# Principles of OOP

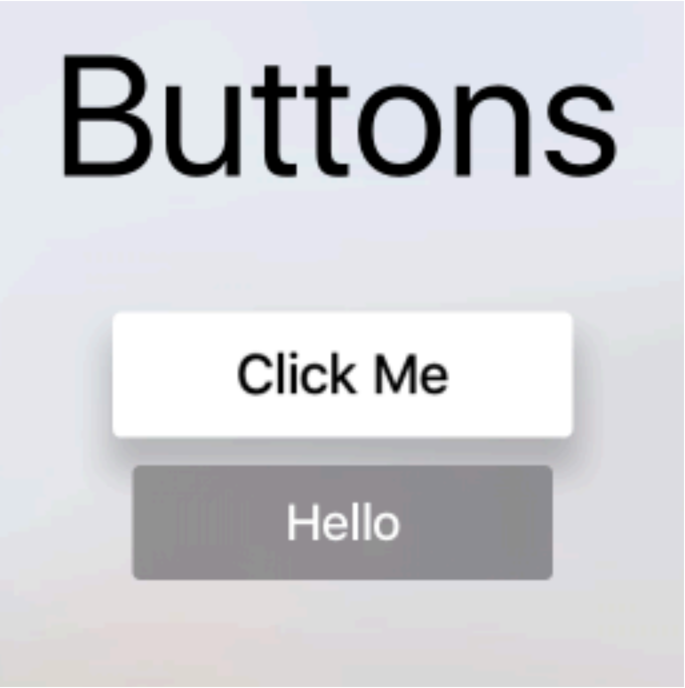
- **Encapsulation**
  - □ Encapsulation is the mechanism of hiding of data implementation by restricting access to public methods
- **Inheritance**
  - □ Inheritances expresses "is a" relationship between two objects. Using proper inheritance, In derived classes we can reuse the code of existing super classes
- **Polymorphism**
  - □ It means one name many forms. Details of what a method does will depend on the object to which it is applied.
- Also
  - □ **Instantiation**
  - □ **Abstraction**
  - □ **Modularity**

# Inheritance Motivations

- Piper is-a Dog
- Alice is-a German Shepherd
- German Sheperd is-a specialisation of Dog
- B is a Specialization of A
  - □ B has all the features of A
  - □ B can provide new features
  - □ B can perform some of the tasks performed by A in a different way

# Inheritance Example

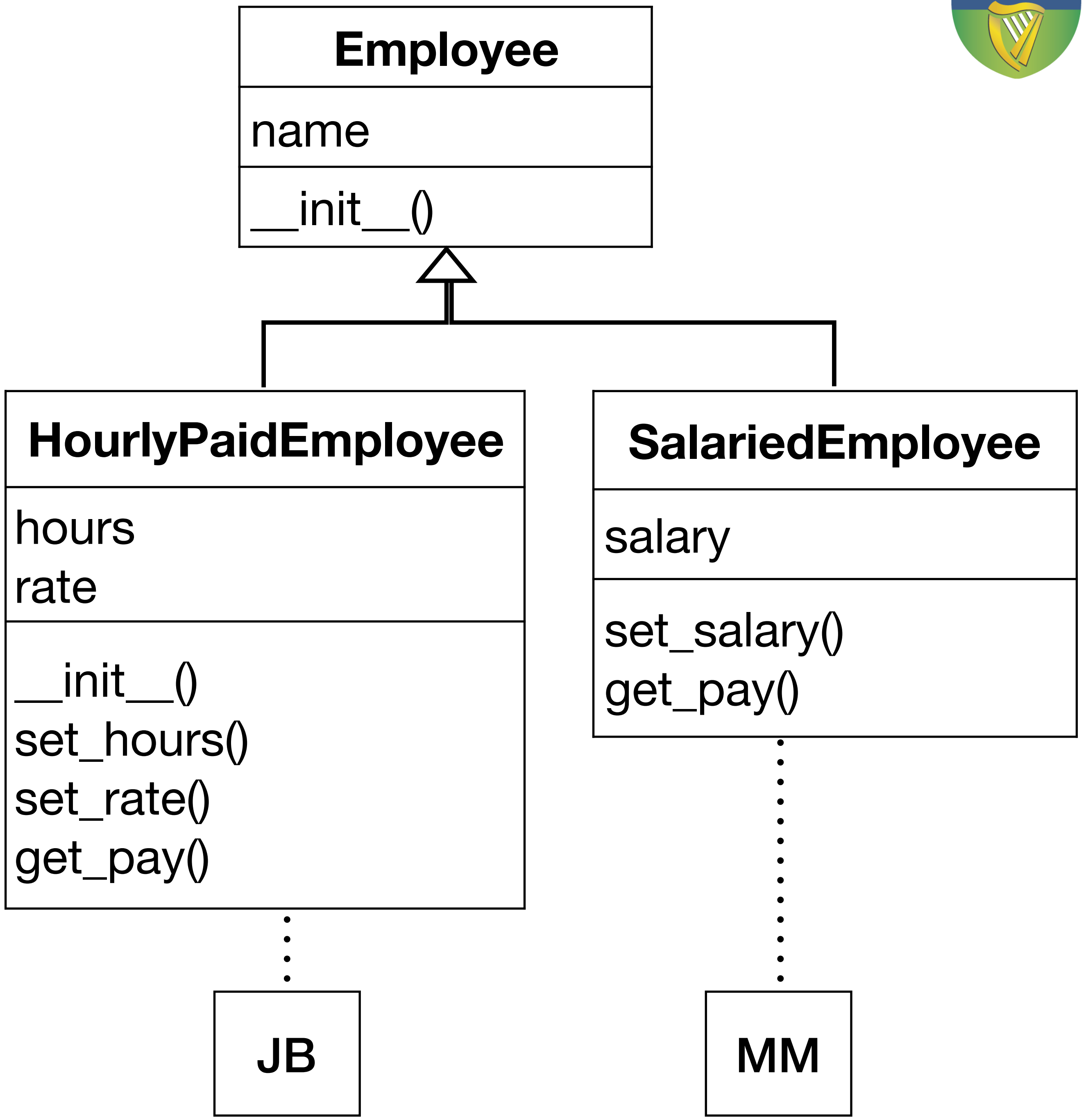
- Label class
  - attributes “text”, font, dimensions etc.
- Button class
  - is-a Label
  - specialisation of label
  - on_click() method
  - extra attribute status {On, Of}



# Inheritance

```
class Employee(
    def __init_
        self.na

class HourlyPai

    def __init_
        Employee.__init__(self, name)
        self.hours = 0
        self.rate =

    def set_hours(self, hours):
        self.hours = hours

    def set_rate(self, r):
        self.rate = r

    def get_pay(self):
        return self.rate * self.hours

class SalariedEmployee(Employee):

    def set_salary(self, sal):
        self.salary = sal

    def get_pay(self):
        return self.salary / 12
```

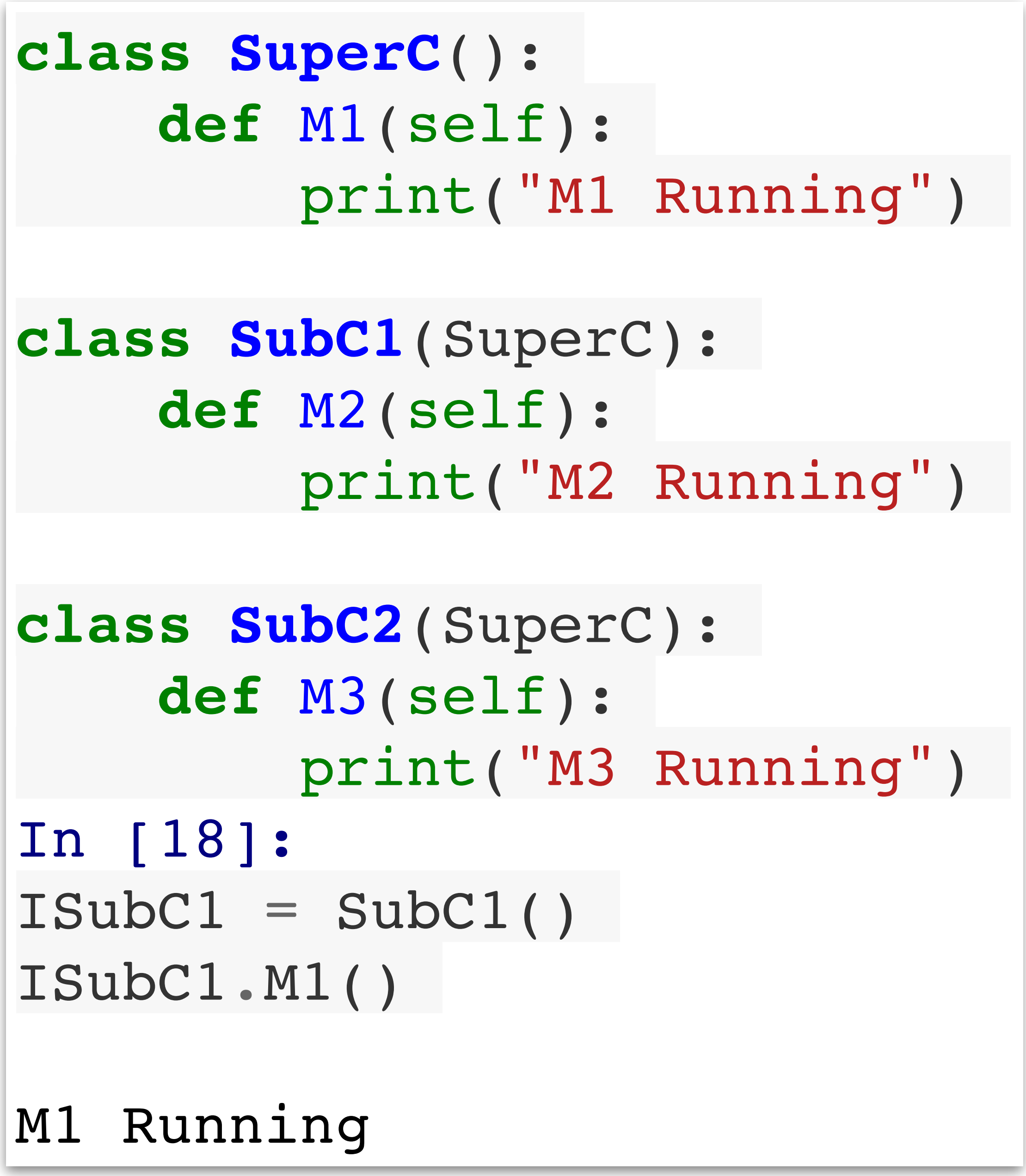
HourlyPaidEmployee & SalariedEmployee are **subclasses** of Employee They inherit data & methods from their **superclass**



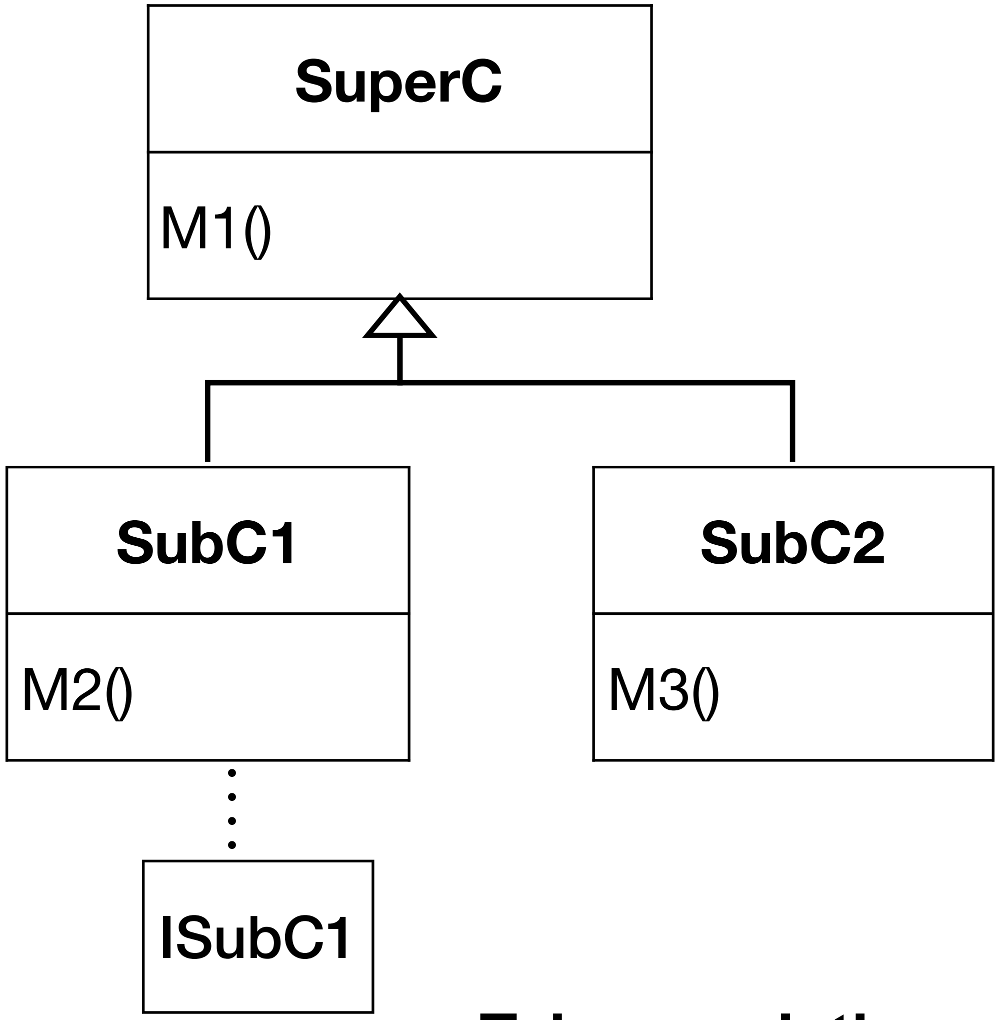
```
JB = HourlyPaidEmployee("Joe Bloggs")
MM = SalariedEmployee("Marvelous Mary")
JB.set_hours(121)
JB.set_rate(10.50)
MM.set_salary(45000)
```

# Inheriting Methods

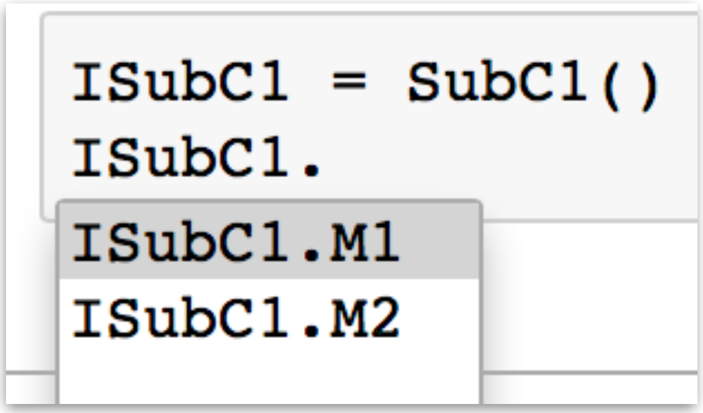
- M1 Inherited from superclass



```
class SuperC():
    def M1(self):
        print("M1 Running")

class SubC1(SuperC):
    def M2(self):
        print("M2 Running")

class SubC2(SuperC):
    def M3(self):
        print("M3 Running")
In [18]:
ISubC1 = SubC1()
ISubC1.M1()

M1 Running
```



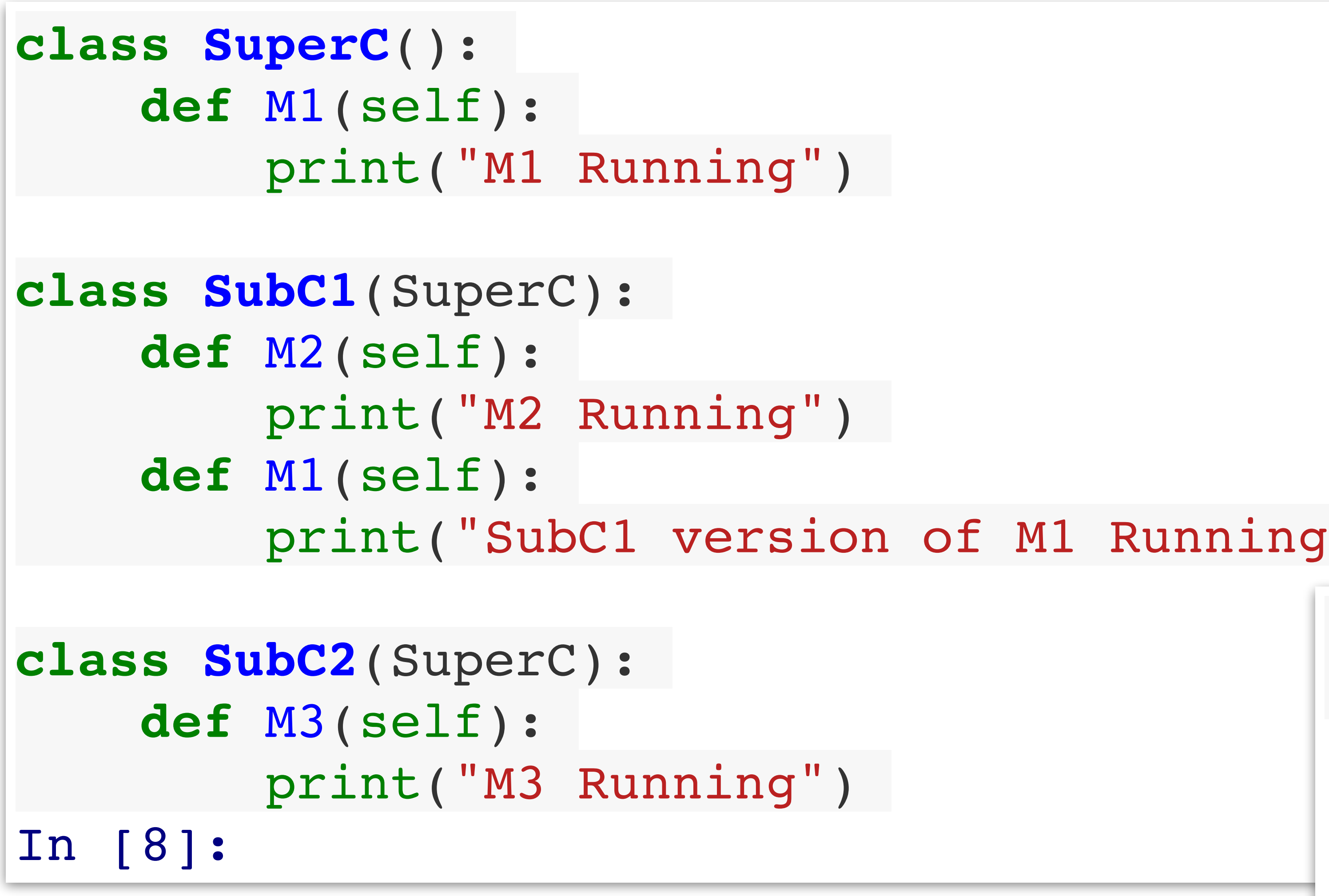
Tab completion



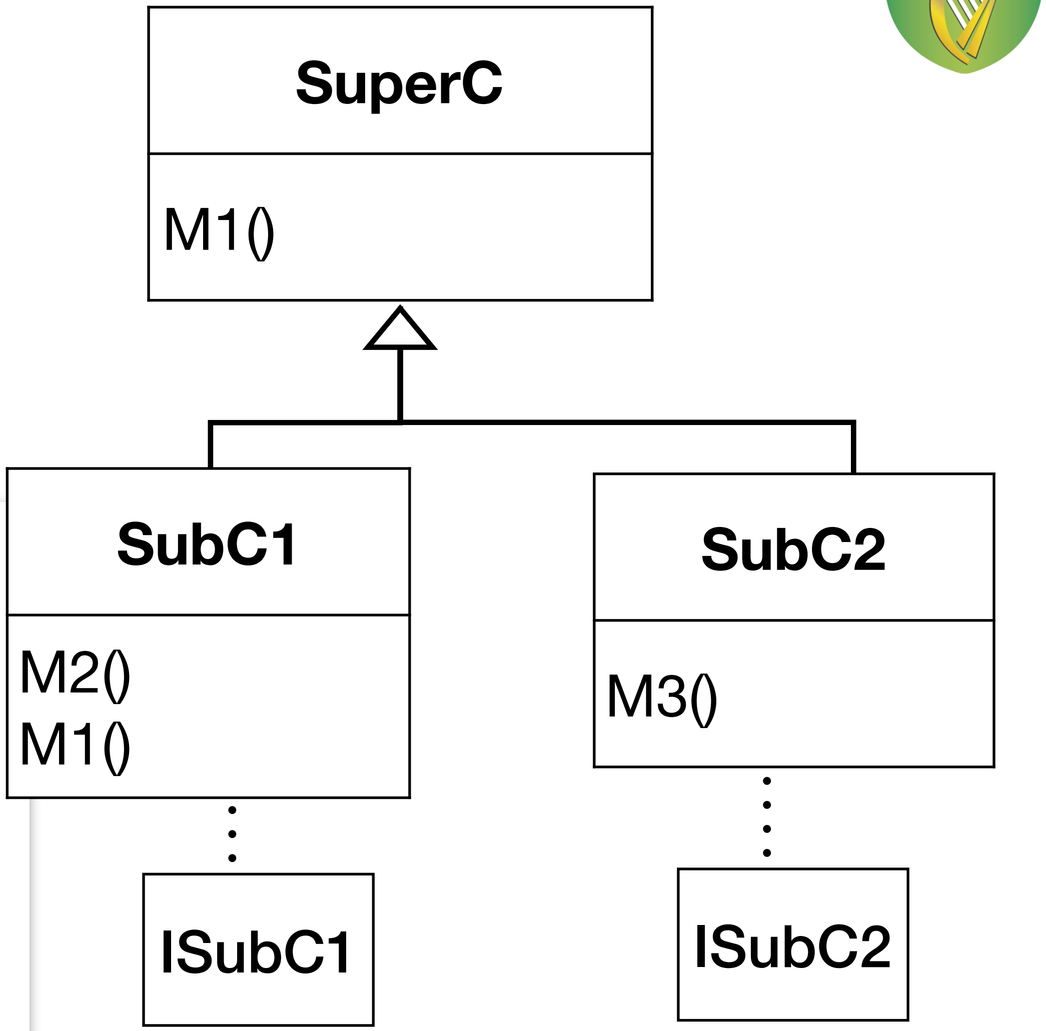
```
ISubC1 = SubC1()
ISubC1.
ISubC1.M1
ISubC1.M2
```

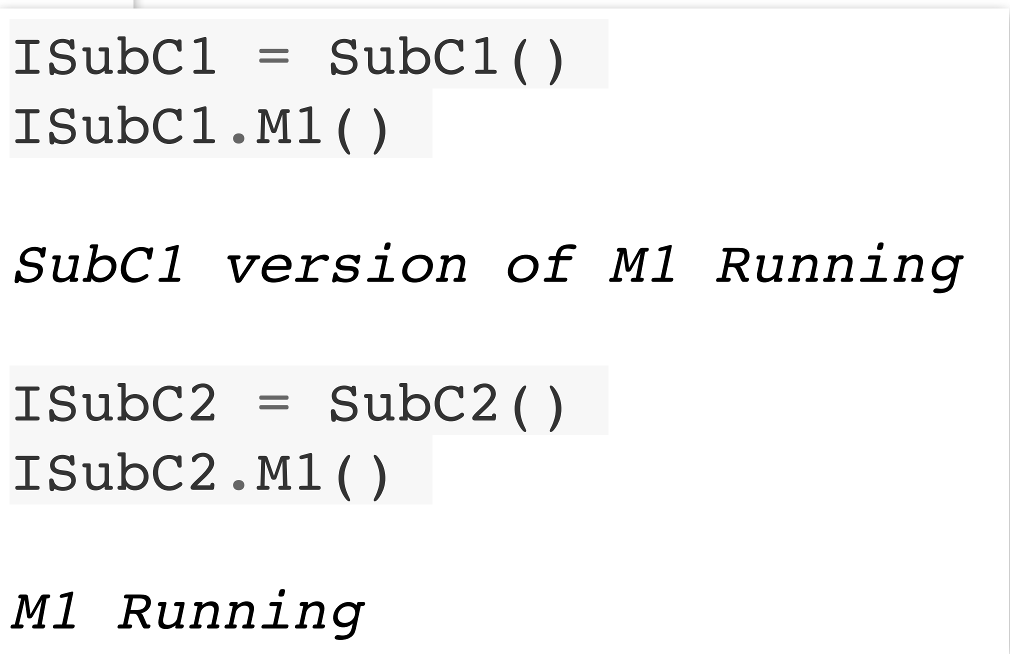
# Overriding Methods

- SubC1 overrides definition of M1
- M1 is now ***polymorphic***
  - □ *lit. “many meanings”*



```
class SuperC():
    def M1(self):
        print("M1 Running")

class SubC1(SuperC):
    def M2(self):
        print("M2 Running")
    def M1(self):
        print("SubC1 version of M1 Running")

class SubC2(SuperC):
    def M3(self):
        print("M3 Running")
In [8]:
```

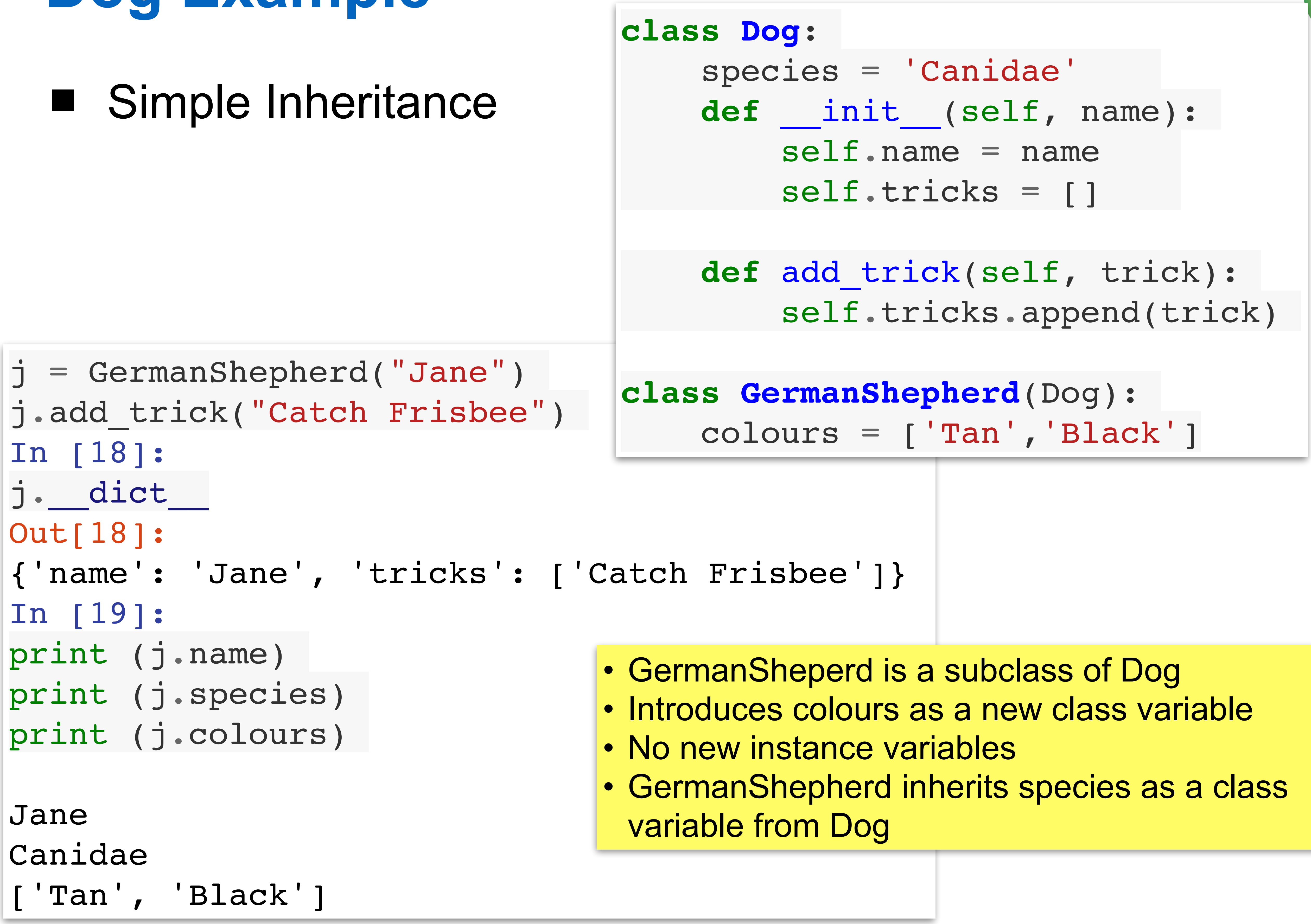




```
ISubC1 = SubC1()
ISubC1.M1()

SubC1 version of M1 Running

ISubC2 = SubC2()
ISubC2.M1()

M1 Running
```

# Dog Example

- Simple Inheritance



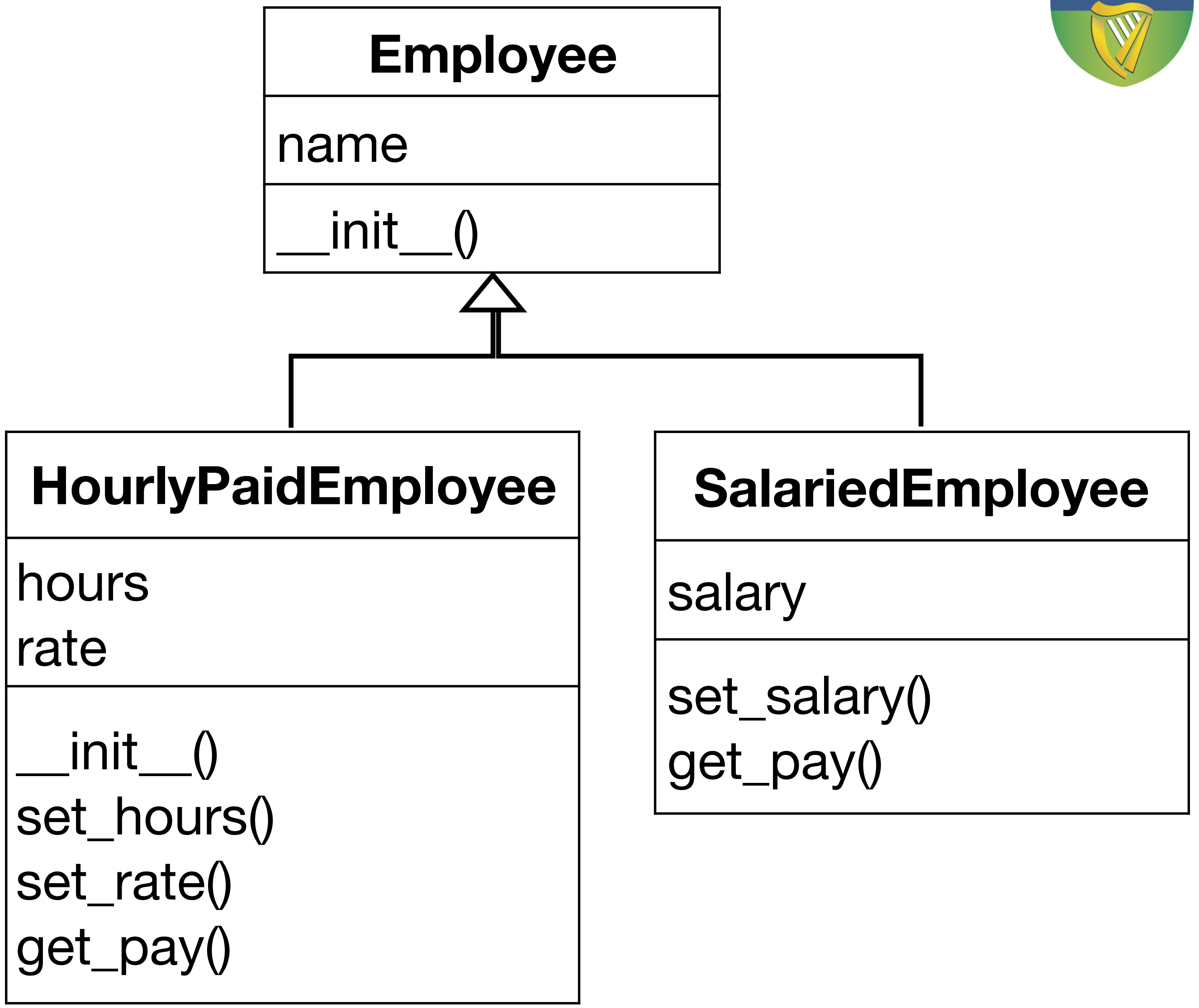
```
class Dog:
    species = 'Canidae'
    def __init__(self, name):
        self.name = name
        self.tricks = []

    def add_trick(self, trick):
        self.tricks.append(trick)

class GermanShepherd(Dog):
    colours = ['Tan','Black']
```

```
j = GermanShepherd("Jane")
j.add_trick("Catch Frisbee")
In [18]:
j.__dict__
Out[18]:
{'name': 'Jane', 'tricks': ['Catch Frisbee']}
In [19]:
print (j.name)
print (j.species)
print (j.colours)

Jane
Canidae
['Tan', 'Black']
```

- GermanSheperd is a subclass of Dog
- Introduces colours as a new class variable
- No new instance variables
- GermanShepherd inherits species as a class variable from Dog

# `__init__` Methods

- Constructor method
  - □ called when an instance created
- Employee example
  - □ 2 `__init__` methods
  - □ HourlyPaidEmployee uses its own (***override***)
  - □ SalariedEmployee ***inherits*** form Employee
- Options:
  - □ use own
  - □ use super class
  - □ use both



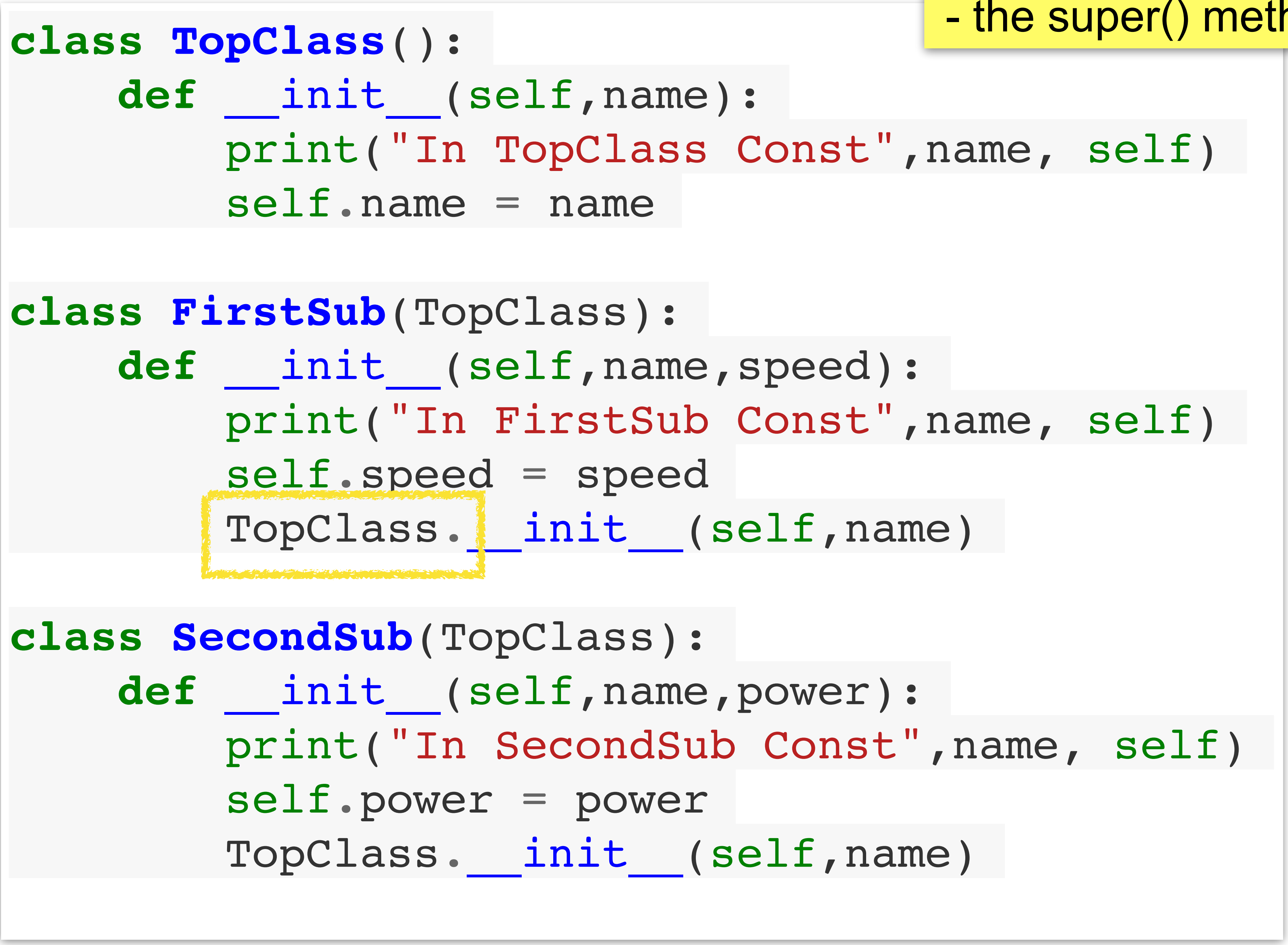
# `__init__` Method Options

- Use own `__init__` method
  - straightforward: superclass inits (if any) will be overridden
- Use superclass
  - also straightforward `__init__` method
- Use both
  - Why?
    - Common init code shared among subclasses
  - How?
    - Subclass init calls superclass init
      - pass on ‘self’ handle
    - But it can get complicated, especially with
      - many levels of inheritance
      - multiple inheritance

# __init__ Method Options

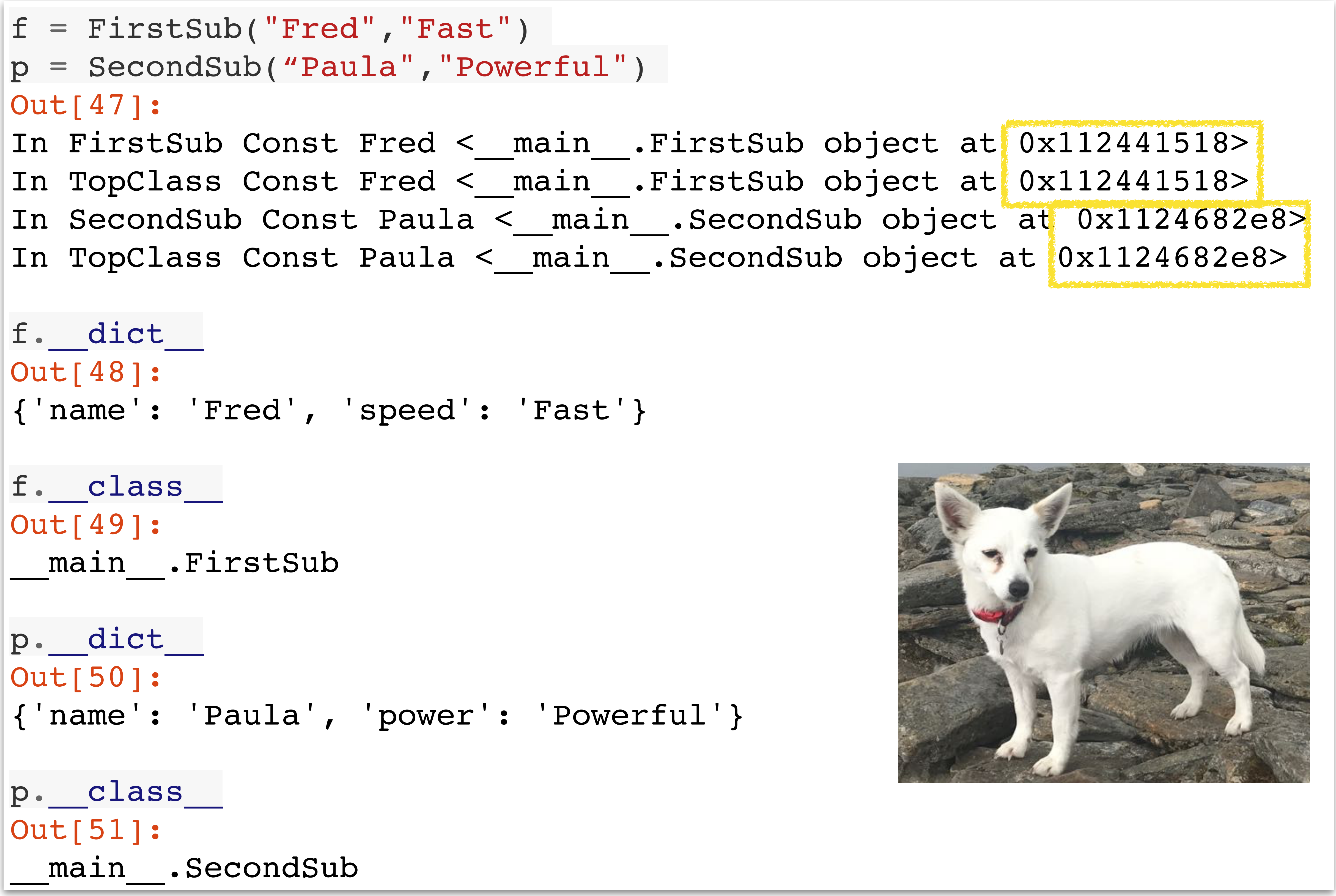
- Using both
  - □ ‘self’ gets passed along

**Note:**
Python has an alternative syntax for accessing the super class
 - the super() method



```
class TopClass():
    def __init__(self,name):
        print("In TopClass Const",name, self)
        self.name = name

class FirstSub(TopClass):
    def __init__(self,name,speed):
        print("In FirstSub Const",name, self)
        self.speed = speed
        TopClass.__init__(self,name)

class SecondSub(TopClass):
    def __init__(self,name,power):
        print("In SecondSub Const",name, self)
        self.power = power
        TopClass.__init__(self,name)
```

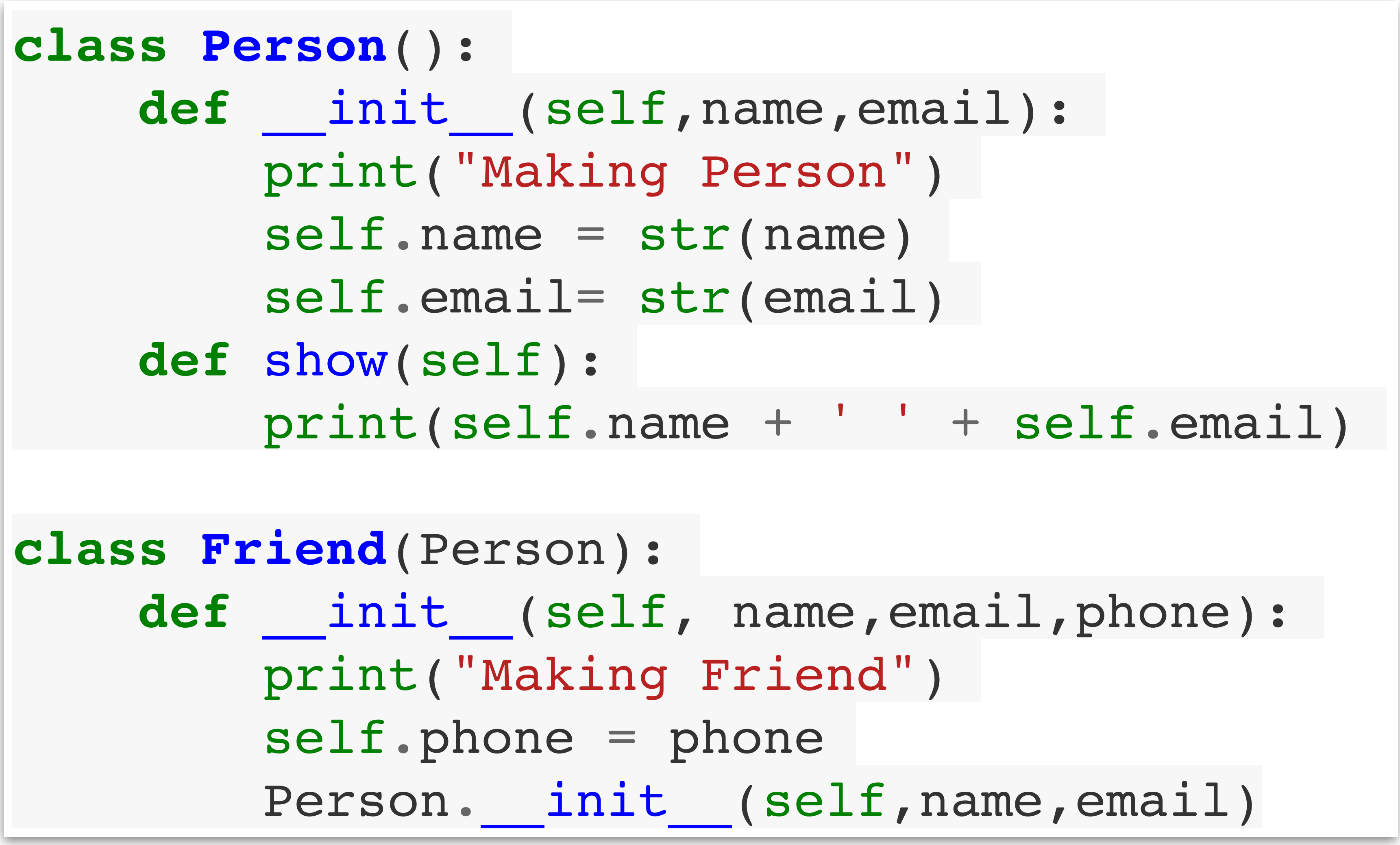
# __init__ Method Options



```
f = FirstSub("Fred","Fast")
p = SecondSub(“Paula","Powerful")
Out[47]:
In FirstSub Const Fred <__main__.FirstSub object at 0x112441518>
In TopClass Const Fred <__main__.FirstSub object at 0x112441518>
In SecondSub Const Paula <__main__.SecondSub object at 0x1124682e8>
In TopClass Const Paula <__main__.SecondSub object at 0x1124682e8>

f.__dict__
Out[48]:
{'name': 'Fred', 'speed': 'Fast'}

f.__class__
Out[49]:
__main__.FirstSub

p.__dict__
Out[50]:
{'name': 'Paula', 'power': 'Powerful'}

p.__class__
Out[51]:
__main__.SecondSub
```

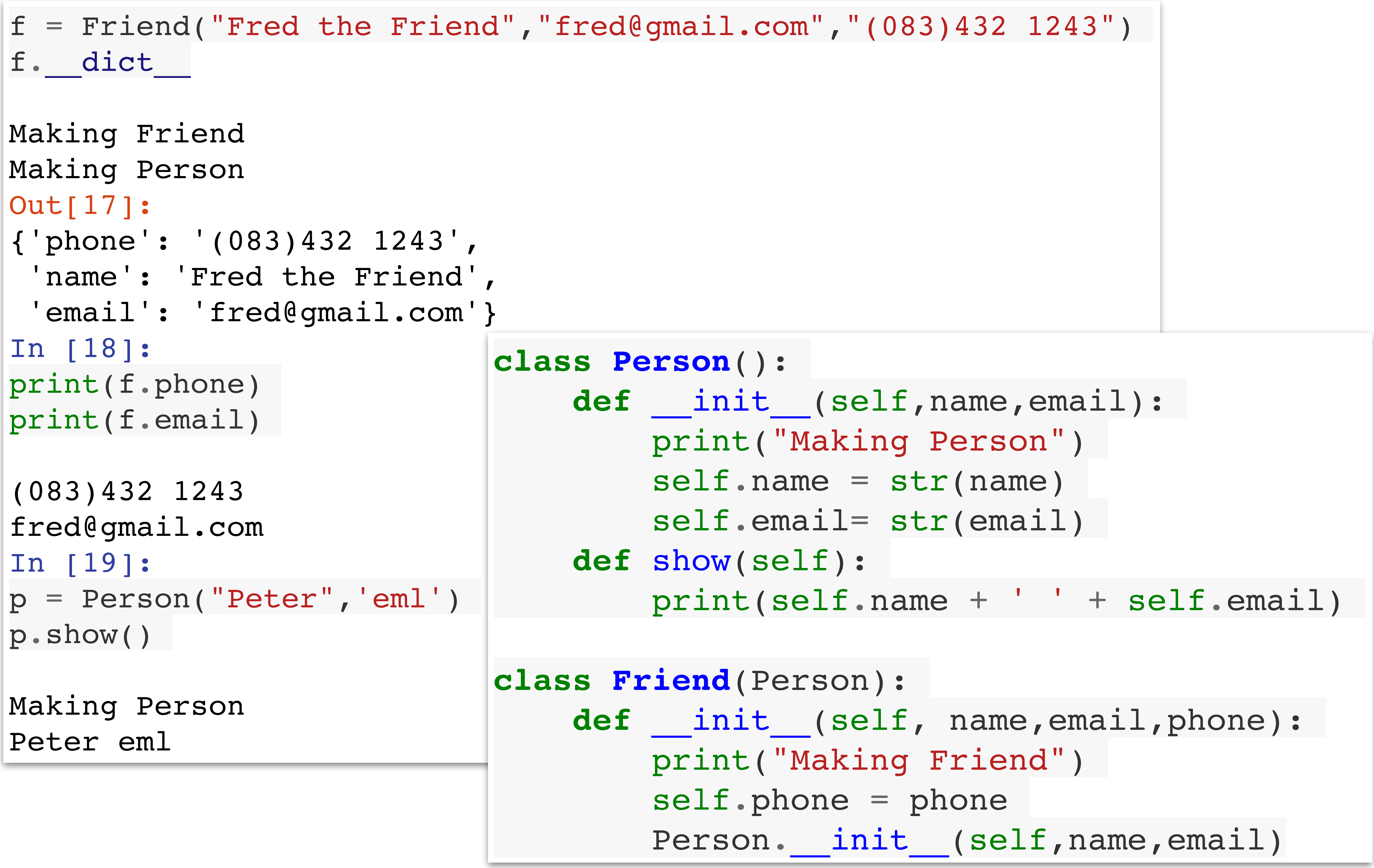
# Friends Example

- Person
    - show and constructor methods
    - name and email attributes
- Friend
    - adds phone attribute
    - constructor (init) calls init from Person
    - show method inherited



```python
class Person():
    def __init__(self,name,email):
        print("Making Person")
        self.name = str(name)
        self.email= str(email)
    def show(self):
        print(self.name + ' ' + self.email)

class Friend(Person):
    def __init__(self, name,email,phone):
        print("Making Friend")
        self.phone = phone
        Person.__init__(self,name,email)
```

# Friends Example

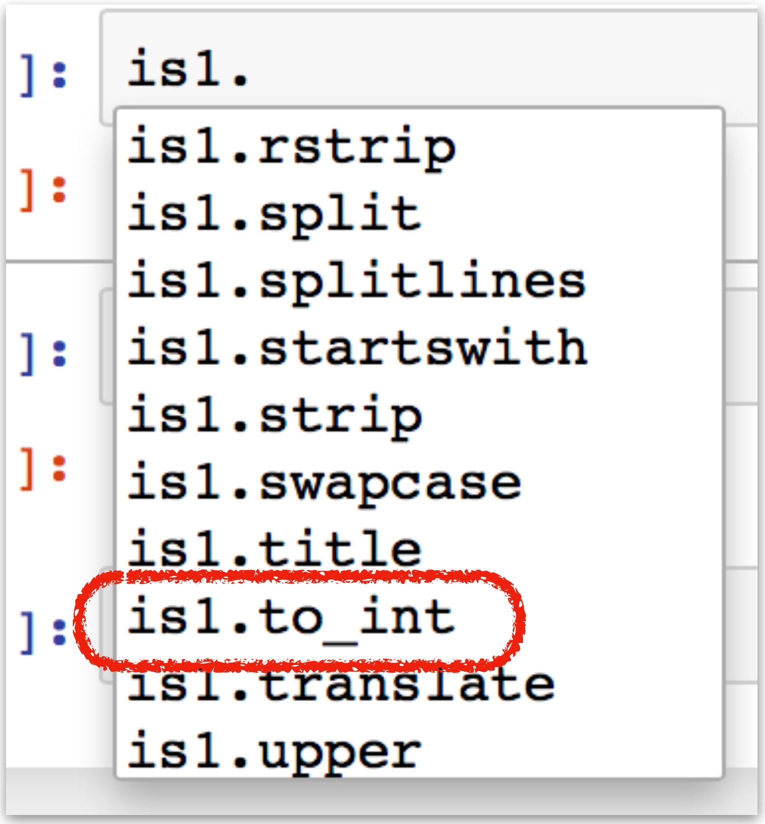


```
f = Friend("Fred the Friend","fred@gmail.com","(083)432 1243")
f.__dict__

Making Friend
Making Person
Out[17]:
{'phone': '(083)432 1243',
 'name': 'Fred the Friend',
 'email': 'fred@gmail.com'}
In [18]:
print(f.phone)
print(f.email)

(083)432 1243
fred@gmail.com
In [19]:
p = Person("Peter",'eml')
p.show()

Making Person
Peter eml
```

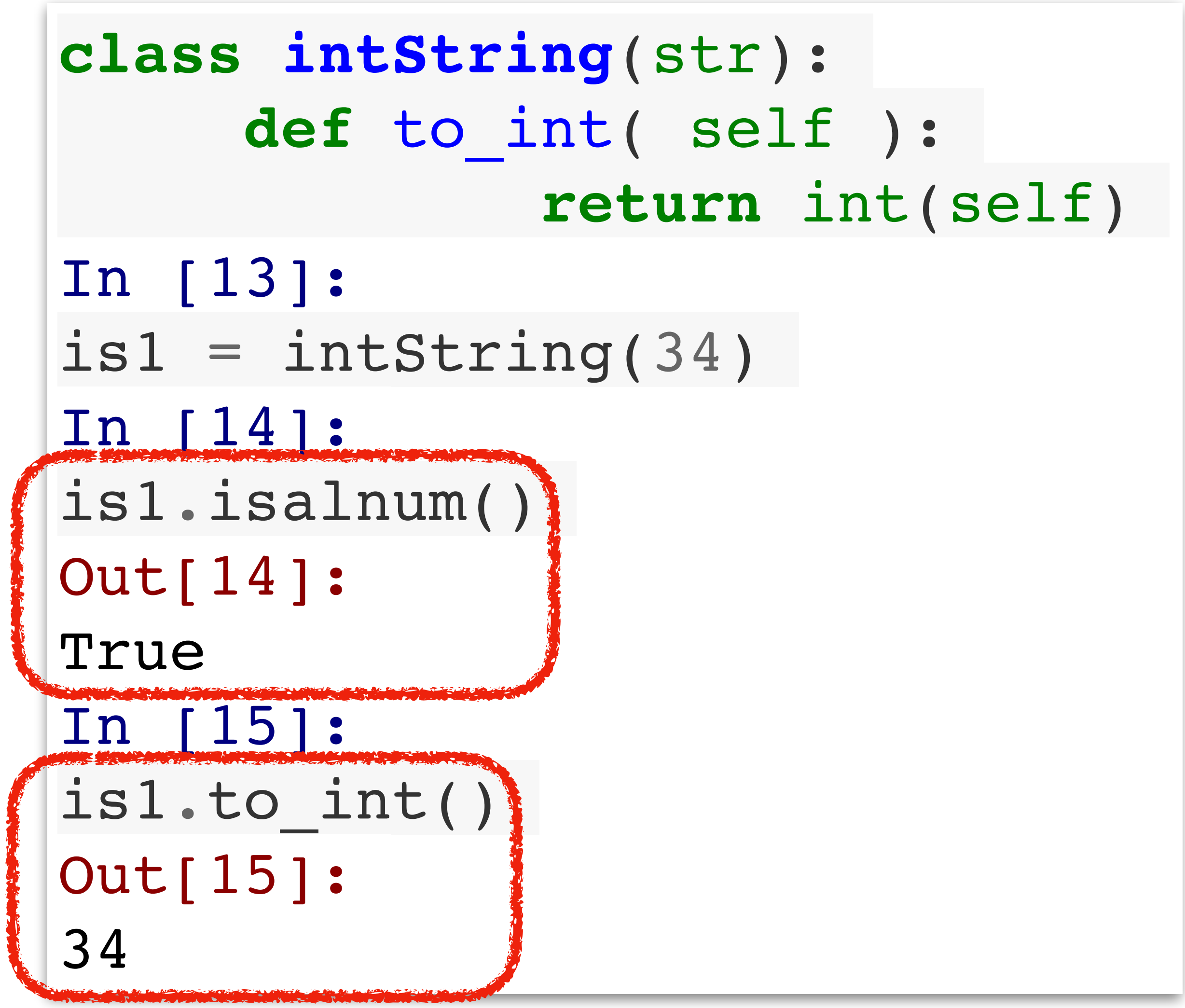
```python
class Person():
    def __init__(self,name,email):
        print("Making Person")
        self.name = str(name)
        self.email= str(email)
    def show(self):
        print(self.name + ' ' + self.email)

class Friend(Person):
    def __init__(self, name,email,phone):
        print("Making Friend")
        self.phone = phone
        Person.__init__(self,name,email)
```

# Extending Built-In Classes

- intString is a sub-class of str
- is1 an instance of intString
- Is1 inherits all str methods



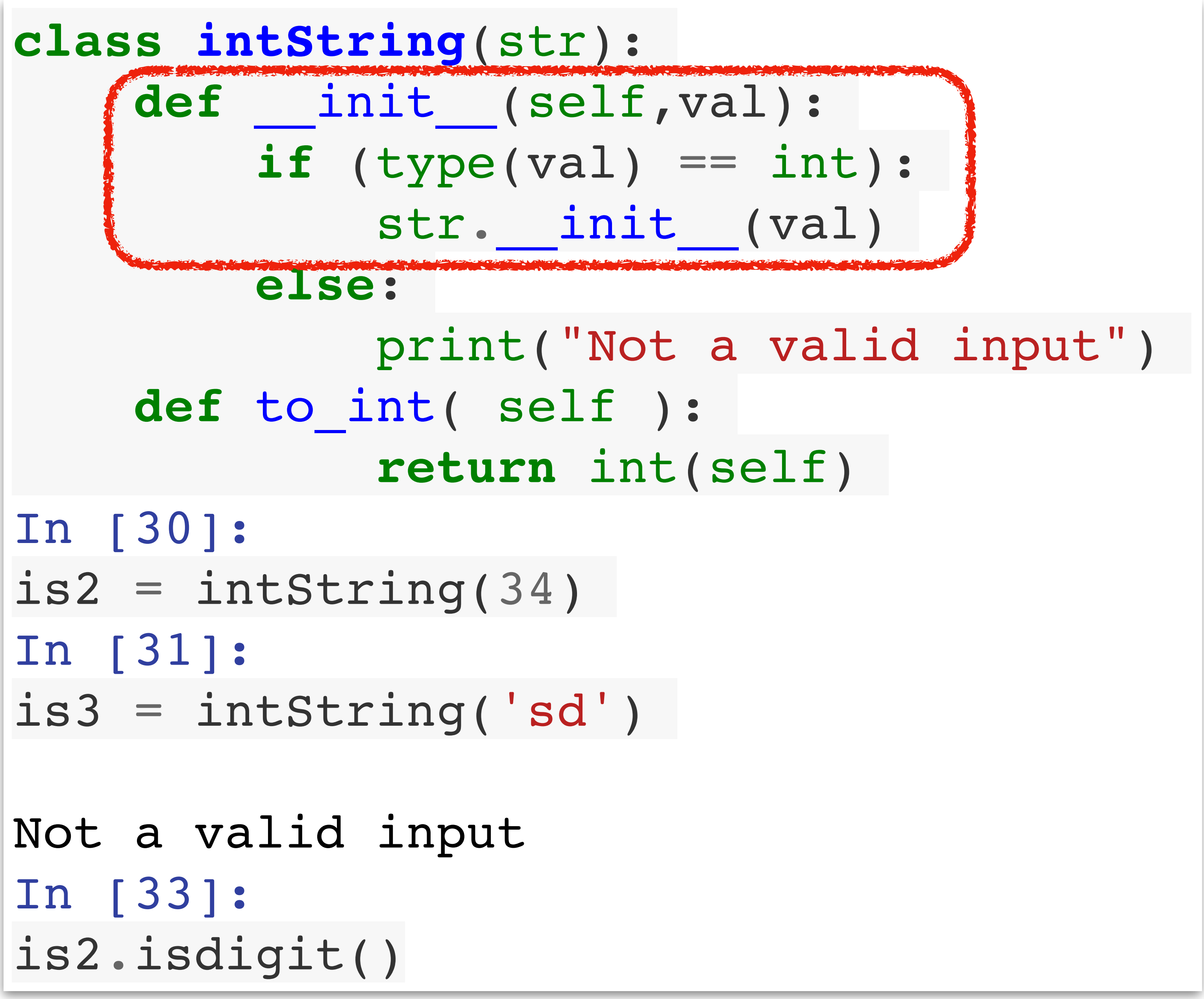
```
]: is1.
   is1.rstrip
   is1.split
   is1.splitlines
   is1.startswith
   is1.strip
   is1.swapcase
   is1.title
   is1.to_int
   is1.translate
   is1.upper
```



```
class intString(str):
    def to_int( self ):
        return int(self)
In [13]:
is1 = intString(34)
In [14]:
is1.isalnum()
Out[14]:
True
In [15]:
is1.to_int()
Out[15]:
34
```

# Extending the str class

- Managing the init process
  - □ **intString** `__init__` calls the `str` `__init__`



```
class intString(str):
  def __init__(self,val):
    if (type(val) == int):
      str.__init__(val)
    else:
      print("Not a valid input")
  def to_int( self ):
      return int(self)
In [30]:
is2 = intString(34)
In [31]:
is3 = intString('sd')

Not a valid input
In [33]:
is2.isdigit()
```

# Exercise: Colleague  - subclass of Person

1. Extend Person with a Colleague subclass
   - The constructor should accept an additional 'office_location' parameter.
2. How would we deal with someone who is both a Colleague and a Friend?

# Simple Inheritance - Summary

- Subclasses ⇔ Specialization
  - □ a German Shepard is a ‘specialized’ kind of dog
  - □ aka ‘extending’
- Inheritance
  - □ **Inherit** methods and data from Superclass or
  - □ **Override** methods and data from Superclass
- Extending built-in classes