Basic Programming

Lesson 06-07

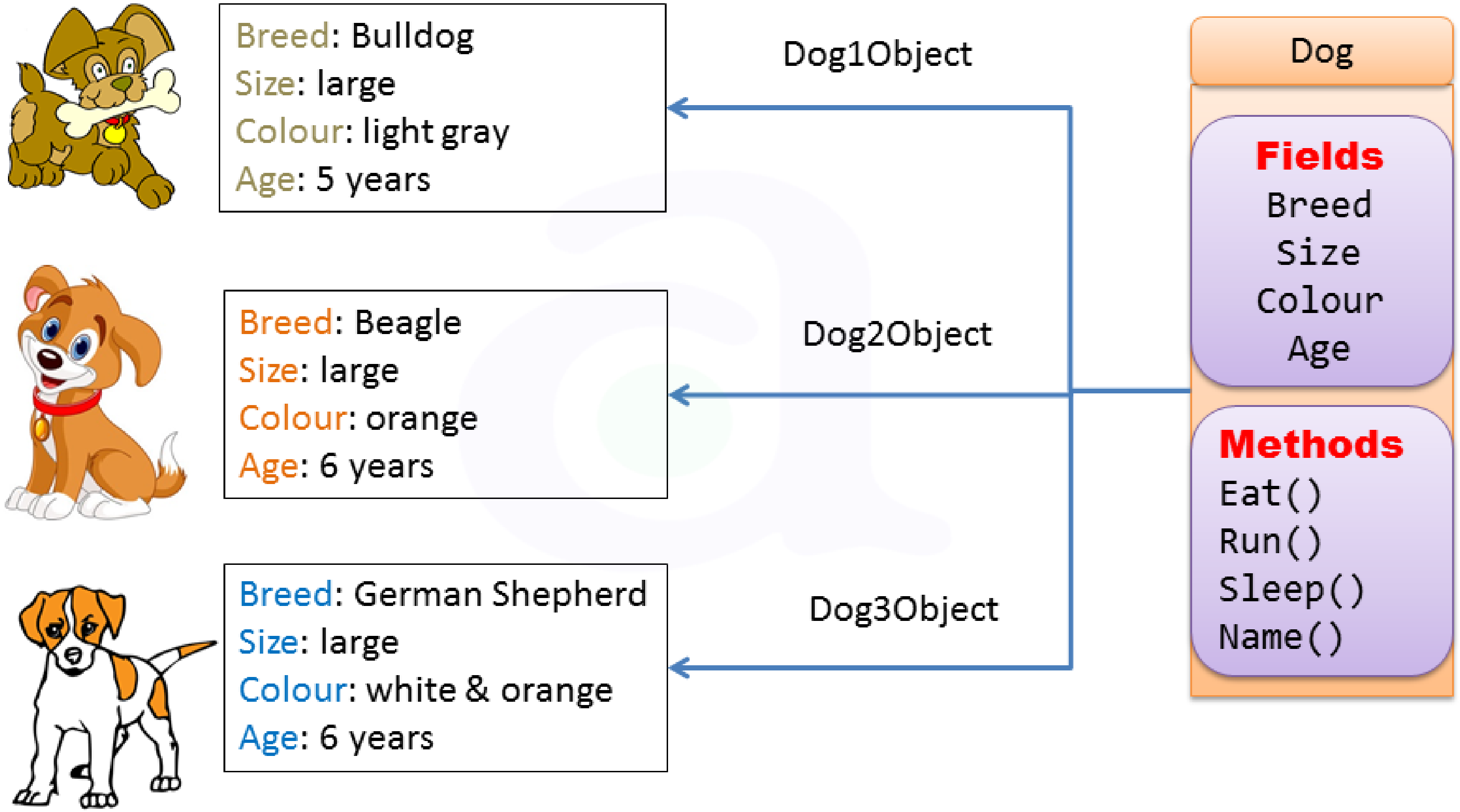
Object Oriented Programming

Encapsulation

Inheritance

Polymorphism

Abstraction



Breed: Bulldog
Size: large
Colour: light gray
Age: 5 years

Breed: Beagle
Size: large
Colour: orange
Age: 6 years

Breed: German Shepherd
Size: large
Colour: white & orange
Age: 6 years

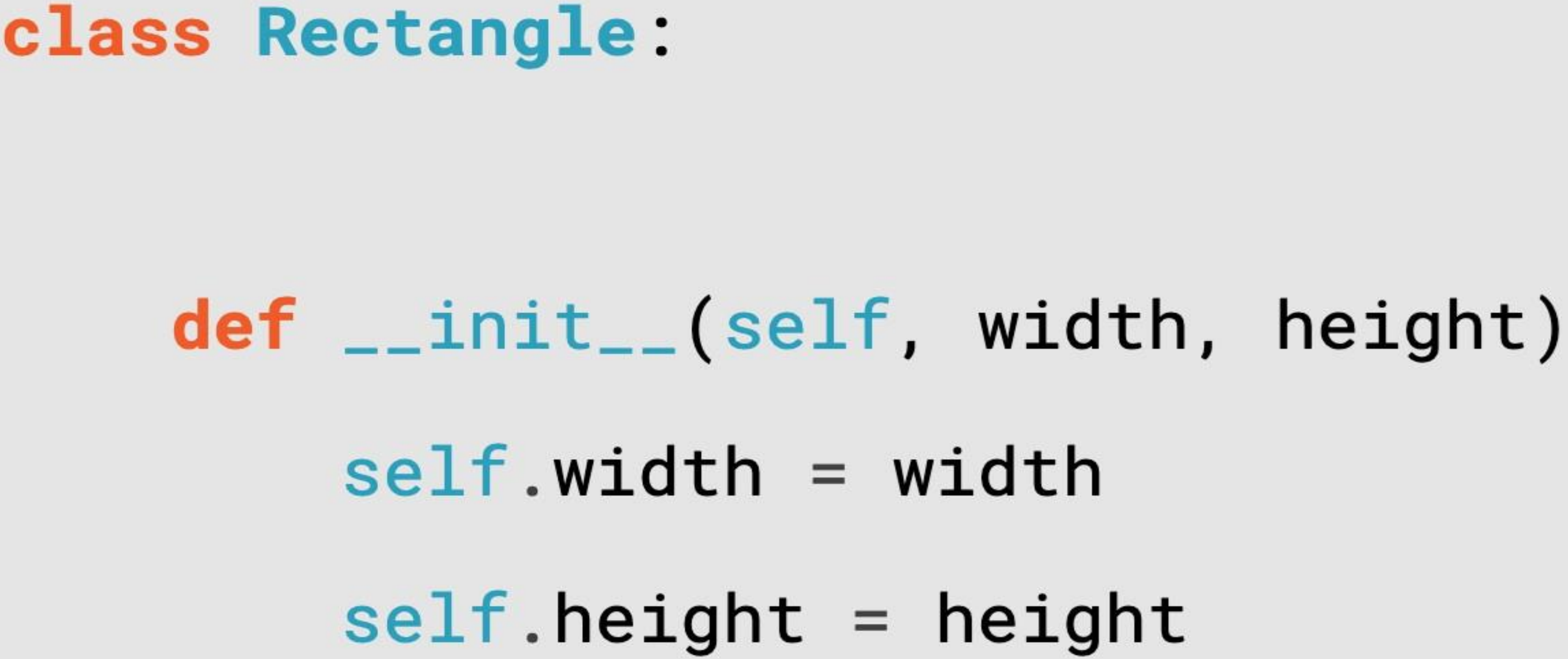
Classes:

```
class Dog:
    pass
```

Objects (Instances):

```
dogObject = Dog()
```

Instance Attributes



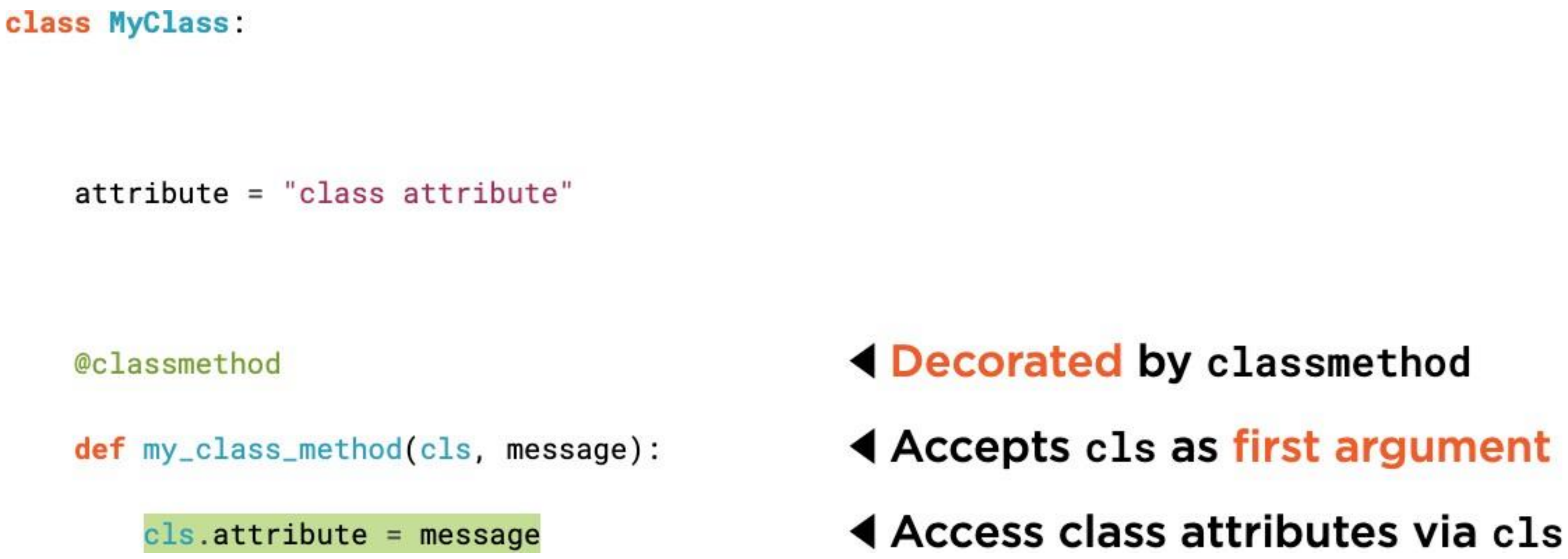
```
class Rectangle:

    def __init__(self, width, height)
        self.width = width
        self.height = height
```

Scopes in Python

Local	Inside the current function
Enclosing	Inside enclosing functions
Global	At the top level of the module
Built-in	In the special builtins module

LEGB

Class Methods



```python
class MyClass:

    attribute = "class attribute"

    @classmethod
    def my_class_method(cls, message):
        cls.attribute = message
```

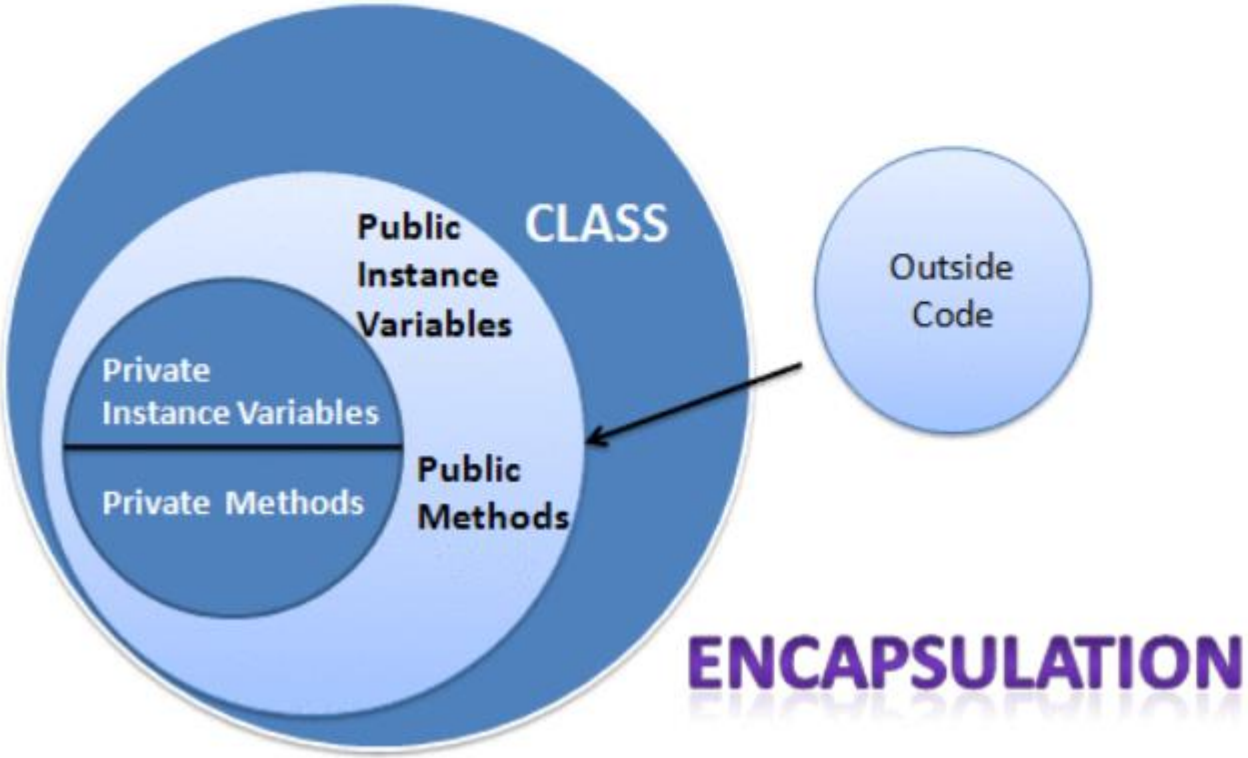
◀ Decorated by `classmethod`

◀ Accepts `cls` as first argument

◀ Access class attributes via `cls`

- cls refers to the class, whereas self refers to the instance. Using the cls keyword, we can only access the members of the class, whereas using the self keyword, we can access both the instance variables and the class attributes.
- With cls, we cannot access the instance variables in a class. cls is passed as an argument to the class method, whereas self is passed as an argument to the instance method. If we initialize variables using self, their scope is the instance’s scope. But, variables initialized with cls have the class as their scope.

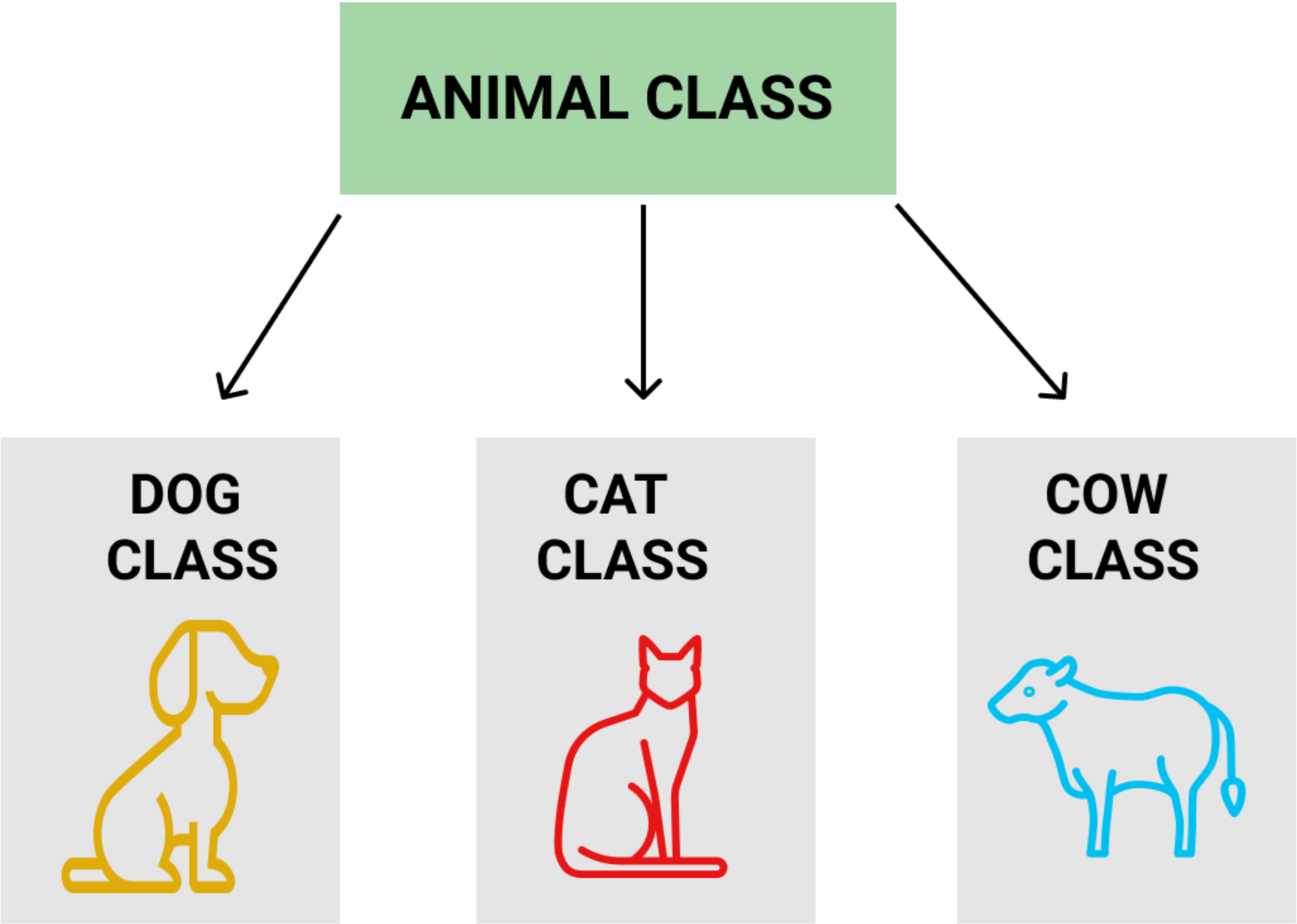
Encapsulation


CLASS
Public Instance Variables
Private Instance Variables
Private Methods
Public Methods
Outside Code
ENCAPSULATION

Getters and Setters Are Not Pythonic

Encapsulate getter and setter methods in properties which behave like attributes

Inheritance



ANIMAL CLASS

DOG CLASS

CAT CLASS

COW CLASS

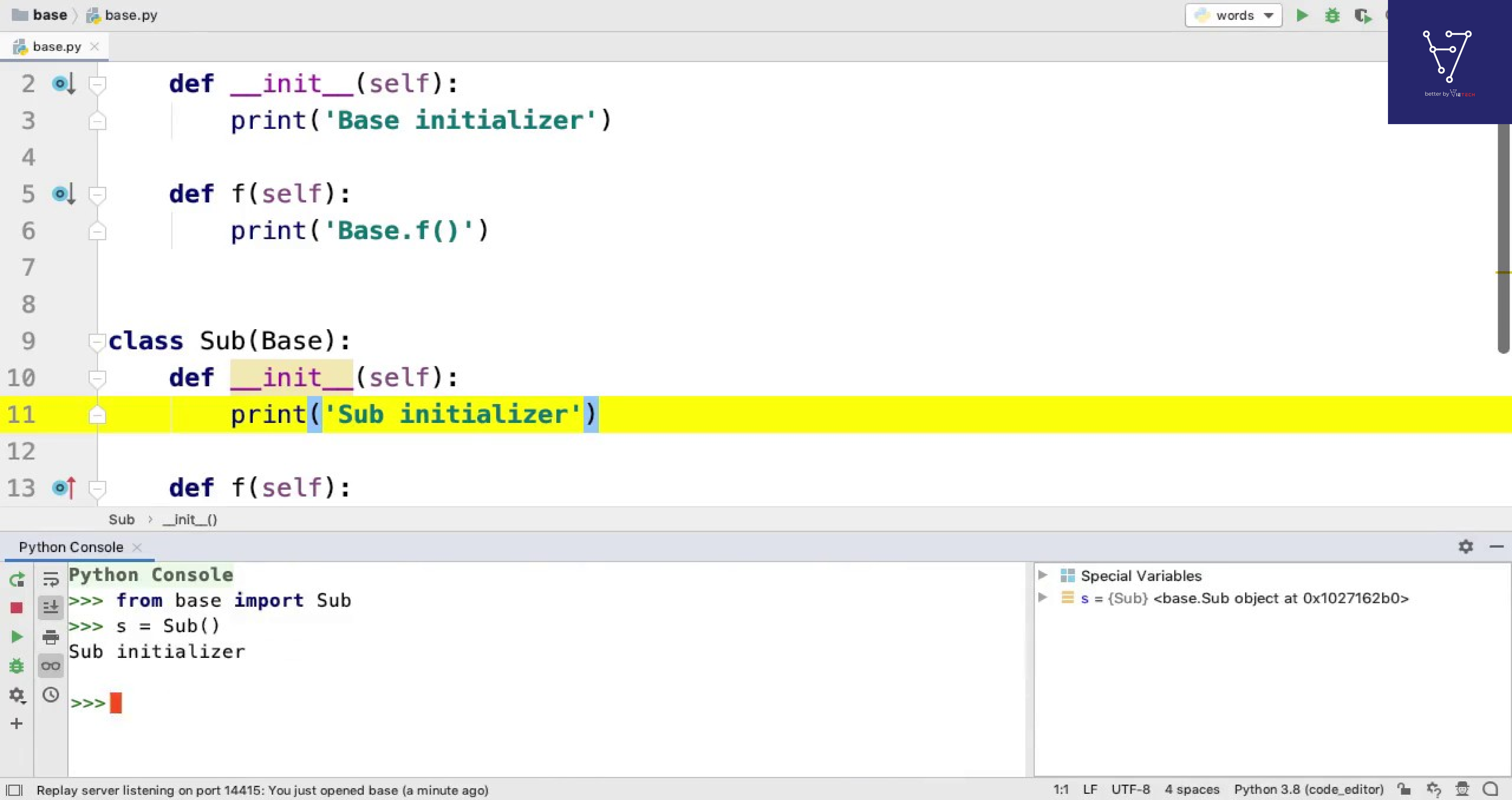
Single Inheritance

Single Inheritance

```
class SubClass(BaseClass):
    . . .
```

Inherits all attributes

May override methods



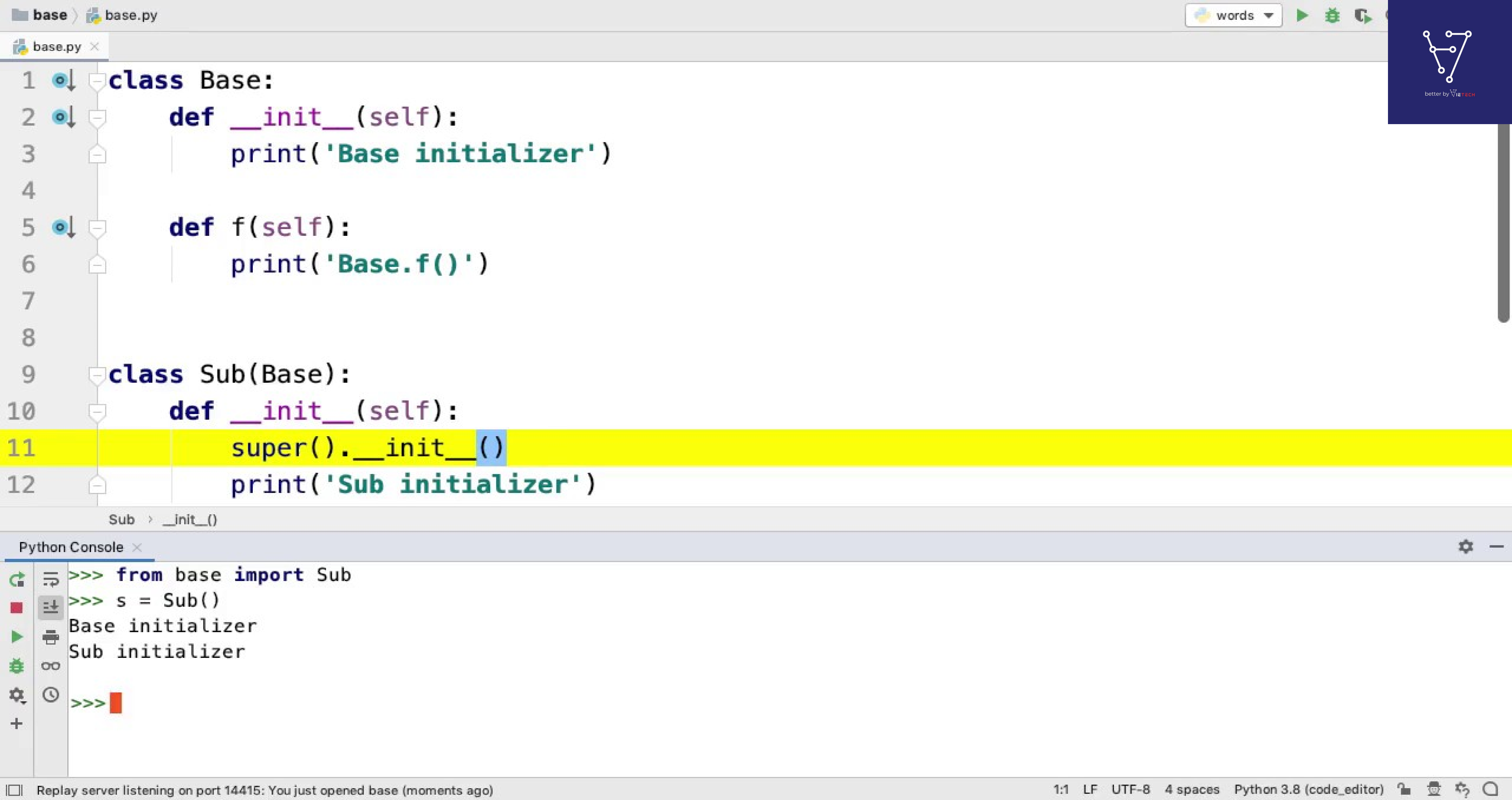
```
    def __init__(self):
        print('Base initializer')

    def f(self):
        print('Base.f()')


class Sub(Base):
    def __init__(self):
        print('Sub initializer')

    def f(self):
```

```
Python Console
>>> from base import Sub
>>> s = Sub()
Sub initializer

>>>
```

Special Variables
s = {Sub} <base.Sub object at 0x1027162b0>

Replay server listening on port 14415: You just opened base (a minute ago)

1:1 LF UTF-8 4 spaces Python 3.8 (code_editor)



```python
class Base:
    def __init__(self):
        print('Base initializer')

    def f(self):
        print('Base.f()')


class Sub(Base):
    def __init__(self):
        super().__init__()
        print('Sub initializer')
```

```
>>> from base import Sub
>>> s = Sub()
Base initializer
Sub initializer

>>>
```

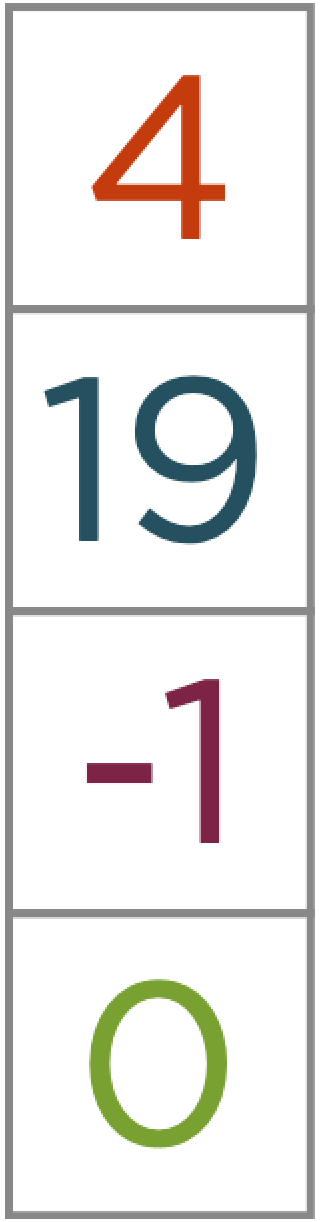
Replay server listening on port 14415: You just opened base (moments ago)

1:1 LF UTF-8 4 spaces Python 3.8 (code_editor)

Type Inspection

Multiple inheritance is not much more complex than single inheritance.

Both rely on a single underlying model.



4

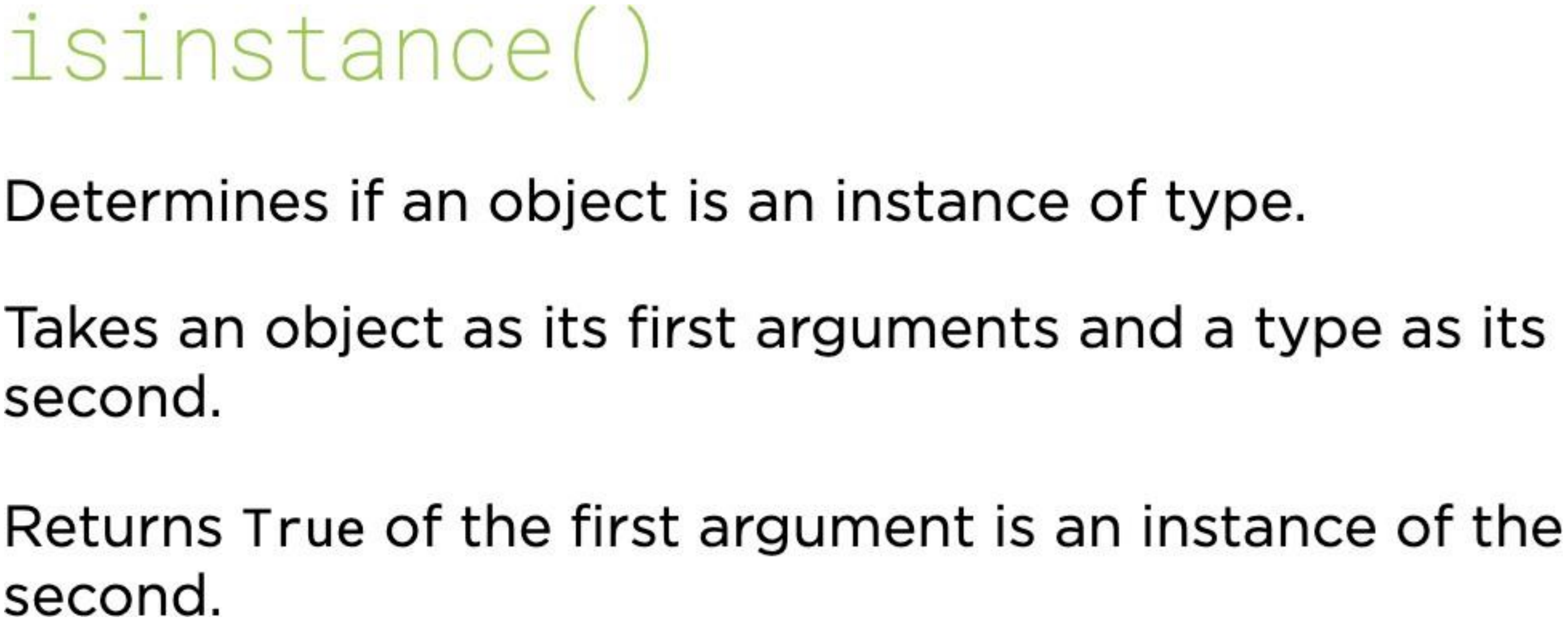
19

-1

0

A new subclass of `SimpleList` called `IntList`

Constrained to containing only integers



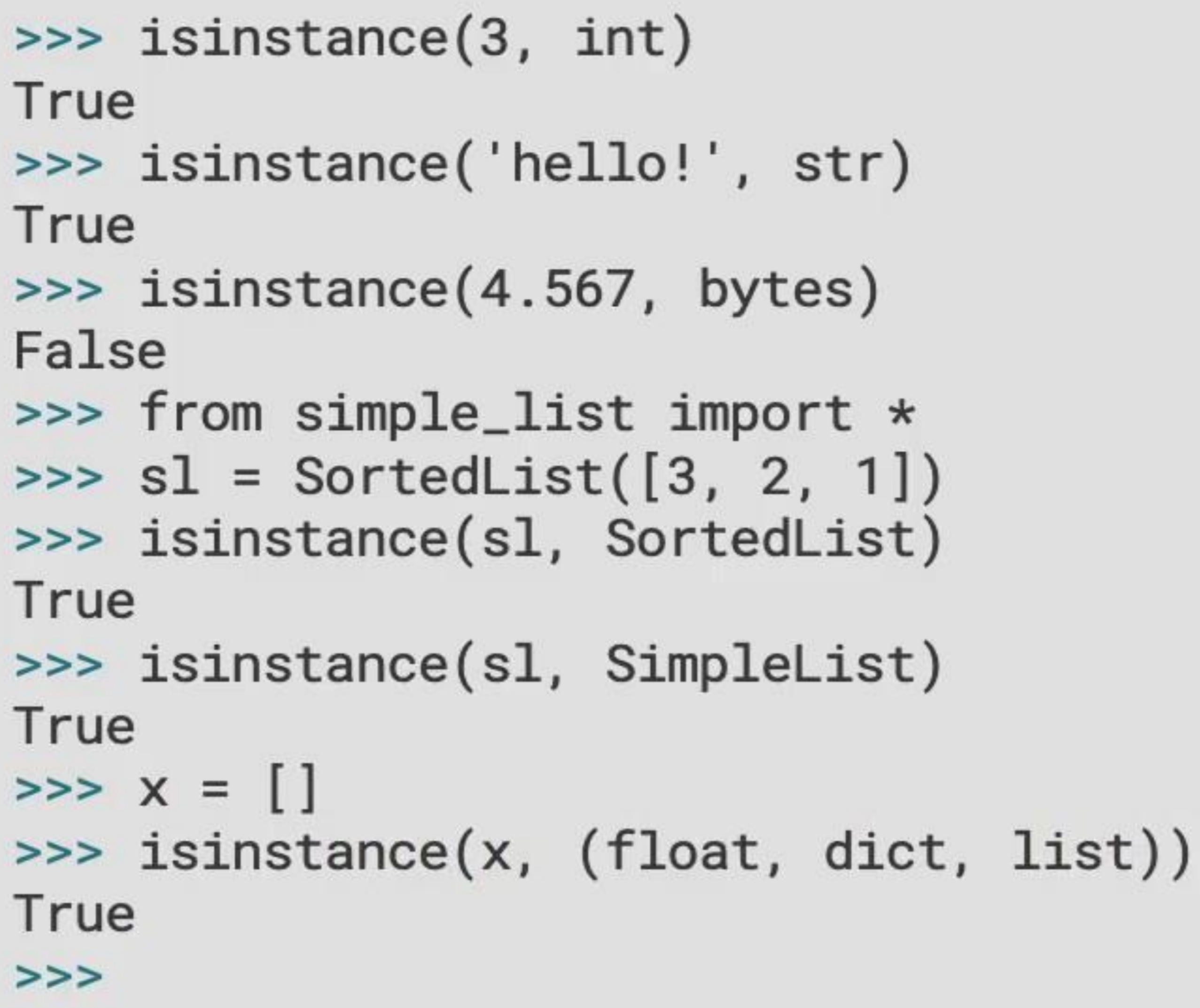
isinstance()

Determines if an object is an instance of type.

Takes an object as its first arguments and a type as its second.

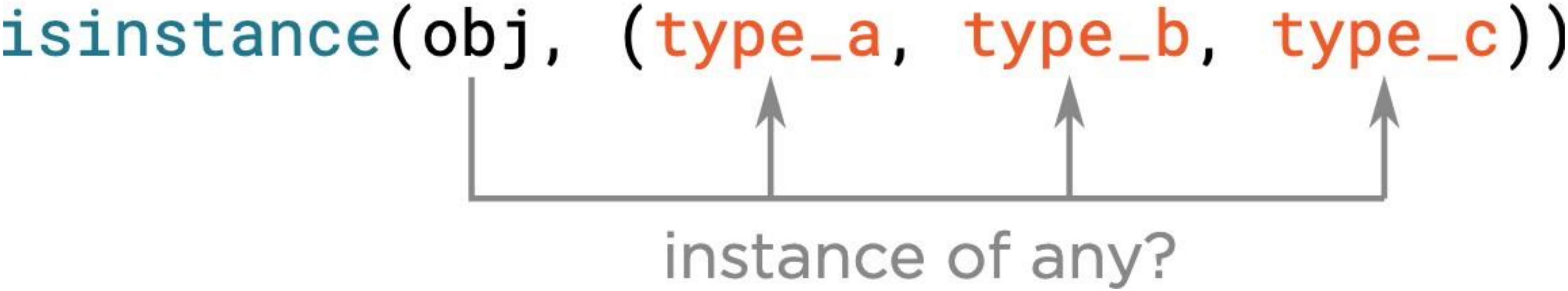
Returns `True` of the first argument is an instance of the second.

isinstance()



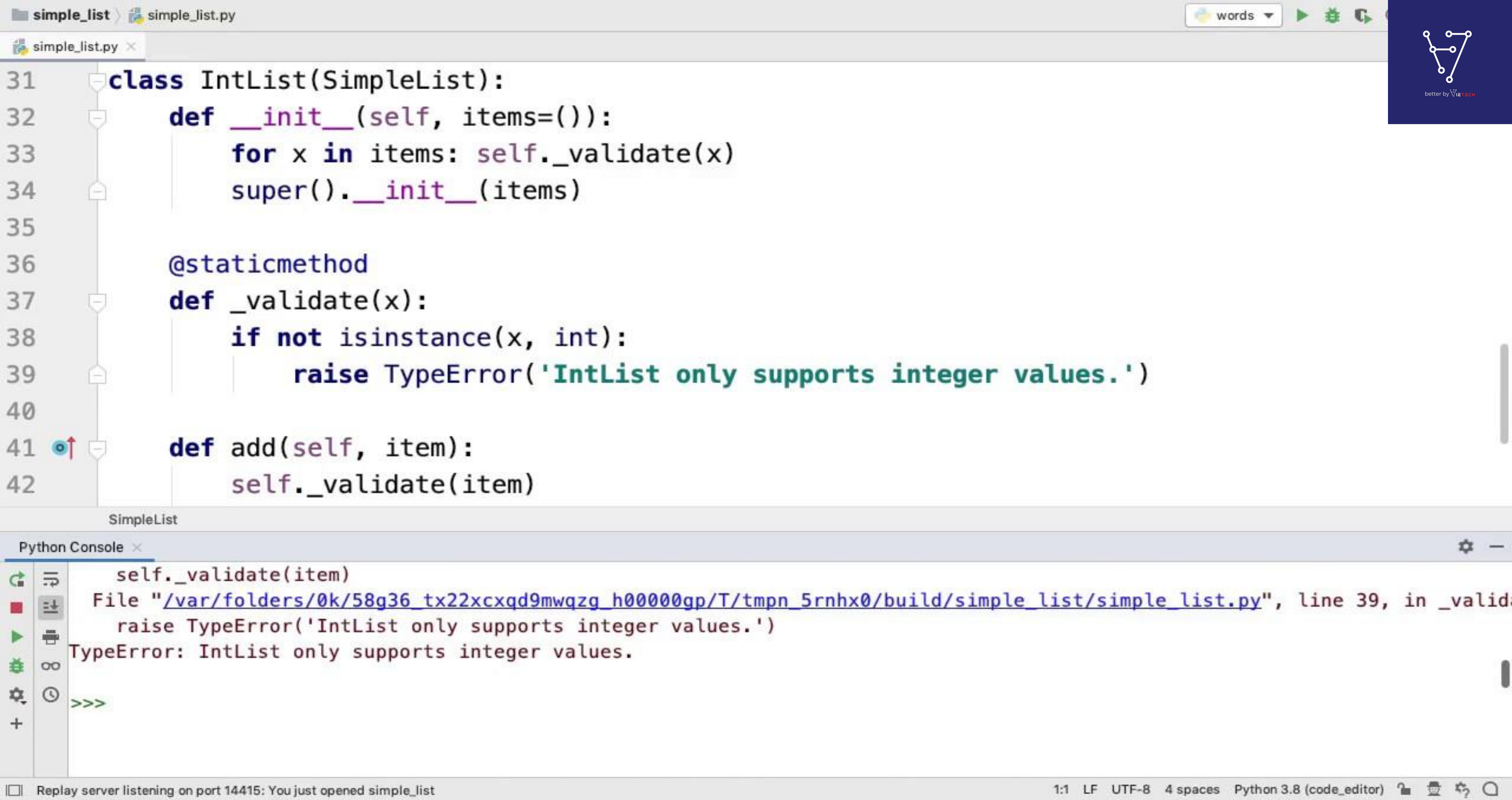
```
>>> isinstance(3, int)
True
>>> isinstance('hello!', str)
True
>>> isinstance(4.567, bytes)
False
>>> from simple_list import *
>>> sl = SortedList([3, 2, 1])
>>> isinstance(sl, SortedList)
True
>>> isinstance(sl, SimpleList)
True
>>> x = []
>>> isinstance(x, (float, dict, list))
True
>>>
```

Checking Multiple Types



```
isinstance(obj, (type_a, type_b, type_c))
```

instance of any?



```python
class IntList(SimpleList):
    def __init__(self, items=()):
        for x in items: self._validate(x)
        super().__init__(items)

    @staticmethod
    def _validate(x):
        if not isinstance(x, int):
            raise TypeError('IntList only supports integer values.')

    def add(self, item):
        self._validate(item)
```

```
    self._validate(item)
  File "/var/folders/0k/58g36_tx22xcxqd9mwqzg_h00000gp/T/tmpn_5rnhx0/build/simple_list/simple_list.py", line 39, in _valid
    raise TypeError('IntList only supports integer values.')
TypeError: IntList only supports integer values.

>>>
```

Replay server listening on port 14415: You just opened simple_list

1:1 LF UTF-8 4 spaces Python 3.8 (code_editor)

Type Checks in Python

`isinstance()` can be used for type checking in Python.

Some people consider type checking a sign of poor design.

Sometimes they're the easiest way to solve a problem.

issubclass()

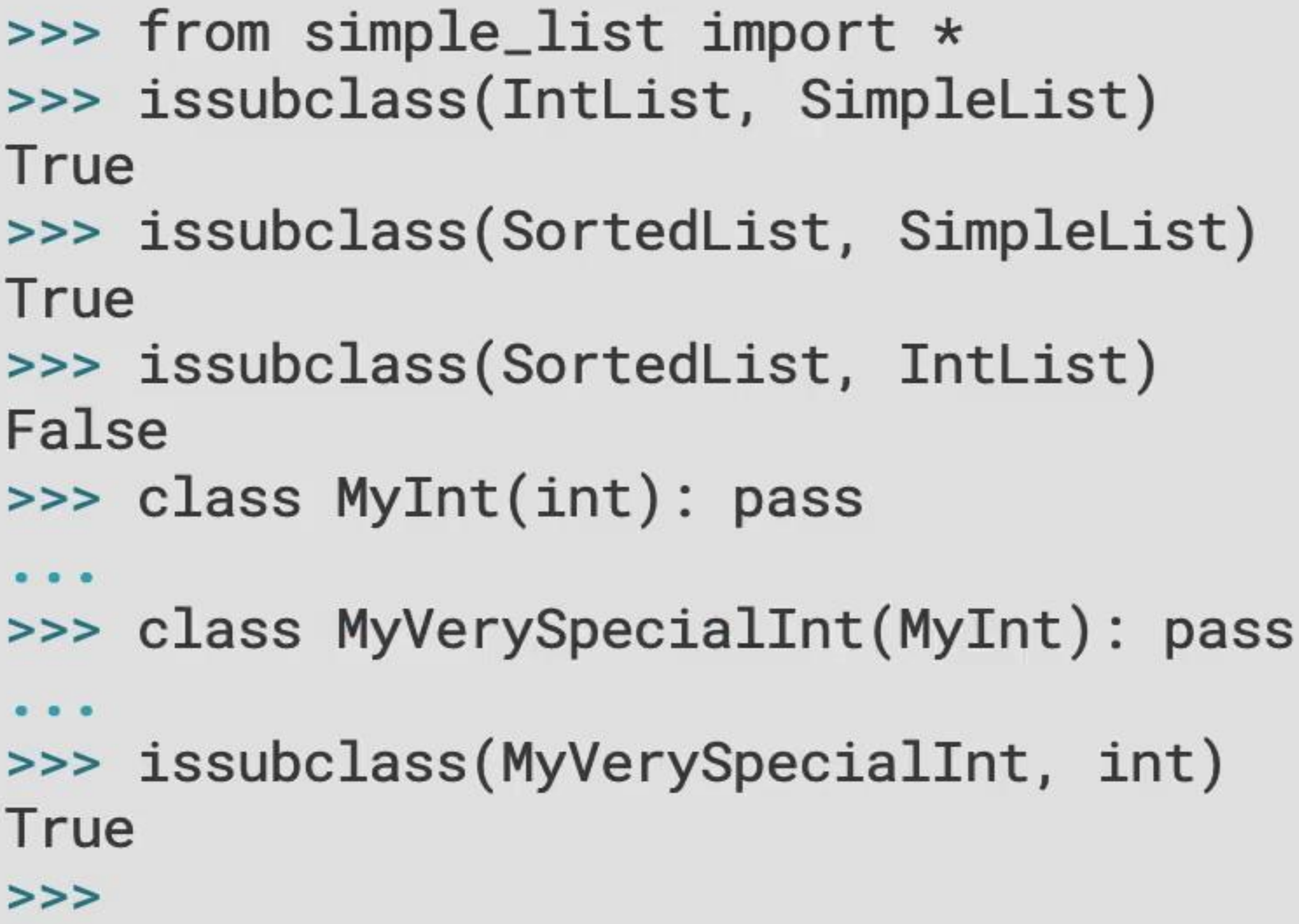
Operates on types to check for sub/superclass relationships.

Determines if one class is a subclass of another.

Takes two arguments, both of which must be types.

Returns `True` if the first argument is a subclass of the second.

issubclass()



```
>>> from simple_list import *
>>> issubclass(IntList, SimpleList)
True
>>> issubclass(SortedList, SimpleList)
True
>>> issubclass(SortedList, IntList)
False
>>> class MyInt(int): pass
...
>>> class MyVerySpecialInt(MyInt): pass
...
>>> issubclass(MyVerySpecialInt, int)
True
>>>
```

Multiple Inheritance

Multiple Inheritance

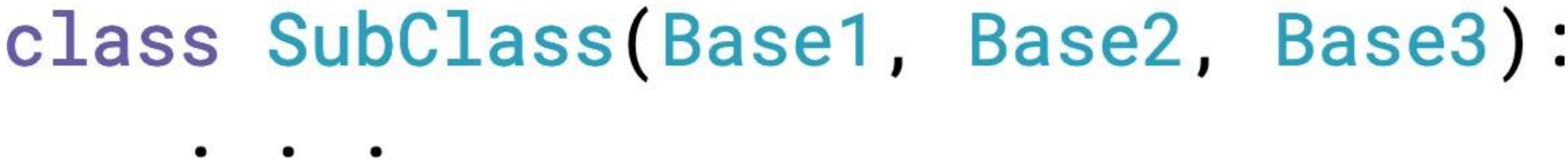
Defining a class with more than one direct base class

Not universal among object-oriented languages

Can lead to certain complexities

Python has a relatively simple system for dealing with them

Multiple Inheritance Syntax



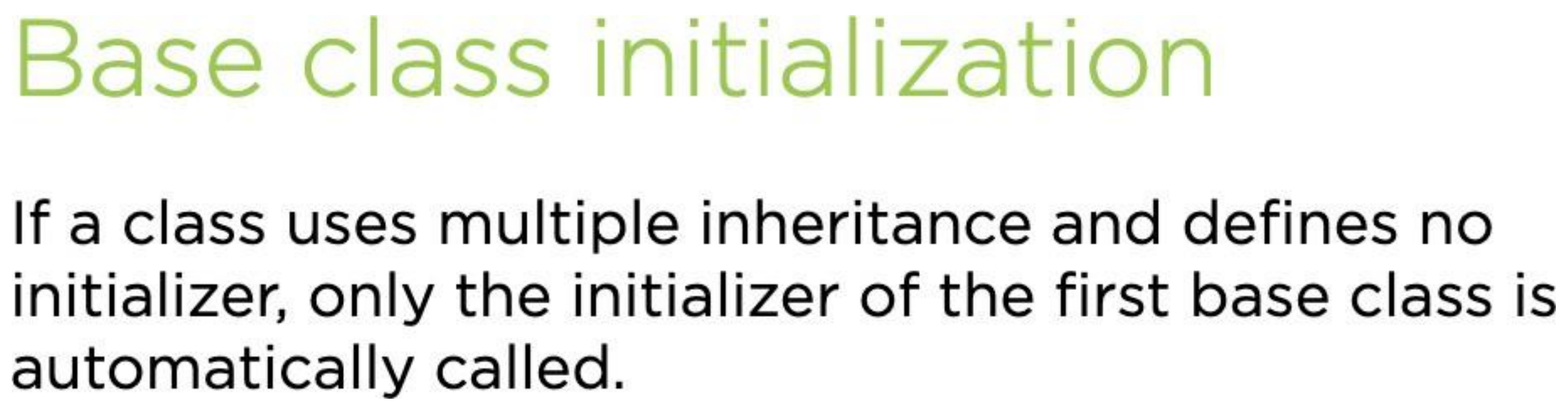
```
class SubClass(Base1, Base2, Base3):
      . . .
```

Name Resolution with Multiple Base Classes

Classes inherit all methods from all of their bases

If there's no method name overlap, names resolve to the obvious method

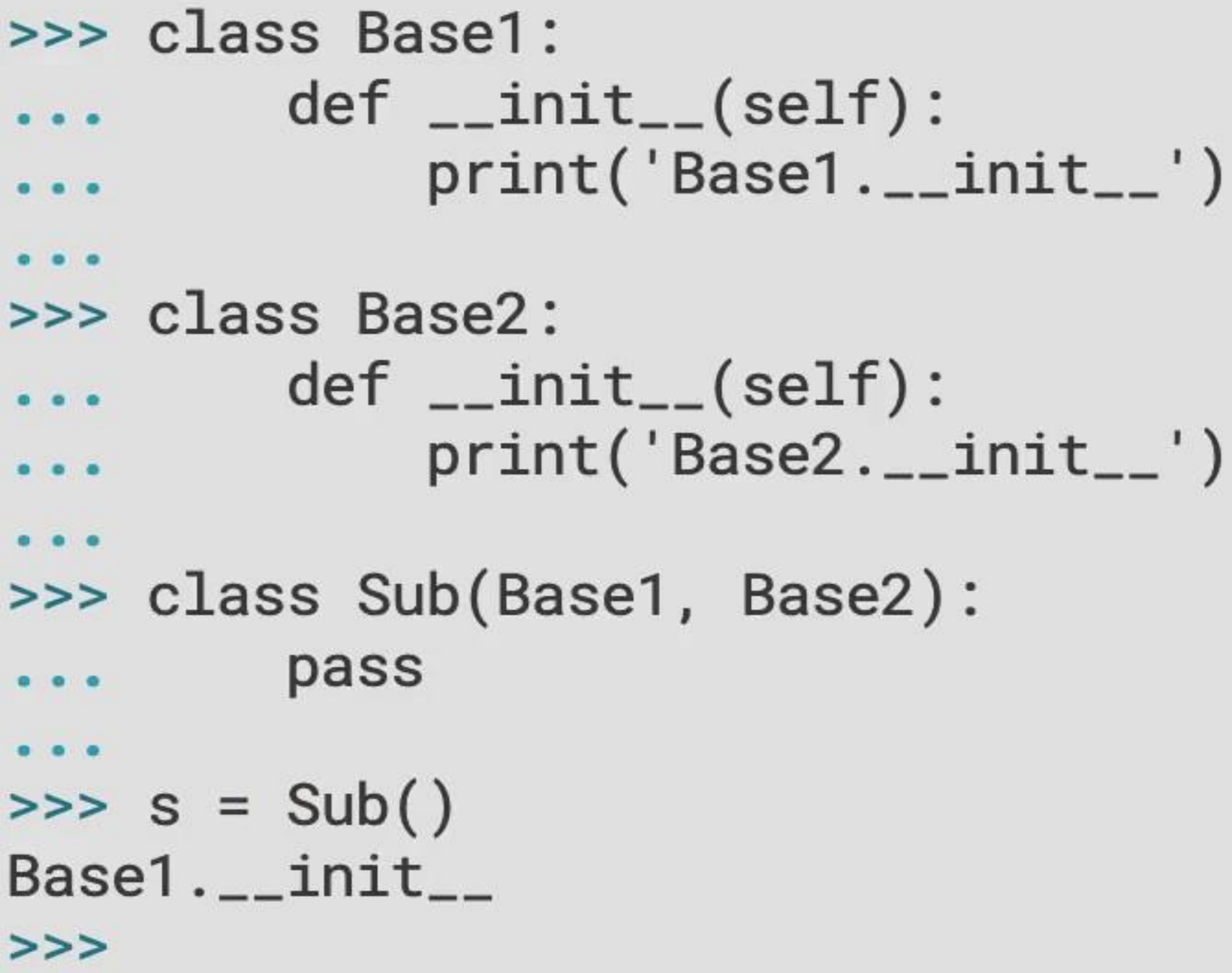
In the case of overlap, Python uses a well-defined "method resolution order" to decide which to use



Base class initialization

If a class uses multiple inheritance and defines no initializer, only the initializer of the first base class is automatically called.

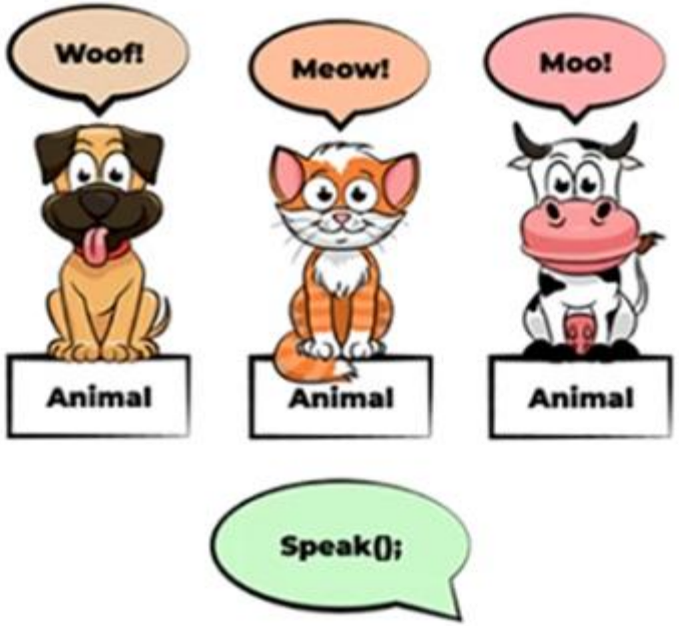
Base Class Initialization



```
>>> class Base1:
...     def __init__(self):
...         print('Base1.__init__')
...
>>> class Base2:
...     def __init__(self):
...         print('Base2.__init__')
...
>>> class Sub(Base1, Base2):
...     pass
...
>>> s = Sub()
Base1.__init__
>>>
```

Polymorphism

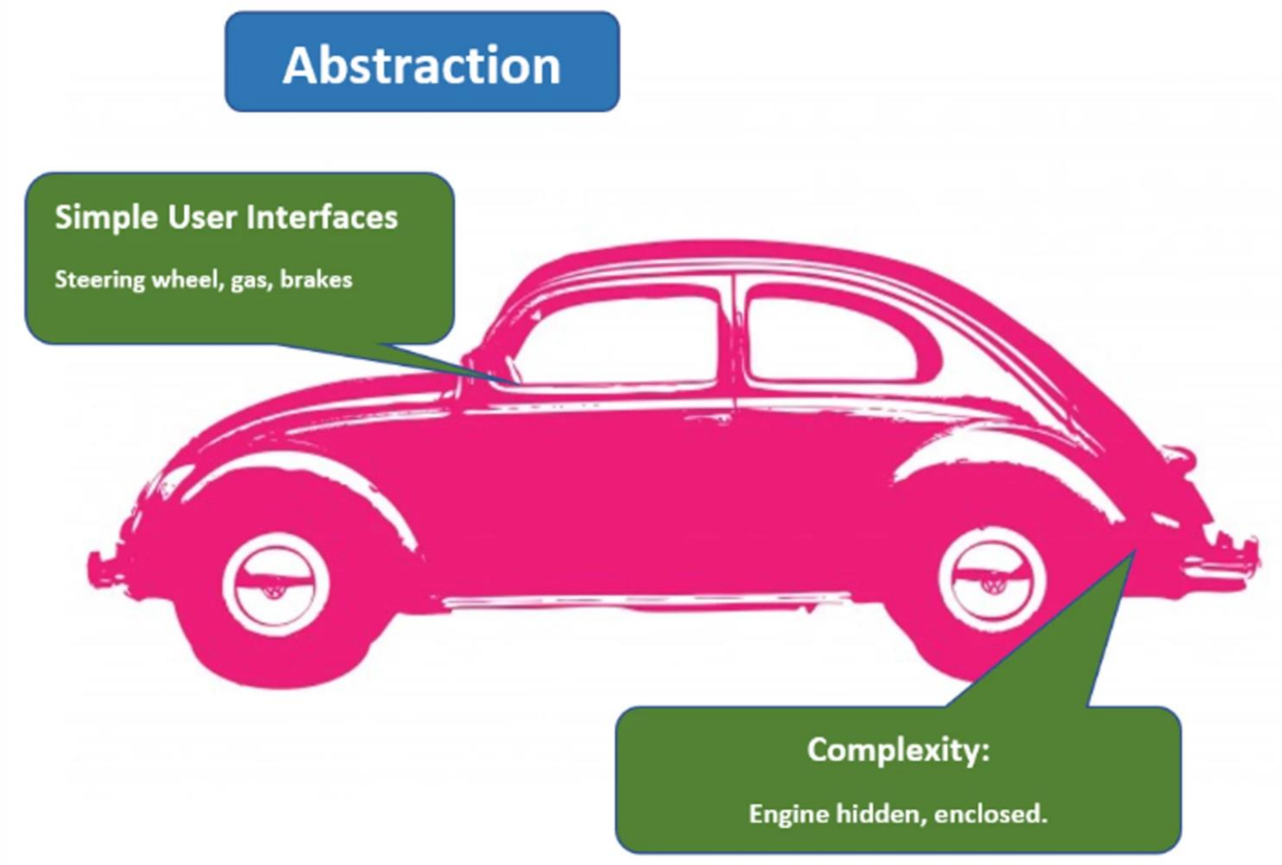
```
class Dog:
    def speak(self):
        print("Woof!")

class Cat:
    def speak(self):
        print("Meow!")

class Cow:
    def speak(self):
        print("Moo!")
```



Abstraction

- Abstraction means that the user interacts with only selected attributes and methods of an object. Abstraction uses simplified, high level tools, to access a complex object.
  - Using simple things to represent complexity
  - Hide complex details from user

- Abstraction is using simple classes to represent complexity.
- Abstraction is an extension of encapsulation.



Abstraction

Simple User Interfaces

Steering wheel, gas, brakes

Complexity:

Engine hidden, enclosed.