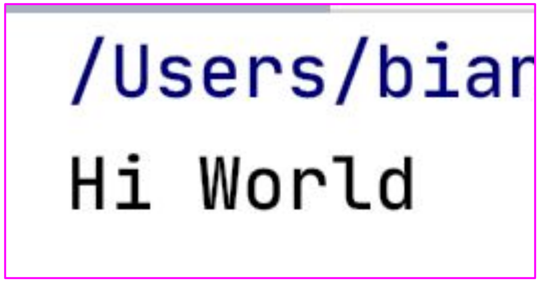
```
/Users/bianca.schoenp
Hello World World
```

Scope Example 1.2 The `global` keyword



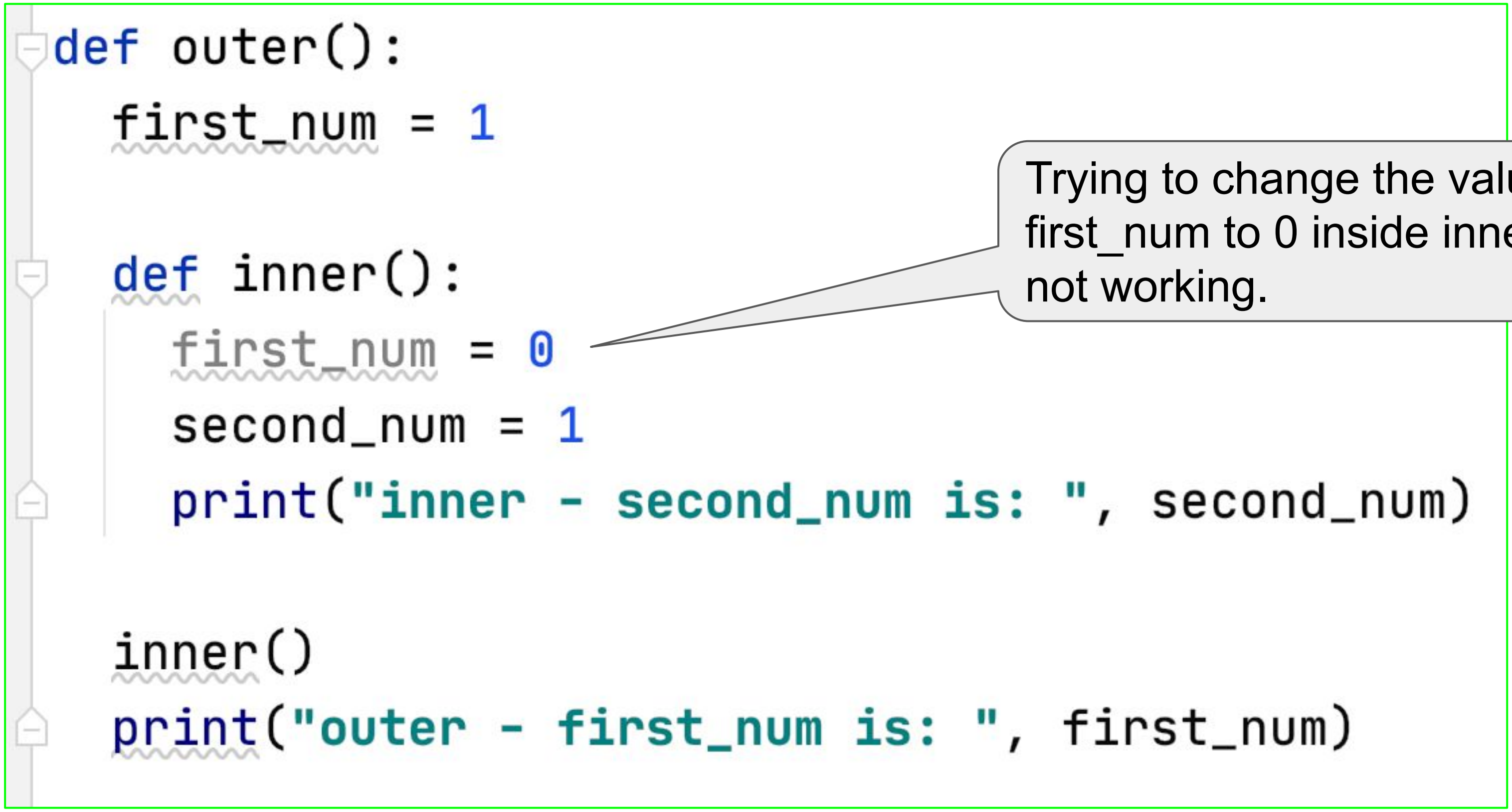
```python
greeting = "Hello World"

def change_greeting(new_greeting):
    global greeting
    greeting = new_greeting

def greeting_world():
    world = "World"
    print(greeting, world)

change_greeting("Hi")
greeting_world()
```

Global Scope



```
/Users/biar
Hi World
```

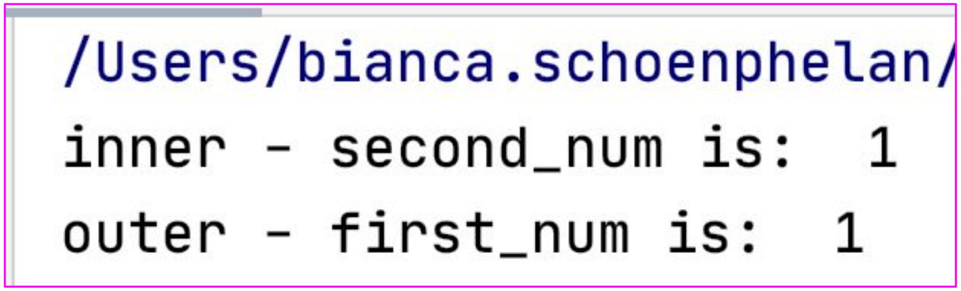
Scope Example 2.1: Enclosing Scope



```python
def outer():
    first_num = 1

    def inner():
        first_num = 0
        second_num = 1
        print("inner - second_num is: ", second_num)

    inner()
    print("outer - first_num is: ", first_num)
```

Trying to change the value of first_num to 0 inside inner(), which is not working.

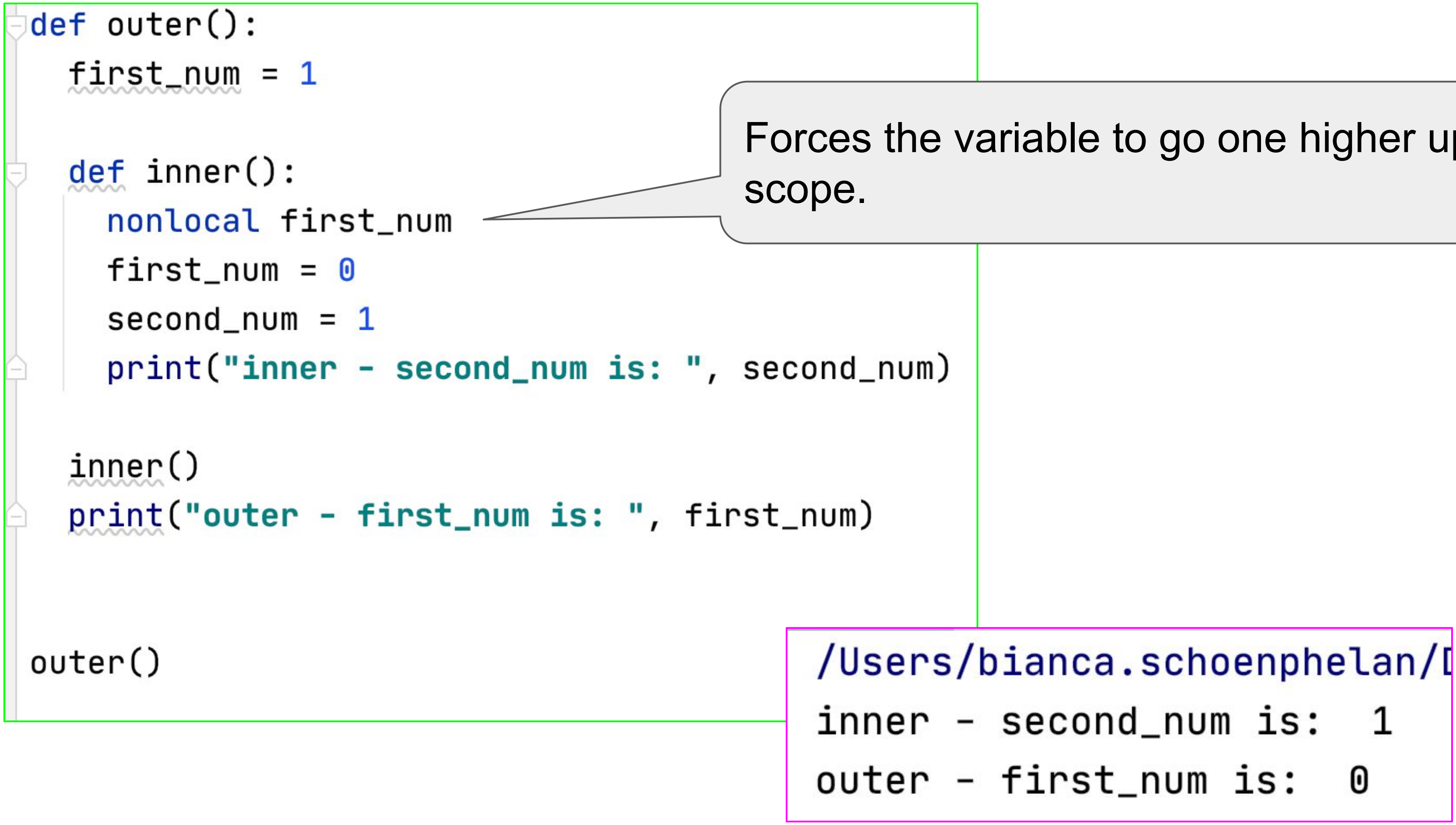
[12]



```
/Users/bianca.schoenphelan/
inner - second_num is:  1
outer - first_num is:  1
```

Scope Example 2.2: Enclosing Scope - the `nonlocal` keyword

[12]

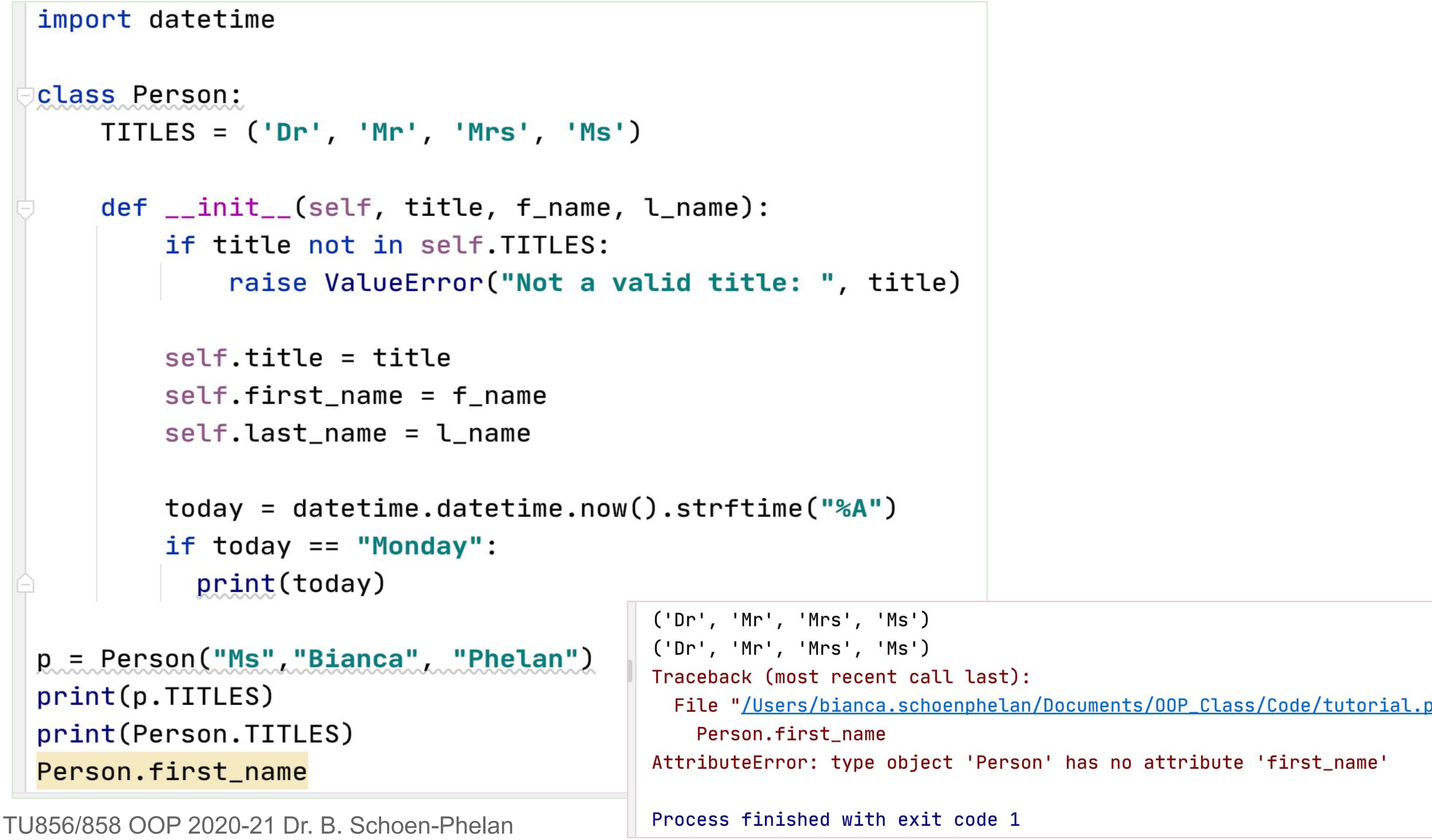


```python
def outer():
    first_num = 1

    def inner():
        nonlocal first_num
        first_num = 0
        second_num = 1
        print("inner - second_num is: ", second_num)

    inner()
    print("outer - first_num is: ", first_num)


outer()
```

Forces the variable to go one higher up in the scope.

```
/Users/bianca.schoenphelan/D
inner - second_num is:  1
outer - first_num is:  0
```

Scope Example 3: Class Instance VS Class Attribute



```
import datetime

class Person:
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, title, f_name, l_name):
        if title not in self.TITLES:
            raise ValueError("Not a valid title: ", title)

        self.title = title
        self.first_name = f_name
        self.last_name = l_name

        today = datetime.datetime.now().strftime("%A")
        if today == "Monday":
          print(today)

p = Person("Ms","Bianca", "Phelan")
print(p.TITLES)
print(Person.TITLES)
Person.first_name
```

```
('Dr', 'Mr', 'Mrs', 'Ms')
('Dr', 'Mr', 'Mrs', 'Ms')
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/Code/tutorial.p
    Person.first_name
AttributeError: type object 'Person' has no attribute 'first_name'

Process finished with exit code 1
```

Mid-module SURVEY

https://forms.gle/PtmNYY6ary56AtUWA

- Anonymous
- What would you like to understand better?

Icon from flaticon.com

Inheritance

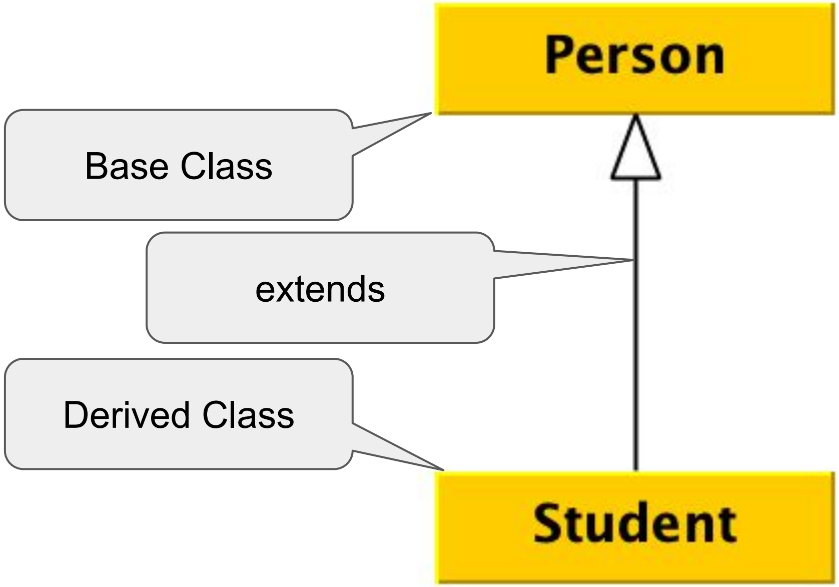
- Inheritance allows us to reuse code
  - See for an example the word game lab
- We create a class and it is allowed to use all the methods and attributes from another class
- This essentially creates a hierarchy from parent class down to child classes, which is often illustrated using a tree structure
- It's a big part of what makes an object-oriented programming language OOP in the first place

The class we inherit from is called the parent class, base class or the superclass.

Inheritance is an **is-a** relationship.

Python supports multiple inheritance.

Inheritance Example



- `Student` inherits from `Person`
- What do they inherit?
  - Methods
  - Attributes

Inheritance models an **is-a** relationship.
For example, a Student **is-a** Person.

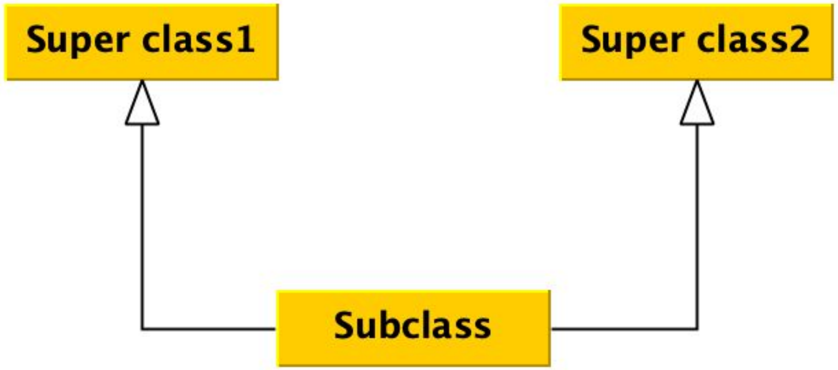
super() in Python

- Python 2 super syntax is much more complicated
- A call to super can be made inside **any** method, not just `__init__`
- All methods can be modified via overriding and calls to super
- Calls to super can happen at any stage of a method, not just as the first line
- `super()` creates a temporary object of the super class which then lets you initialise members.

Method Resolution Order

- Every class has a `__mro__` attribute that tells you how Python resolves the hierarchy
- This means that hierarchy order of the path that the Python interpreter takes to find the method in the parent to execute

Multiple Inheritance



First mentioned parent is first in the MRO.

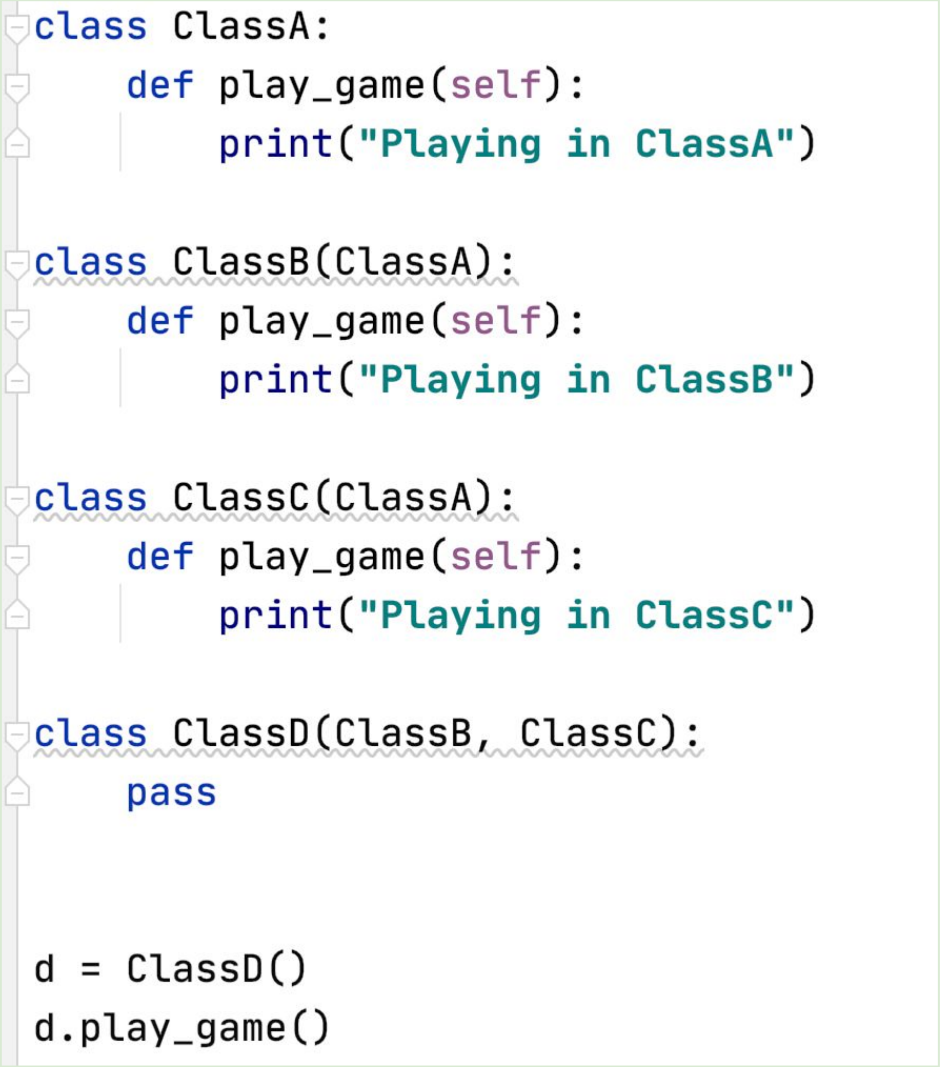
- If both super classes have a method of the same name, then which one is being called?
- Also what about __init__?
    - Called twice, unless you use super(), see example from tutorial

```
class Sub(Super1, Super2):
    pass
```

Multiple Inheritance

- A class inherits from more than one method
  - Not allowed in Java
  - Allowed also in C++
- Less useful than it sounds
- Often results in hard to maintain and understand code
- Gets messy if we try to call methods from the parent class
  - What does super refer to?
  - How do we know the order to call them in?
  - MRO might not be obvious
- Argument passing also tricky, Python uses ‘pseudo pointer’ called **kwargs

Multiple Inheritance Example 1

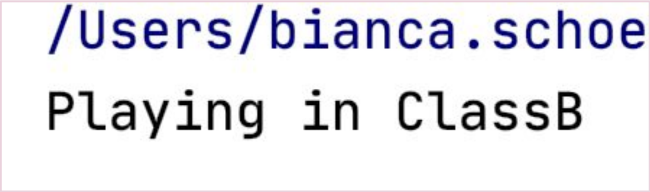


```python
class ClassA:
    def play_game(self):
        print("Playing in ClassA")

class ClassB(ClassA):
    def play_game(self):
        print("Playing in ClassB")

class ClassC(ClassA):
    def play_game(self):
        print("Playing in ClassC")

class ClassD(ClassB, ClassC):
    pass


d = ClassD()
d.play_game()
```

Case 1: Method overriden in all parent classes.

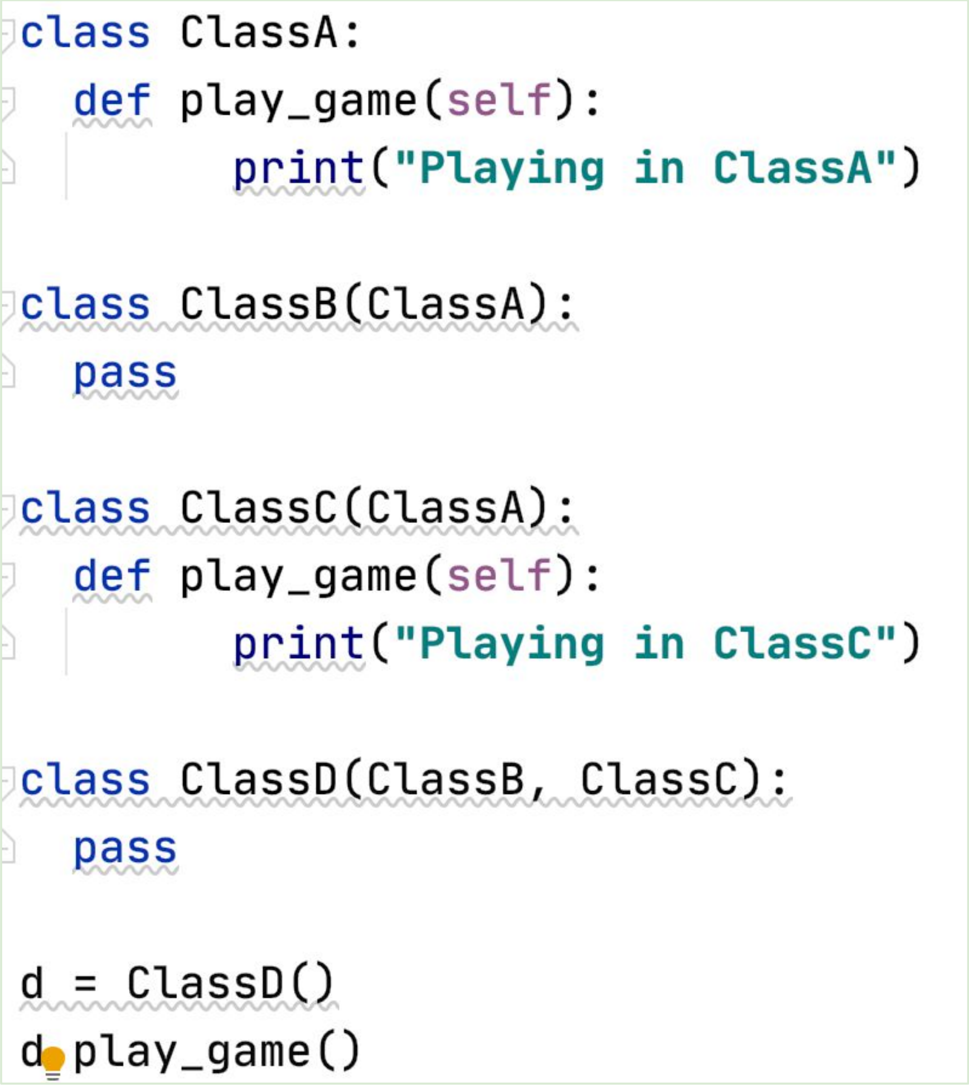
[10]

This output depends on which class is mentioned first in the brackets.



```
/Users/bianca.schoe
Playing in ClassB
```

Multiple Inheritance Example 2

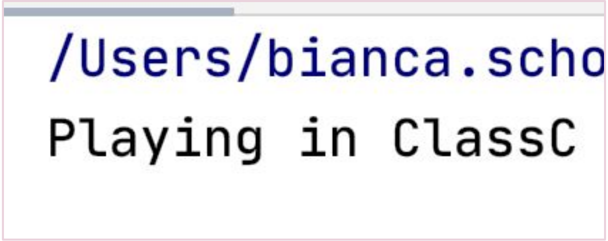


```
class ClassA:
    def play_game(self):
        print("Playing in ClassA")

class ClassB(ClassA):
    pass

class ClassC(ClassA):
    def play_game(self):
        print("Playing in ClassC")

class ClassD(ClassB, ClassC):
    pass

d = ClassD()
d.play_game()
```

Case 2: Method overriden in some parent classes.

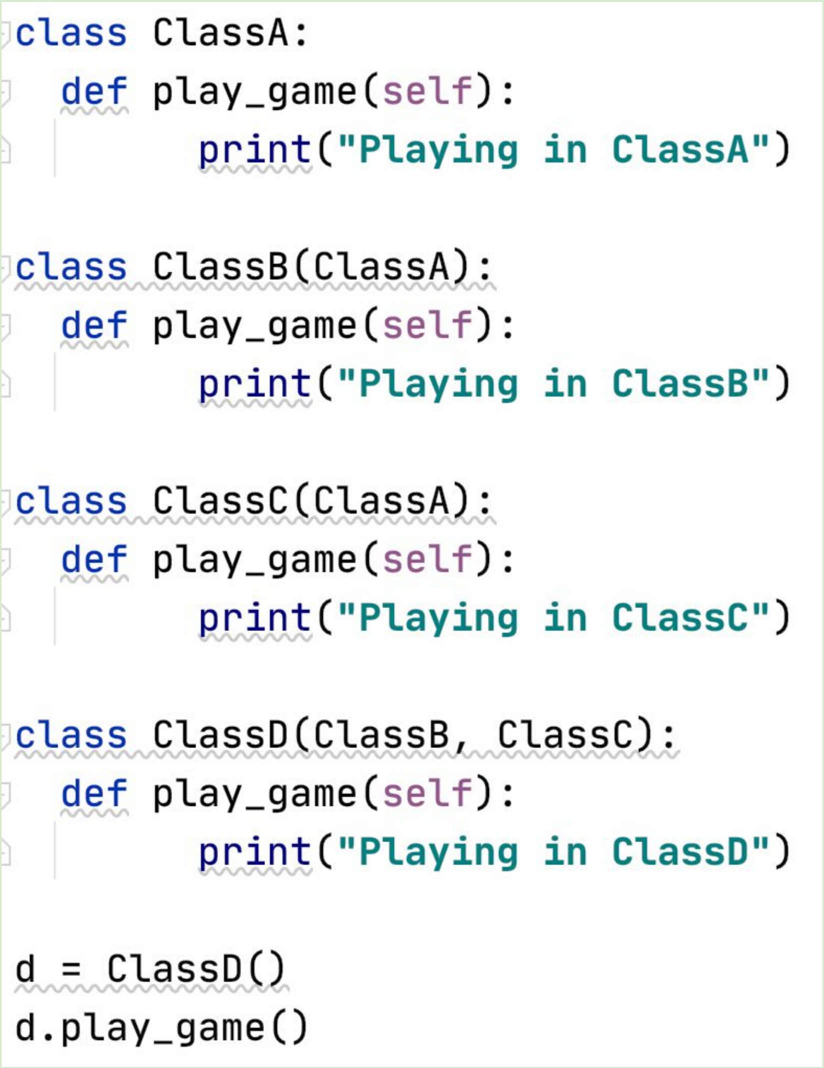
[10]

This output depends on where the first mention in the mro is of this method.



```
/Users/bianca.scho
Playing in ClassC
```

Multiple Inheritance Example 3

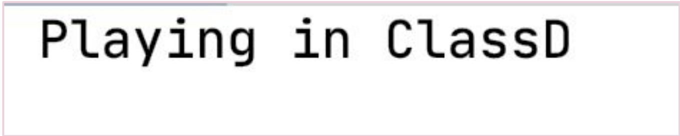


```
class ClassA:
    def play_game(self):
        print("Playing in ClassA")

class ClassB(ClassA):
    def play_game(self):
        print("Playing in ClassB")

class ClassC(ClassA):
    def play_game(self):
        print("Playing in ClassC")

class ClassD(ClassB, ClassC):
    def play_game(self):
        print("Playing in ClassD")

d = ClassD()
d.play_game()
```

Case 3: Method overriden in all classes.

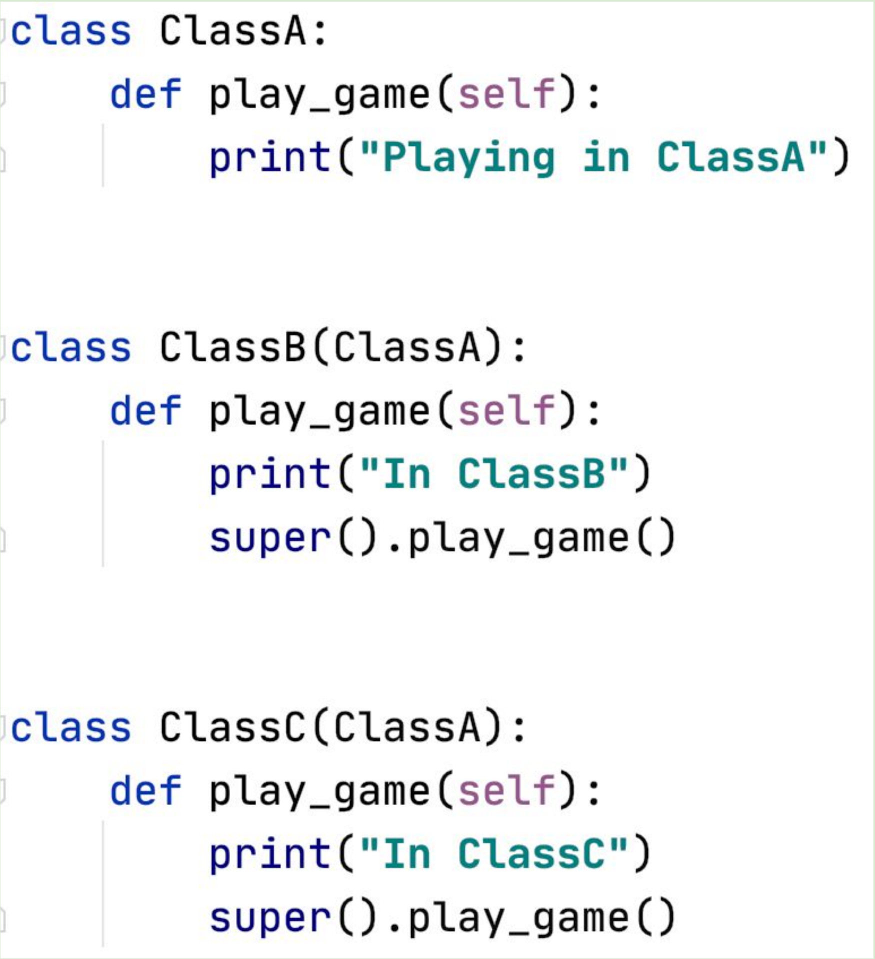
[10]

This output depends on where the first mention in the mro is of this method.

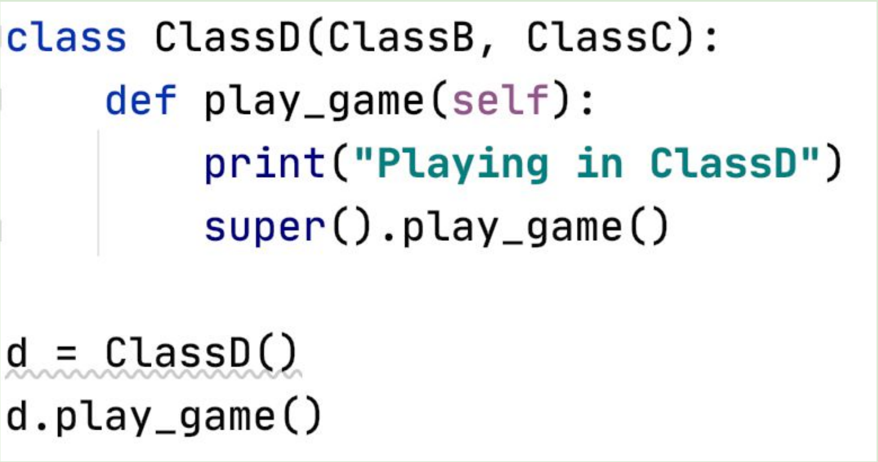


```
Playing in ClassD
```

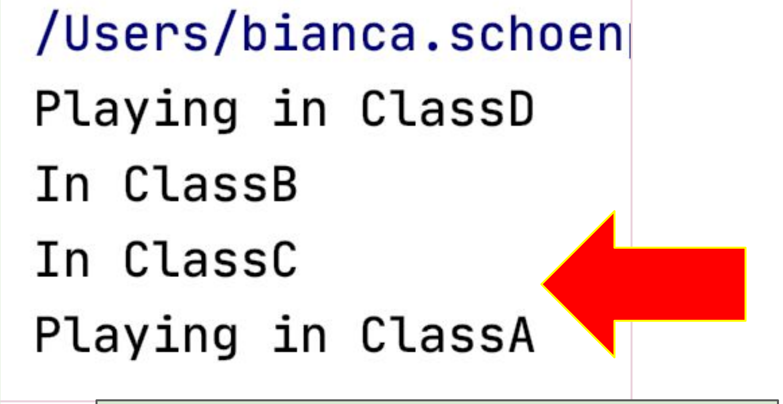
Multiple Inheritance: Example 4



```python
class ClassA:
    def play_game(self):
        print("Playing in ClassA")


class ClassB(ClassA):
    def play_game(self):
        print("In ClassB")
        super().play_game()


class ClassC(ClassA):
    def play_game(self):
        print("In ClassC")
        super().play_game()
```



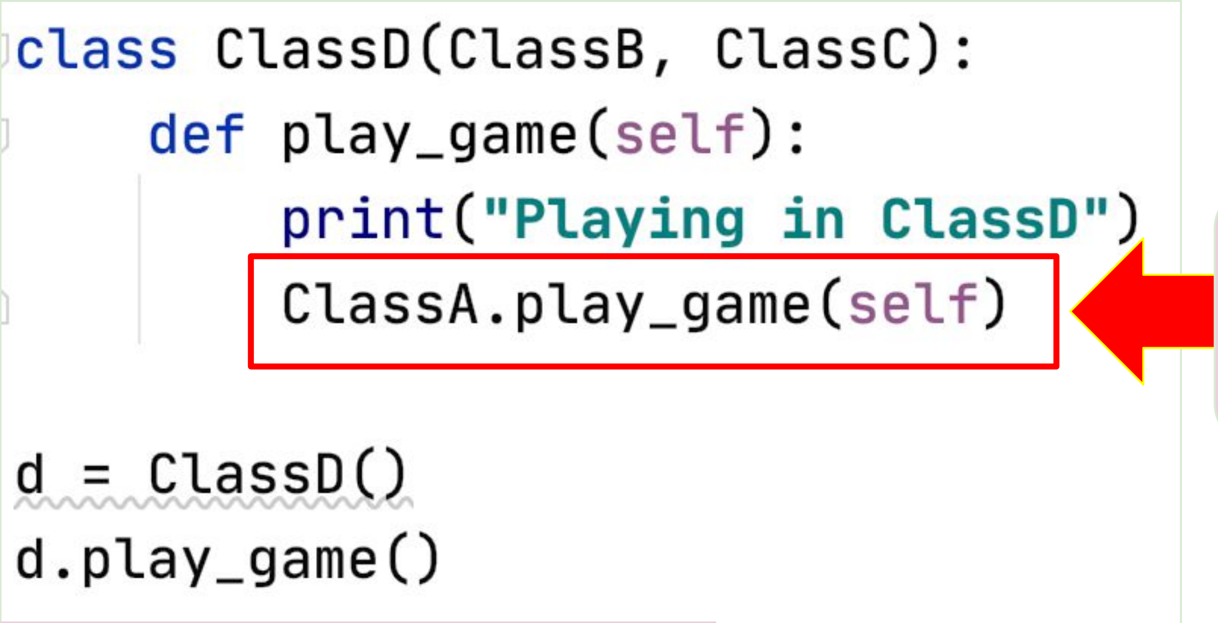
```python
class ClassD(ClassB, ClassC):
    def play_game(self):
        print("Playing in ClassD")
        super().play_game()


d = ClassD()
d.play_game()
```



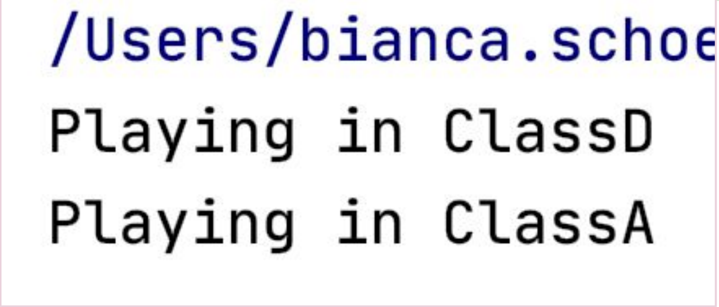
```
/Users/bianca.schoen
Playing in ClassD
In ClassB
In ClassC
Playing in ClassA
```

Case 4: Calls to super()

Multiple Inheritance, Control who’s next in line with explicit calls



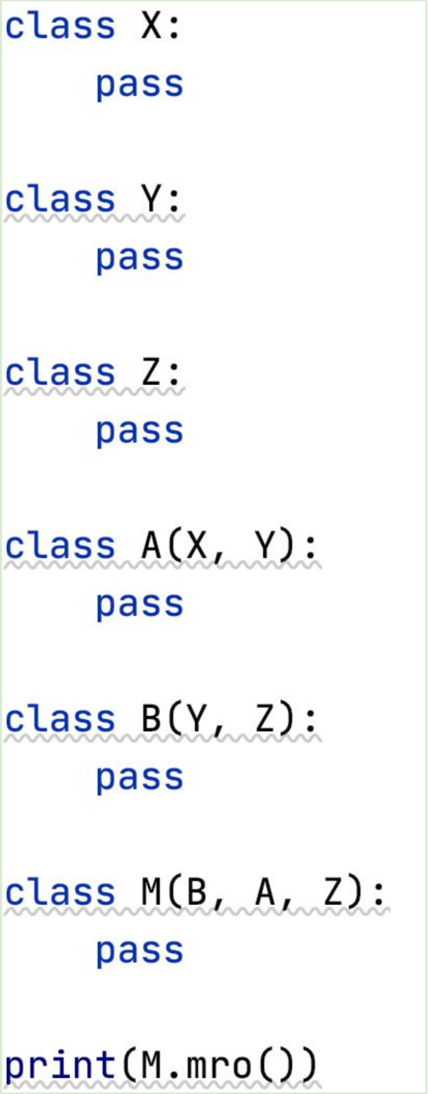
```
class ClassD(ClassB, ClassC):
    def play_game(self):
        print("Playing in ClassD")
        ClassA.play_game(self)

d = ClassD()
d.play_game()
```



```
/Users/bianca.schoe
Playing in ClassD
Playing in ClassA
```

Notice the use of `self`.

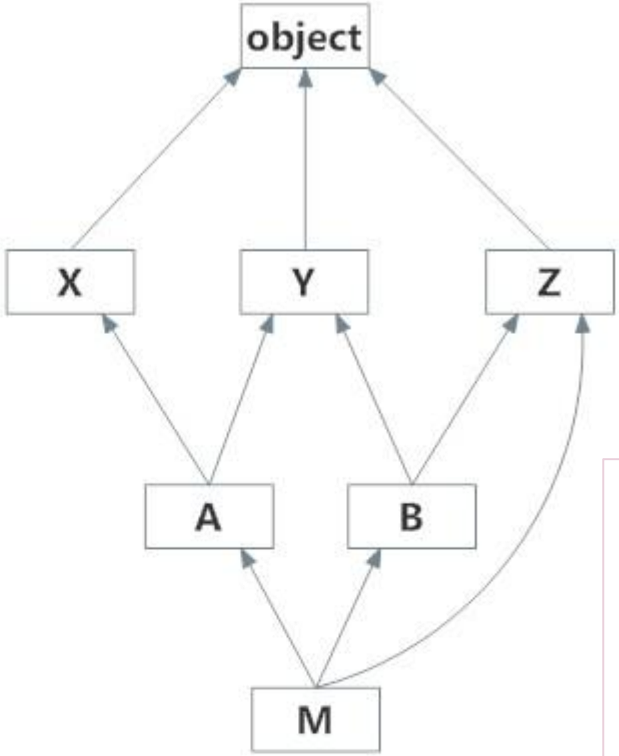
Multiple Inheritance: Example 5



```
class X:
    pass

class Y:
    pass

class Z:
    pass

class A(X, Y):
    pass

class B(Y, Z):
    pass

class M(B, A, Z):
    pass

print(M.mro())
```

What is the MRO?

[9]



```
[<class '__main__.M'>, <class
'__main__.B'>, <class
'__main__.A'>, <class
'__main__.X'>, <class
'__main__.Y'>, <class
'__main__.Z'>, <class
'object'>]
```

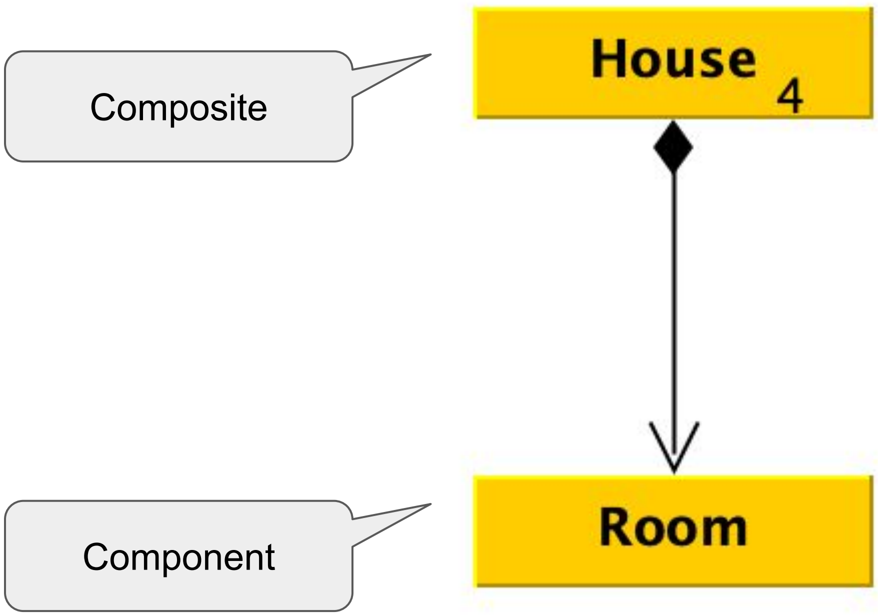
Composition

Composition

Composition enables the re-use of code without having to inherit.

- Creates a **part-of** relationship
- For the creation of very complex objects
  - It combines objects of other types

- The number indicates the the composite class contains 4 objects of the type component.
- * indicates a variable number of components
- 1...4 indicates a range of components from min to max, also 1…* possible

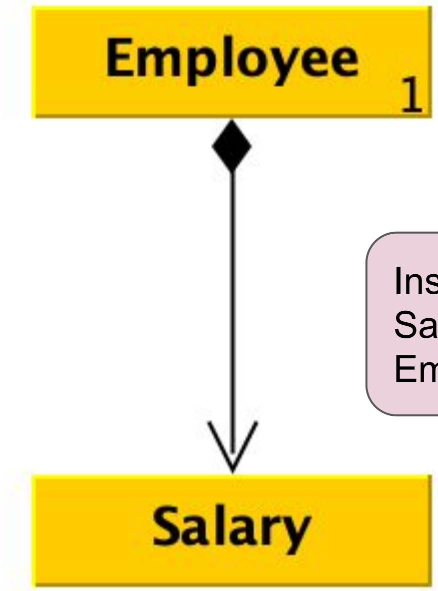


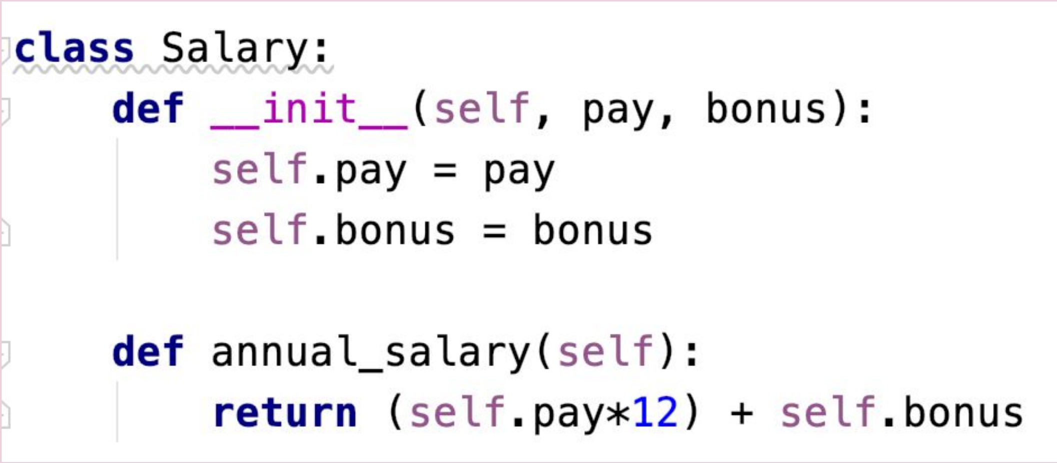
[6]

Composition

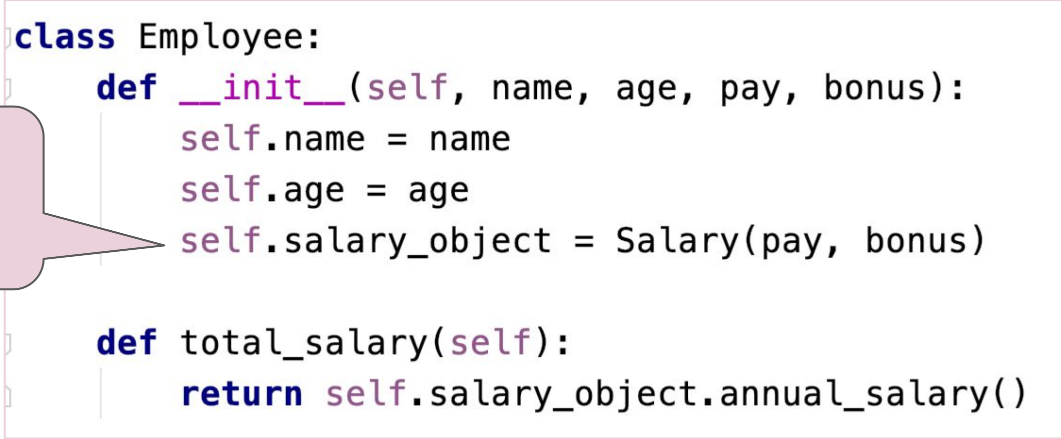
- Is considered the simpler principle (compared to inheritance)
- We collect several objects together to create a new object
- We want to use some aspect of another class without ‘promising’ all the other class’s features
- Like inheritance the aim is to re-use code, just a different way of designing your program
- Example: a car is composed of an engine, transmission, etc
- Composition provides different levels of abstractions
- Games, such as Chess, are a popular example of composition in computer systems

Example Composition



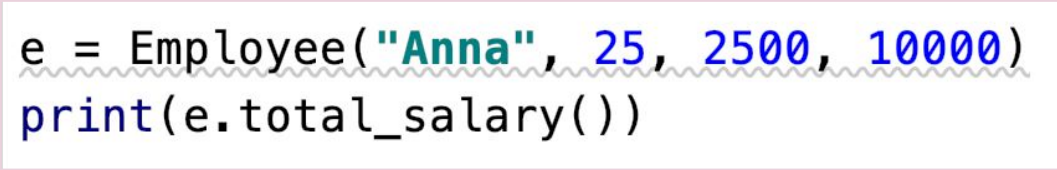


```
class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def annual_salary(self):
        return (self.pay*12) + self.bonus
```



```
class Employee:
    def __init__(self, name, age, pay, bonus):
        self.name = name
        self.age = age
        self.salary_object = Salary(pay, bonus)

    def total_salary(self):
        return self.salary_object.annual_salary()
```

Instantiating Salary inside an Employee class!



```
e = Employee("Anna", 25, 2500, 10000)
print(e.total_salary())
```

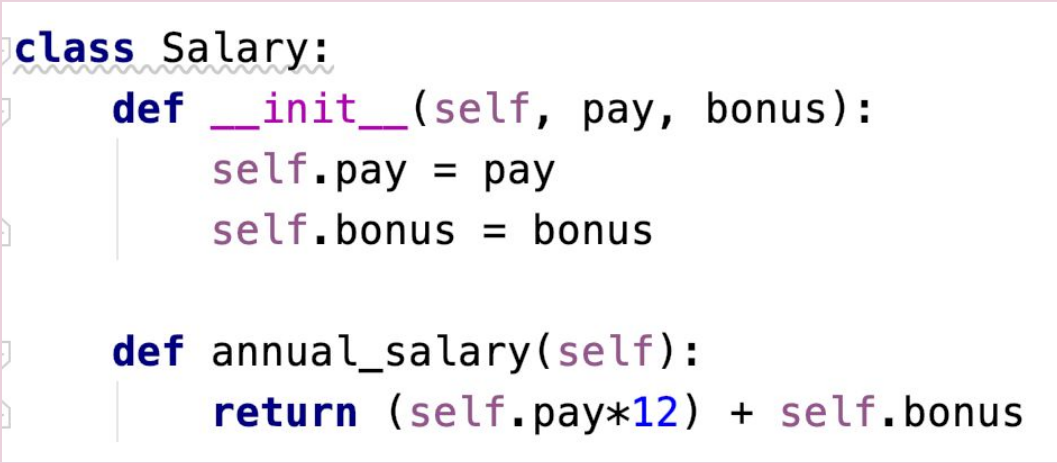
Employee has delegated responsibility to another class, the Salary.

[9]

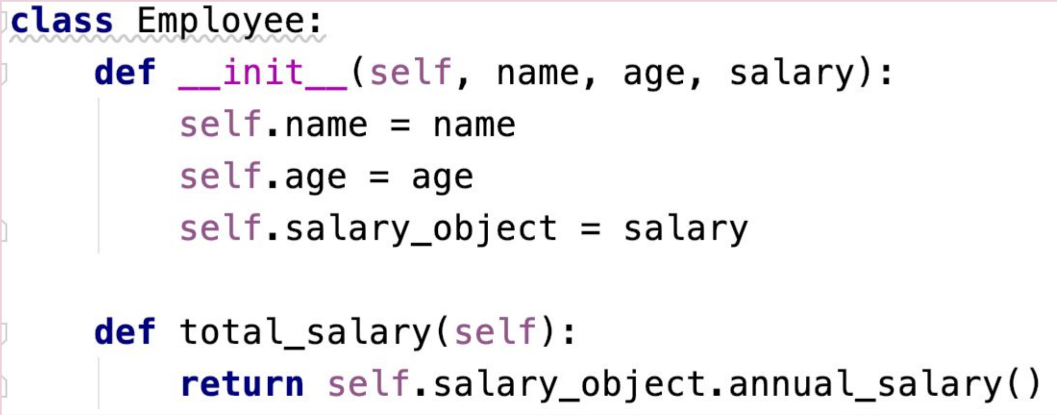
Aggregation

Aggregation

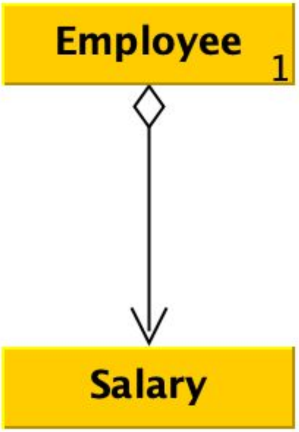
- A weak form of composition
- We have a **has-a** relationship
- Uni-directional (one-way) association
- Both objects are independent of each other



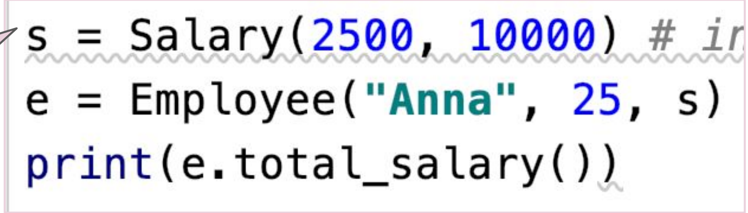
```
class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def annual_salary(self):
        return (self.pay*12) + self.bonus
```



```
class Employee:
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary_object = salary

    def total_salary(self):
        return self.salary_object.annual_salary()
```



Instantiating Salary as a separate, independent class!



```
s = Salary(2500, 10000) # ir
e = Employee("Anna", 25, s)
print(e.total_salary())
```

[10]

Summary

- ★ **Inheritance**
- ★ **Composition and Aggregation**

References

1. Solution to the diamond problem in python, http://www.aizac.info/a-solution-to-the-diamond-problem-in-python/, accessed Oct 2018.
2. Python Course, Multiple Inheritance, https://www.python-course.eu/python3_multiple_inheritance.php, accessed Oct 2018.
3. Python 3: Object-oriented programming, 2nd edition, Dusty Phillips, 2015, Packt Publishing.
4. Real Python, Inheritance and Composition, https://realpython.com/inheritance-composition-python/, accessed Nov 2019.
5. Tutorial for Beginners 33 - Composition, https://www.youtube.com/watch?v=lhiH-6ygGl8, accessed Nov 2019.
6. Python Tutorial for Beginners 34 - Aggregation, https://www.youtube.com/watch?v=rOo_BosuJBE, accessed Nov 2019.
7. Object-oriented Programming in Python, Classes, https://python-textbok.readthedocs.io/en/1.0/Classes.html, accessed Nov 2020.
8. Python Datacamp, Scope of a Variable, https://www.datacamp.com/community/tutorials/scope-of-variables-python?utm_source=adwords_ppc&utm_campaignid=898687156&utm_adgroupid=48947256715&utm_device=c&utm_keyword=&utm_matchtype=b&utm_network=g&utm_adpostion=&utm_creative=332602034349&utm_targetid=dsa-429603003980&utm_loc_interest_ms=&utm_loc_physical_ms=1007850&gclid=Cj0KCQiA-rj9BRCAARIsANB_4ACnJSyW9s26JvWYww6GskrSEuU3SYIuYbx7AQKP4RWUyE6Fxnka8twaAmALEALw_wcB, accessed Nov 2020
9. Programiz, https://www.programiz.com/python-programming/multiple-inheritance, accessed Nov 2020
10. Geeks For Geeks, Multiple Inheritance, https://www.geeksforgeeks.org/multiple-inheritance-in-python/, accessed Nov 2020.