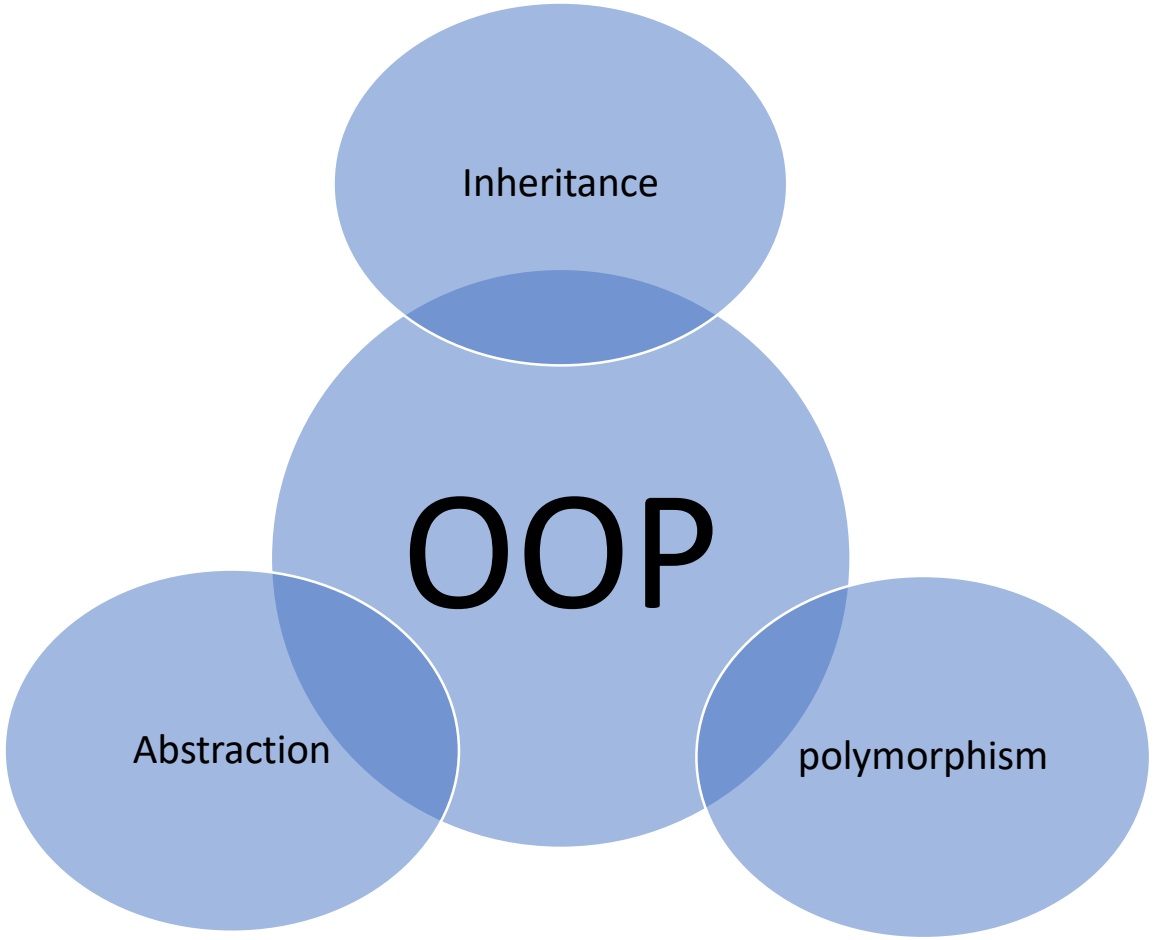
Python for Data Scientists
L6: Classes and inheritance

Concepts of OOP



Classes

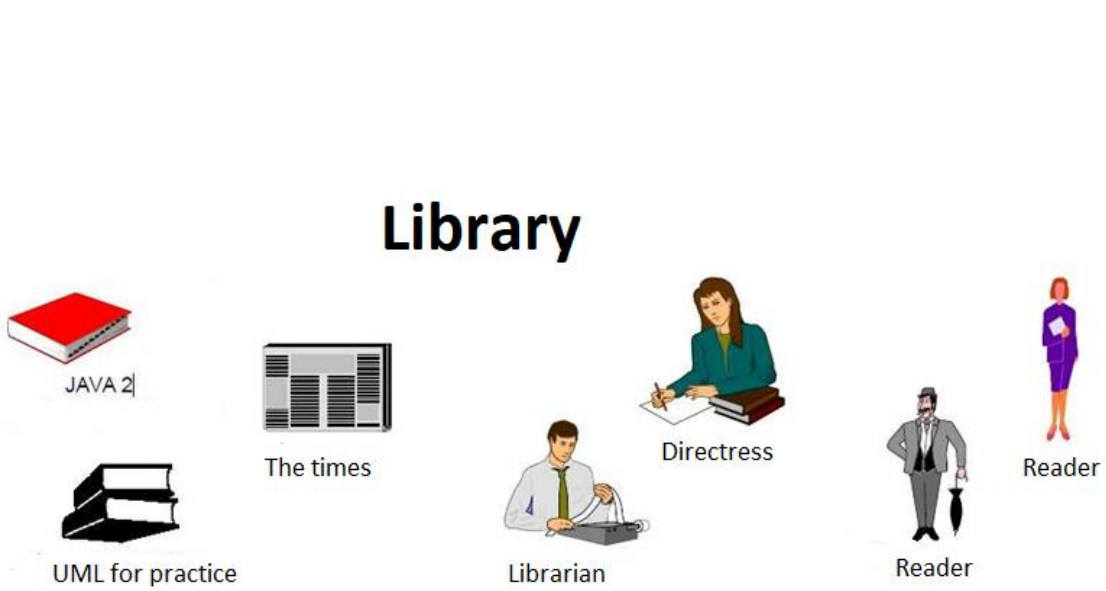
- Implementing a new class:
  - Define the class
  - Define data attributes
  - Define methods

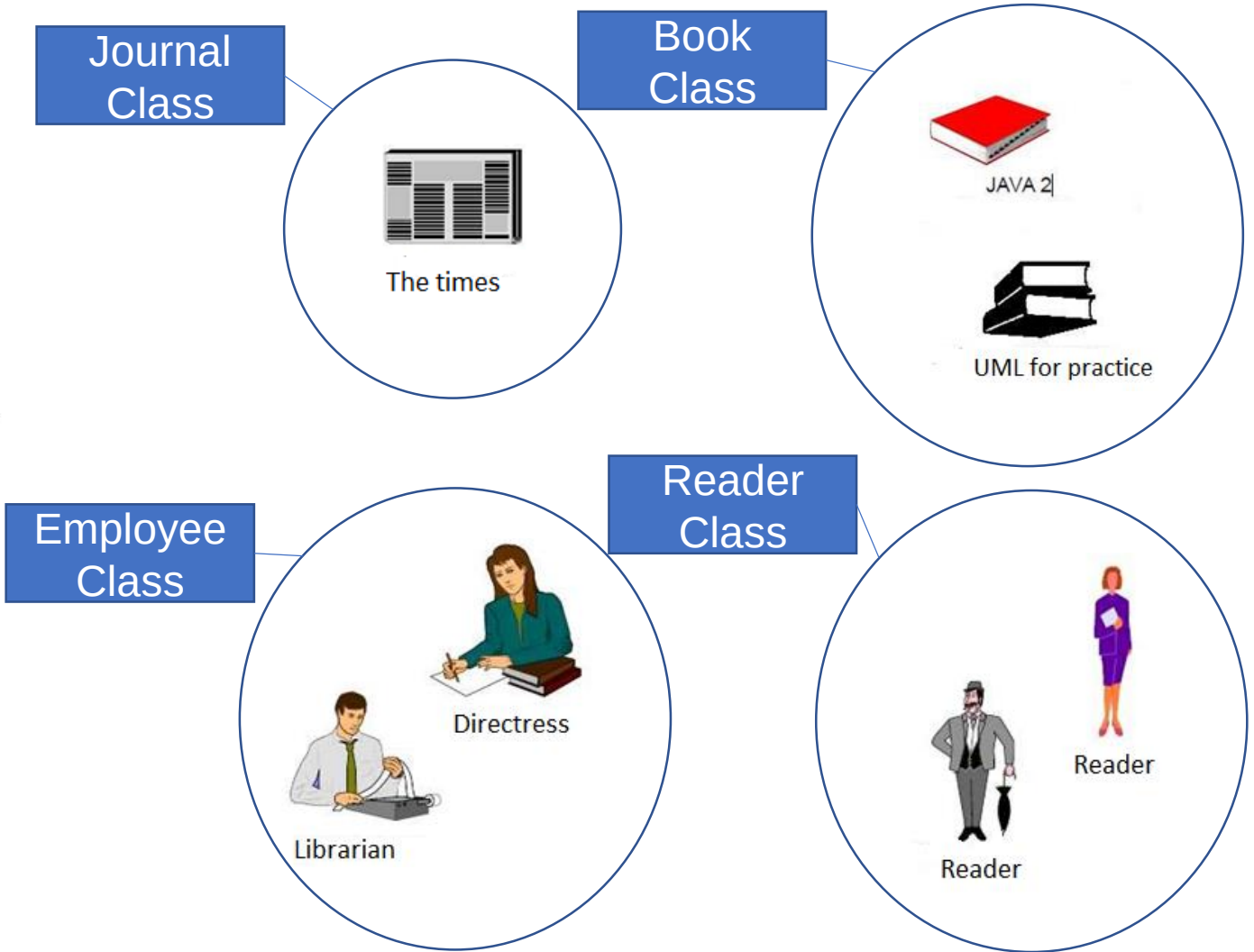
    → class defines data and methods **common across all instances**

- Using the new class
  - Create instance of this class
  - Manipulate these instances

    → instance has the **structure of the class**

Classes

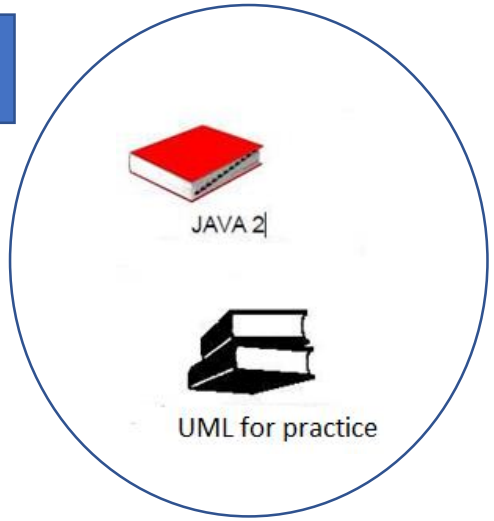

Library
JAVA 2
The times
UML for practice
Librarian
Directress
Reader
Reader


Journal Class
The times
Book Class
JAVA 2
UML for practice
Employee Class
Directress
Librarian
Reader Class
Reader
Reader

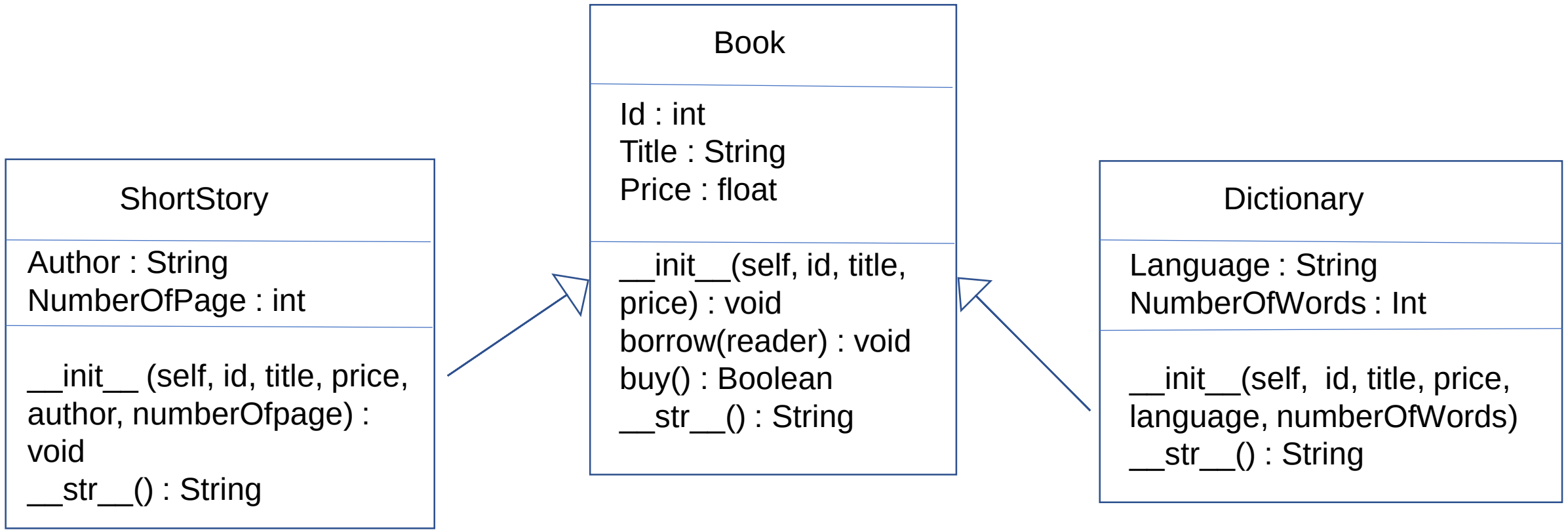
Classes

- Properties :
  - id
  - title
  - price
- Behavior :
  - __init__(self, id, title, price): void
  - borrow(reader): void
  - buy () : Boolean
  - __str__() : String

Book Class



Hierarchies



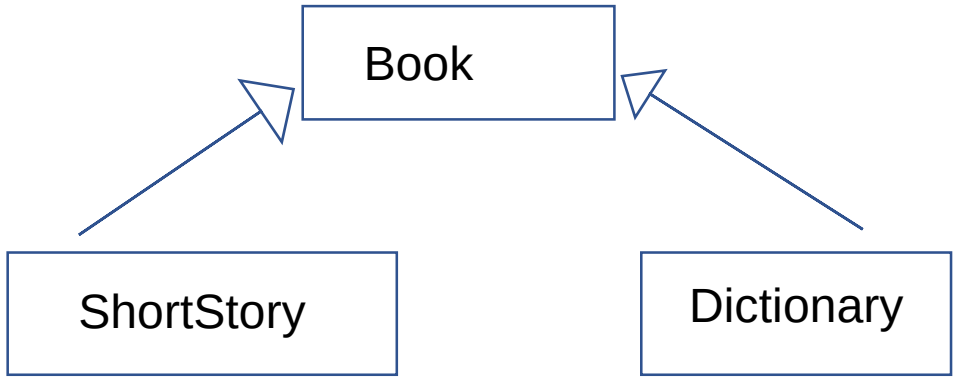
Inheritance

- The ability of a new class to be created, from an existing class by extending it, is called ***inheritance***.

- The new class “inherits” all of the features of its parent, avoiding the creation of new code from scratch.

Inheritance

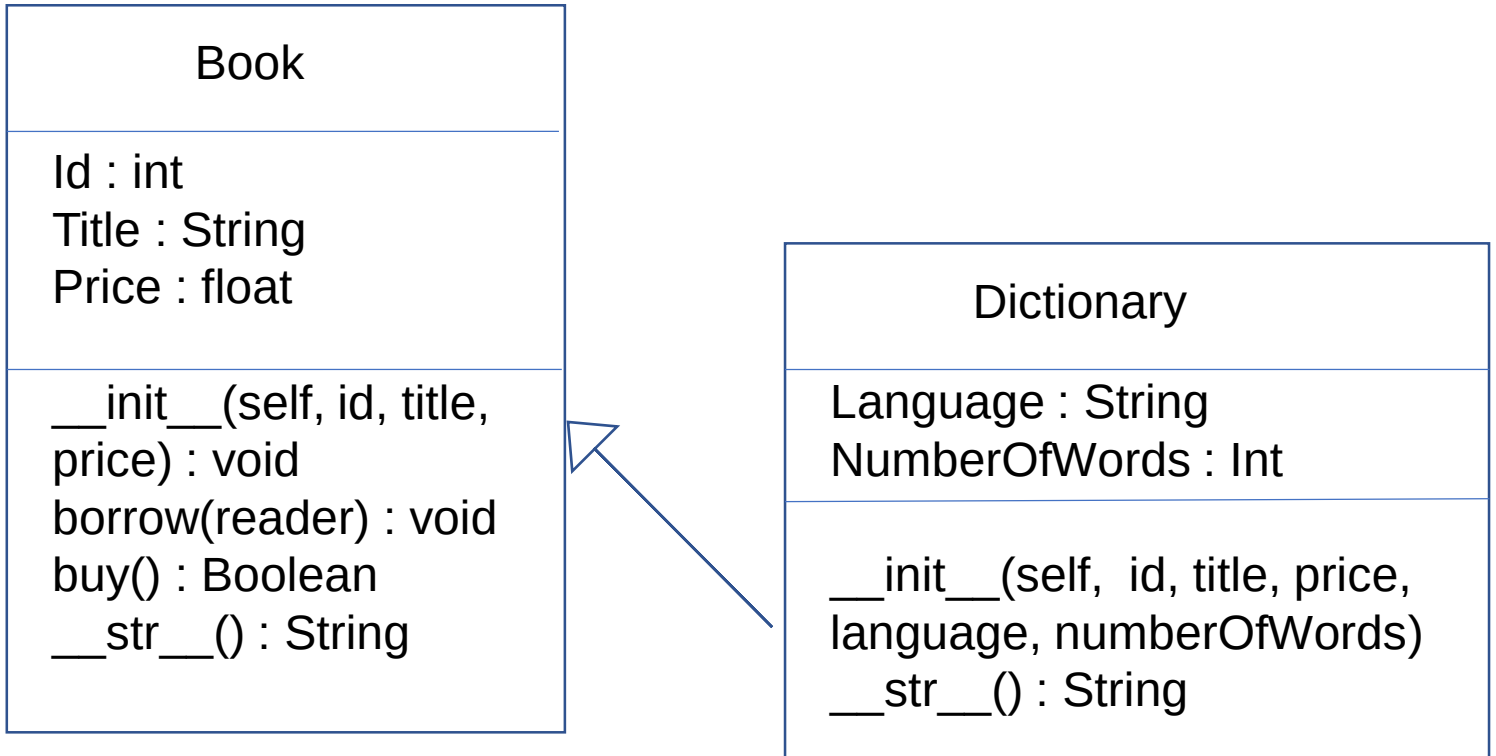
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.



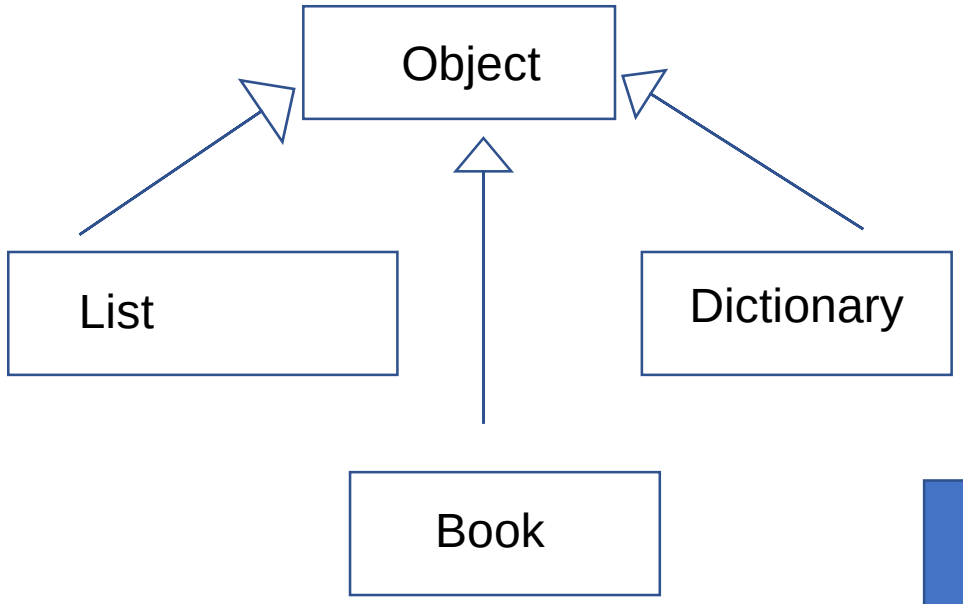
Inheritance

Child class:

- **Inherits** all data and behaviors of parent class
- **Add** more **info**
- **Add** more **behavior**
- **Override** behavior



The Cosmic Superclass: object



The object Class is a Superclass of Every Python Class

Create a Parent class

- Any class can be a parent class
- The syntax is the same as creating any other class

```
class Book:
    def __init__(self, id, title, price):
        self.id = id
        self.title = title
        self.price = price

    def __str__(self):
        return 'id : {self.id}, title : {self.title}, price :{self.price}'.format(self=self)

    def buy(self):
        pass

    def borrow(self, reader):
        pass
```

Create a Parent class

- The class object implements basic operations in Python

```
class Book(object):
    def __init__(self, id, title, price):
        self.id = id
        self.title = title
        self.price = price

    def __str__(self):
        return 'id : {self.id}, title : {self.title}, price :
{self.price}'.format(self=self)

    def buy(self):
        pass

    def borrow(self, reader):
        pass
```

Create a child class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class

```
class ShortStory(Book):
    pass


SS1 = ShortStory(100,"Harry Potter", 500)
print(SS1)
```

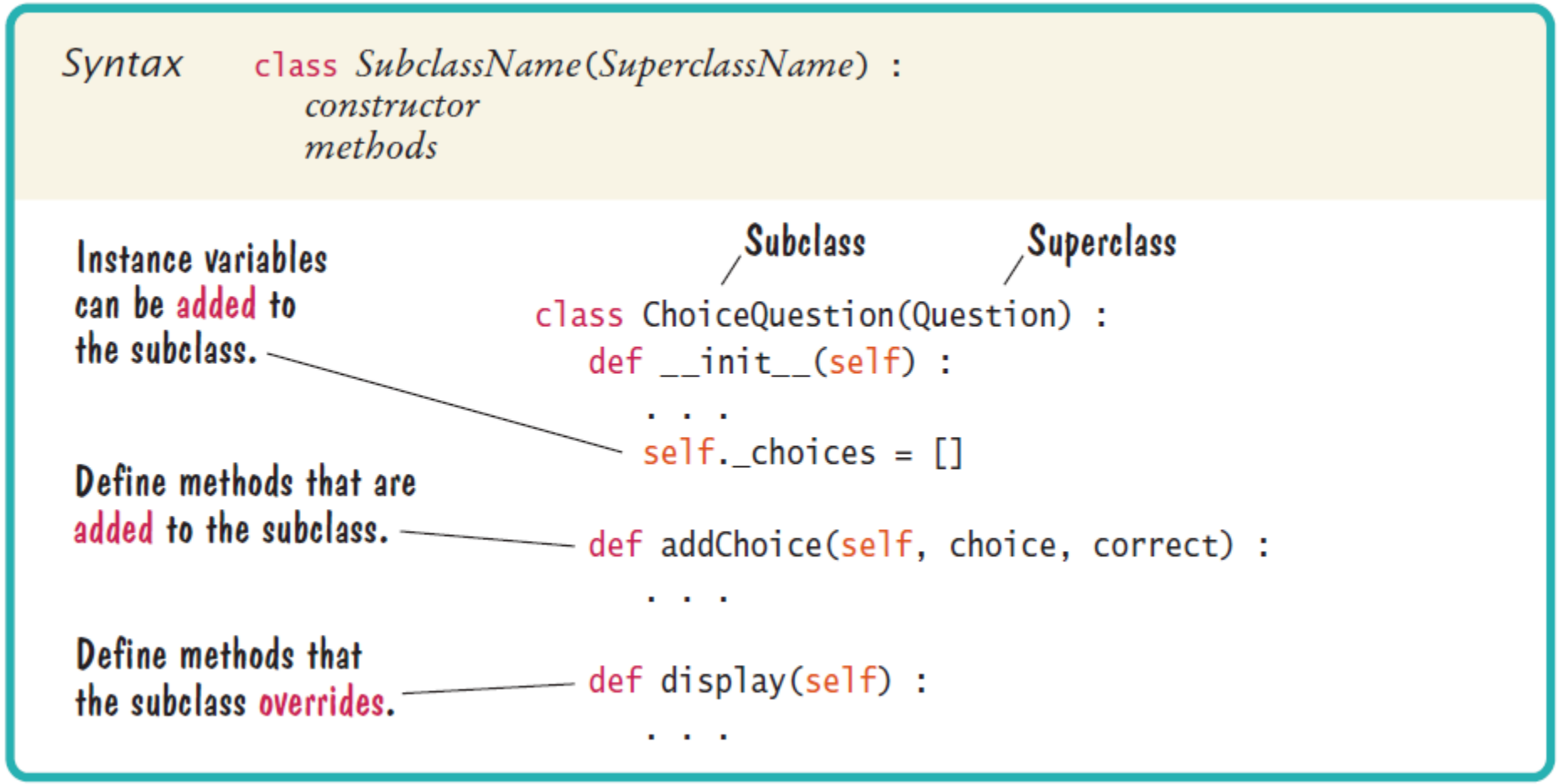
id : 100, title : Harry Potter, price : 500

→ The ShortStory class has the same properties and methods as the Book class.
→ The ShortStory class will use the __init__() of its parent

Subclass Definition

Syntax

```
class SubclassName(SuperclassName) :
   constructor
   methods
```



Python for everyone, Cay S. Horstmann, Rance D. Necaise

Create a child class: __init__()

Add the __init__() function to the child class instead of the pass Keyword

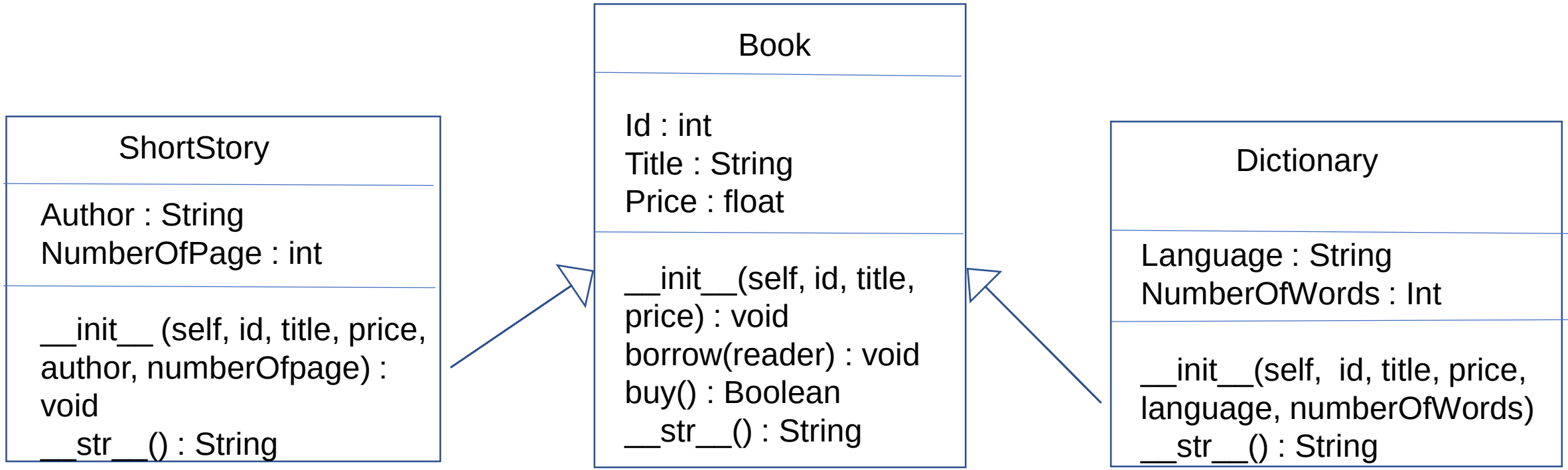
- By adding the __init__() function, the child class will no longer inherit the parent's __init__() function

→ The child __init__() function **overrides** the inheritance of the parent's __init__() function

Create a child class: __init__()

```
class ShortStory(Book):
    def __init__(self, id, title, price, author, nbPages):
        self.id = id
        self.title = title
        self.price = price
        self.author = author
        self.numberOfPages = nbPages
```

To keep the inheritance of the parent's __init__() function, we need to add a call to the parent's __init__() function



Create a child class: __init__()

```
class ShortStory(Book):
    def __init__(self, id, title, price, author, nbPages):
        super().__init__(id, title, price)
        self.author = author
        self.numberOfPages = nbPages
```

Create a child class: __init__()

Syntax

```
class SubclassName(SuperclassName) :
    def __init__(self, parameterName1, parameterName2, . . .) :
        super().__init__(arguments)
        constructor body
```

- super() : make the child class inherit all the methods and properties from its parent.
- it will automatically inherit the methods and properties from its parent without specifying the name of the parent.

Poll

What will be the output of this code?

A- 0 0

B- null 0

C- AttributeError

```
class A:
    def __init__(self, i = 0):
        self.i = i

class B(A):
    def __init__(self, j = 0):
        self.j = j


b = B()
print(b.i)
print(b.j)
```

AttributeError: 'B' object has no attribute 'i’

Which method to use?

- look for a method name in **current class definition**
- if not found, look for method name **up the hierarchy** (in parent, then grandparent, and so on)
- use first method up the hierarchy that you found with that method name

Which method to use?

```
class A:
    def __init__(self, id):
        self.id = id
    def str(self):
        print('Object', self.id)

class B(A):
    def __init__(self, id):
        super().__init__(id)


a = A(1)
b = B(2)
a.str()
b.str()
```

Object 1
Object 2

Poll

What will be the output of this code?

A- i from A is 60

B- i from A is 90

C- None of the above

```
class A:
    def __init__(self):
        self.calcI(30)
        print("i from A is", self.i)

    def calcI(self, i):
        self.i = 2 * i;


class B(A):
    def __init__(self):
        super().__init__()

    def calcI(self, i):
        self.i = 3 * i;


b = B()
```

Poll

What will be the output of this code?

A- Demo’s check Demo’s check

B- Demo’s check AttributeError

C- None of the above

```
class A:
    def __check(self):
        return " Demo's check "
    def display(self):
        print(self.__check())

class B(A):
    def display(self):
        print(self.__check())

d1 = A()
d1.display()
d2 = B()
d2.display()
```

Private and protected methods

```
class A:
    def _check(self):
        return " Demo's check "
    def display(self):
        print(self._check())

class B(A):
    def display(self):
        print(self._check())

d1 = A()
d1.display()
d2 = B()
d2.display()
```

```
Demo's check
Demo's check
```

Overriding Methods

An overriding method can extend or replace the functionality of the superclass method.

Overriding Methods

```
class A:
    def __init__(self, id):
        self.id = id
    def str(self):
        print('Object A', self.id)

class B(A):
    def __init__(self, id):
        super().__init__(id)
    def str(self):
        print('Object B', self.id)

a = A(1)
b = B(2)
a.str()
b.str()
```

Object A 1
Object B 2

Polymorphism

Polymorphism: same function name can be used for different types

→ This makes programming more intuitive and easier.

In Python, there are different types of polymorphism:

- Polymorphism with function and objects
- Polymorphism with class methods
- Polymorphism with inheritance

Polymorphism with Function and Objects

Function can take any object

```
Vegetable
Green
Fruit
Red
```

```
class Strawberry():
    def type(self):
        print("Fruit")

    def color(self):
        print("Red")

class Spinach():
    def type(self):
        print("Vegetable")

    def color(self):
        print("Green")


def function(obj):
    obj.type()
    obj.color()

obj_spinach = Spinach()
obj_strawberry = Strawberry()
function(obj_spinach)
function(obj_strawberry)
```

Polymorphism with Class Methods

Python uses two different class types in the same way :

- create a for loop that iterates through a **tuple** of objects.
- call the methods without being concerned about which class type each object is.

```
Vegetable
Green
Fruit
Red
```

```
class Strawberry():
    def type(self):
        print("Fruit")

    def color(self):
        print("Red")

class Spinach():
    def type(self):
        print("Vegetable")

    def color(self):
        print("Green")


def function(obj):
    obj.type()
    obj.color()

obj_spinach = Spinach()
obj_strawberry = Strawberry()
for obj in (obj_spinach, obj_strawberry):
    obj.type()
    obj.color()
```

Polymorphism with Inheritance

- In OOP these multiple forms refer to multiple forms of the same method

- The method inherited from the parent class doesn’t fit into the child class → Solution: re-implement method in the child class : Method overriding

Polymorphism with Inheritance

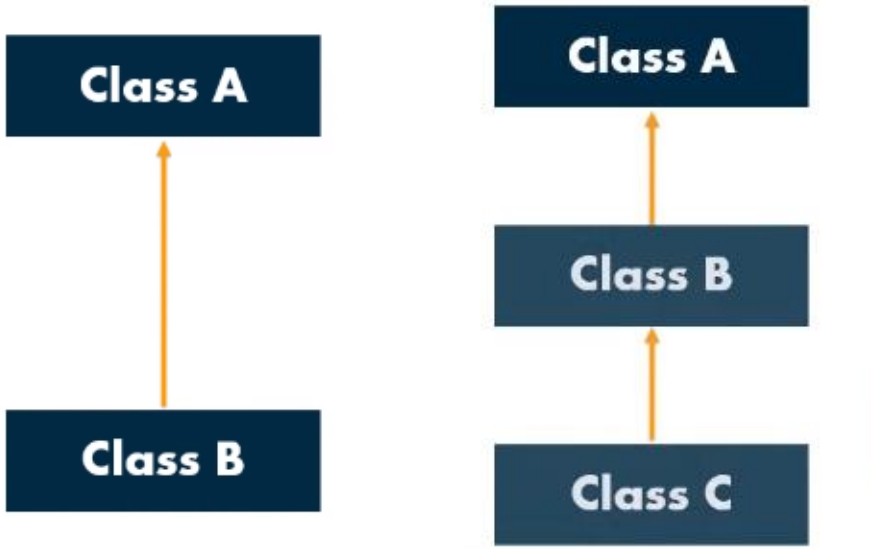
```
class Bird:
    def description(self):
        print("There are different types of birds")

    def flight(self):
        print("Most of the birds can fly but some cannot")


class pigeon(Bird):
    def flight(self):
        print("Pigeons can fly")


class penguin(Bird):
    def flight(self):
        print("Penguins do not fly")
```

```
obj_bird = Bird()
obj_pig = pigeon()
obj_peng = penguin()

obj_bird.description()
obj_bird.flight()

obj_pig.description()
obj_pig.flight()

obj_peng.description()
obj_peng.flight()
```

```
There are different types of birds
Most of the birds can fly but some cannot
There are different types of birds
Pigeons can fly
There are different types of birds
Penguins do not fly
```

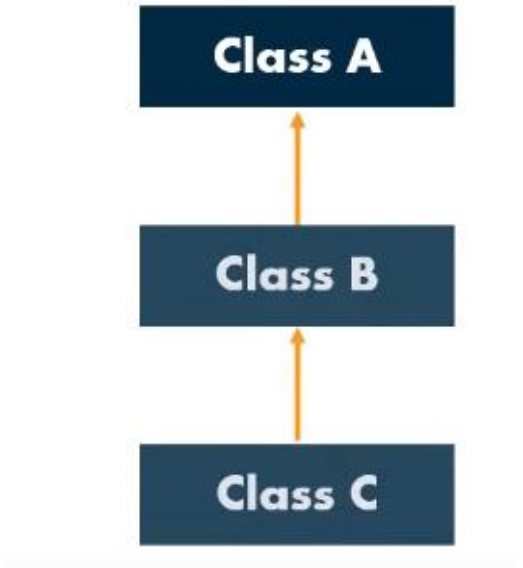
Inheritance: Types

- Single
- Multilevel
- Multiple



Multilevel inheritance

- Methods and proprieties of the base class and the derived class are inherited into the new derived class.



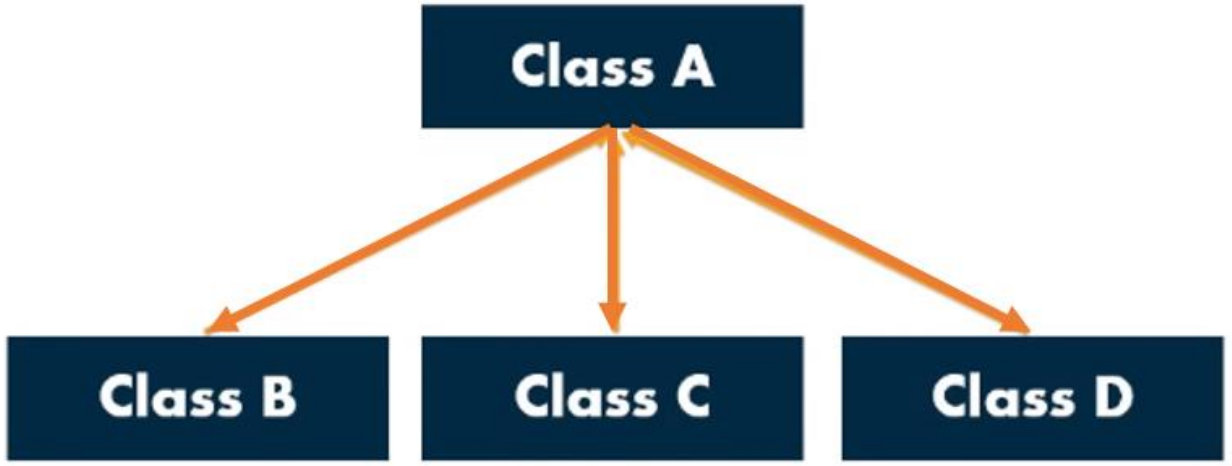
Multilevel inheritance

```
class First(object):
    def __init__(self):
        super().__init__()
        print("First")


class Second(First):
    def __init__(self):
        super().__init__()
        print("Second")


class Third(Second):
    def __init__(self):
        super().__init__()
        print("Third")


Third()
```

```
First
Second
Third
```

Multiple inheritance

Methods and proprieties of all the base classes are inherited into the derived class.



Multiple inheritance

```
class First(object):
    def __init__(self):
        super().__init__()
        print("First")


class Second(object):
    def __init__(self):
        super().__init__()
        print("Second")


class Third(Second, First):
    def __init__(self):
        super().__init__()
        print("Third")

Third()
```

```
First
Second
Third
```

Multiple inheritance (Poll)

```
class B:
    def x(self):
        print('x: B')


class C:
    def x(self):
        print('x: C')


class D(B, C):
    pass


d = D()
d.x()
print(D.mro())
```

What will be the output of this code?

A- x: B

B- x: C

C- None of the above

```
x: B
[<class '__main__.D'>, <class '__main__.B'>, <class
'__main__.C'>, <class 'object'>]
```

Abstract classes

- Abstract classes : contain one or more abstract methods.

- Abstract method : method that is declared but contains no implementation.

→Abstract classes :

→cannot be instantiated

→require subclasses to provide implementations for the abstract methods.

Abstract classes

- Python proposed a module which provides the infrastructure for defining Abstract Base Classes

- This module is called : abc

Abstract classes

```
from abc import ABC, abstractmethod


class AbstractClassExample(ABC):

    def __init__(self, value):
        self.value = value
        super().__init__()

    @abstractmethod
    def function(self):
        pass

class ChildClass(AbstractClassExample):
    pass

c1 = ChildClass(4)
```

c1 = ChildClass(4)
TypeError: Can't instantiate abstract class
ChildClass with abstract methods function

Abstract classes

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def __init__(self, value):
        self.value = value
        super().__init__()

    @abstractmethod
    def function(self):
        pass

class ChildClass(AbstractClassExample):
    def function(self):
        return self.value * 2

c1 = ChildClass(4)
print(c1.function())
```

A class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

Abstract classes

- Be aware that an abstract method can have an implementation in the abstract class! But still, in the subclasses, the implementation must be override.

- The abstract method can be called with super(). This allows some basic functionality in the abstract method, which can be enriched by the subclass implementation.

Abstract classes

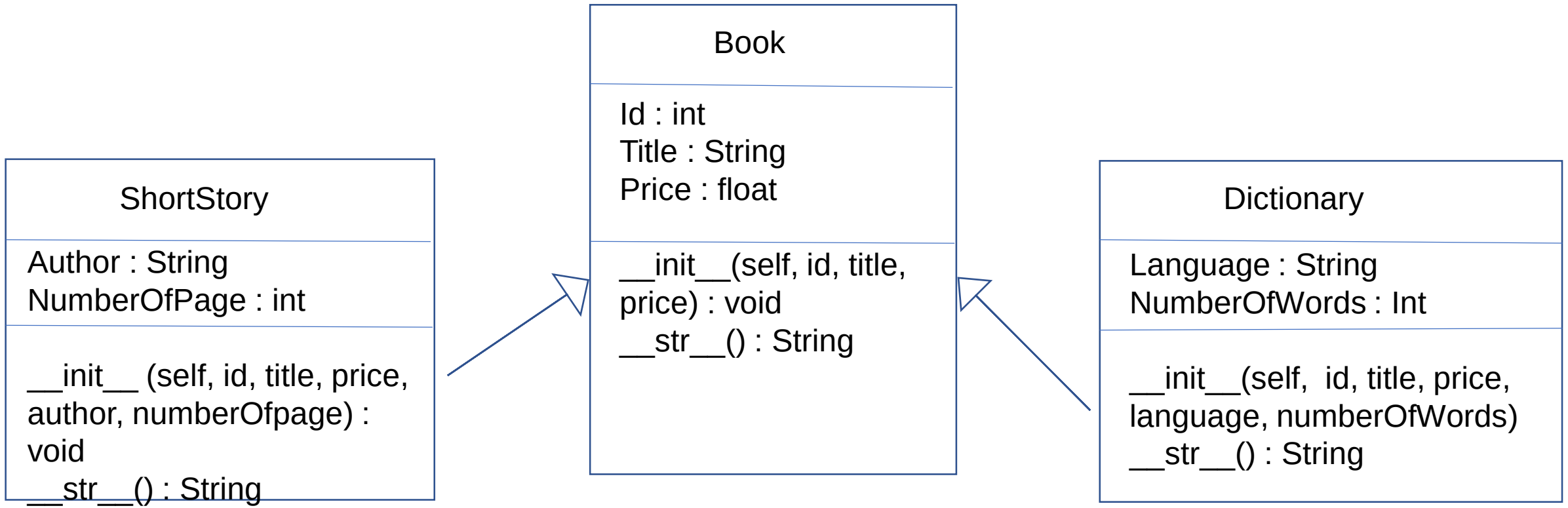
```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def __init__(self, value):
        self.value = value
        super().__init__()

    @abstractmethod
    def function(self):
        print("Some implementation!")

class ChildClass(AbstractClassExample):
    def function(self):
        super().function()

c1 = ChildClass(4)
print(c1.function())
```

Some implementation!
None

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def __init__(self, value):
        self.value = value
        super().__init__()

    @abstractmethod
    def function(self):
        print("Some implementation!")

class ChildClass(AbstractClassExample):
    def function(self):
        super().function()
        print("The enrichment from
AnotherSubclass")

c1 = ChildClass(4)
print(c1.function())
```

Some implementation!
The enrichment from AnotherSubclass
None

Recap


Book
Id : int
Title : String
Price : float
__init__(self, id, title, price) : void
__str__() : String
ShortStory
Author : String
NumberOfPage : int
__init__ (self, id, title, price, author, numberOfpage) : void
__str__() : String
Dictionary
Language : String
NumberOfWords : Int
__init__(self, id, title, price, language, numberOfWords)
__str__() : String

Recap

__init__() function

```
class Book:
    def __init__(self, id, title, price):
        self.id = id
        self.title = title
        self.price = price

    def __str__(self):
        return 'id : {self.id}, title : {self.title}, price : {self.price}'.format(self=self)


class ShortStory(Book):
    def __init__(self, id, title, price, author, nbPages):
        super().__init__(id, title, price)
        self.author = author
        self.nbPages = nbPages

    def __str__(self):
        s = super().__str__()
        return s + ' author : {self.author}, Number of pages : {self.nbPages}'.format(self=self)
```

Inheritance

```
class Dictionary(Book):
    def __init__(self, id, title, price, language, nbWords):
        super().__init__(id, title, price)
        self.language = language
        self.nbWords = nbWords

    def __str__(self):
        s= super().__str__()
        return s+'language : {self.language}, Number of words : {self.nbWords}'.format(self=self)


SS1 = ShortStory(101,"The gift of the magi", 100, "O. Henry", 36)
D1 = Dictionary(102, "Larousse", 300, "Espagnol", 32000)
for obj in (SS1, D1):
    print(obj.__str__())
```

Polymorphism with Inheritance

Polymorphism with Class Methods

OOP Recap

- Create your own **collections of data**
- **Organize** information
- **Division** of work
- Access information in a **consistent** manner
- Add **layers** of complexity
- Like functions, classes are a mechanism for **decomposition** and **abstraction** in programming

Next modules

Data structure and algorithms:

- Algorithms: searching and sorting algorithms
- Data structure in computer science