SWCON104
Web & Python Programming

Class

Department of Software Convergence

경희대학교
KYUNG HEE UNIVERSITY

Today

- Object Oriented Programming (OOP)
  - Object (= Instance)
  - Class (= Type)
  - Method (= Member Function)
  - Inheritance

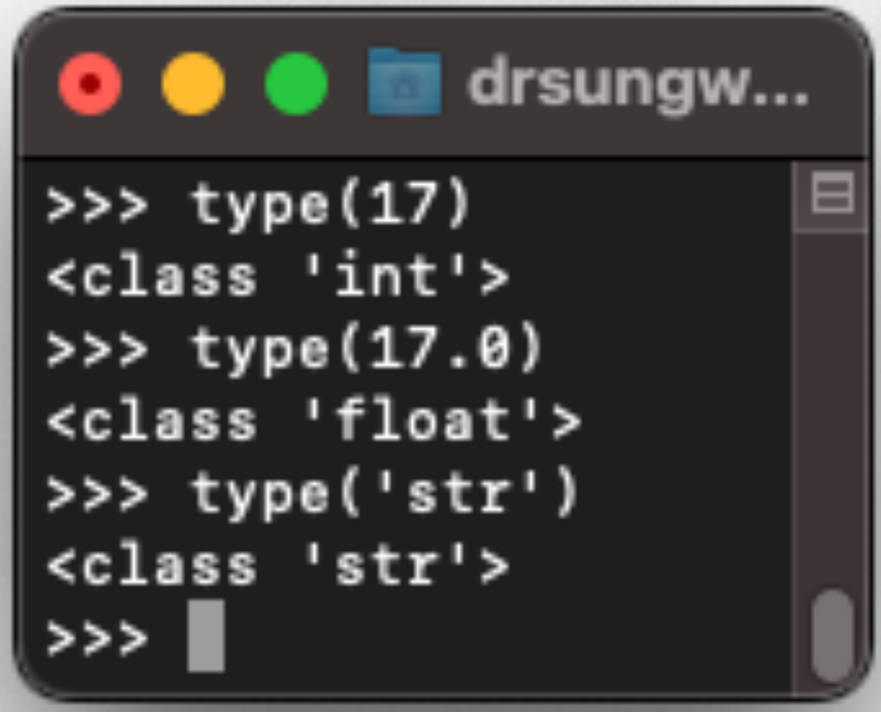
[Textbook]
Practical Programming
(An Introduction to Computer Science Using Python),
by Paul Gries, Jennifer Campbell, Jason Montojo,
The Pragmatic Bookshelf, 2017

Practice

- Practice_15_Class_SourceCode_Lab.zip

Types

- Standard types
  - Integer
  - Float
  - String
  - Boolean
  - List
  - Set
  - Tuple
  - Dictionary



- Type is a class
- How to make your own type

Object and variable

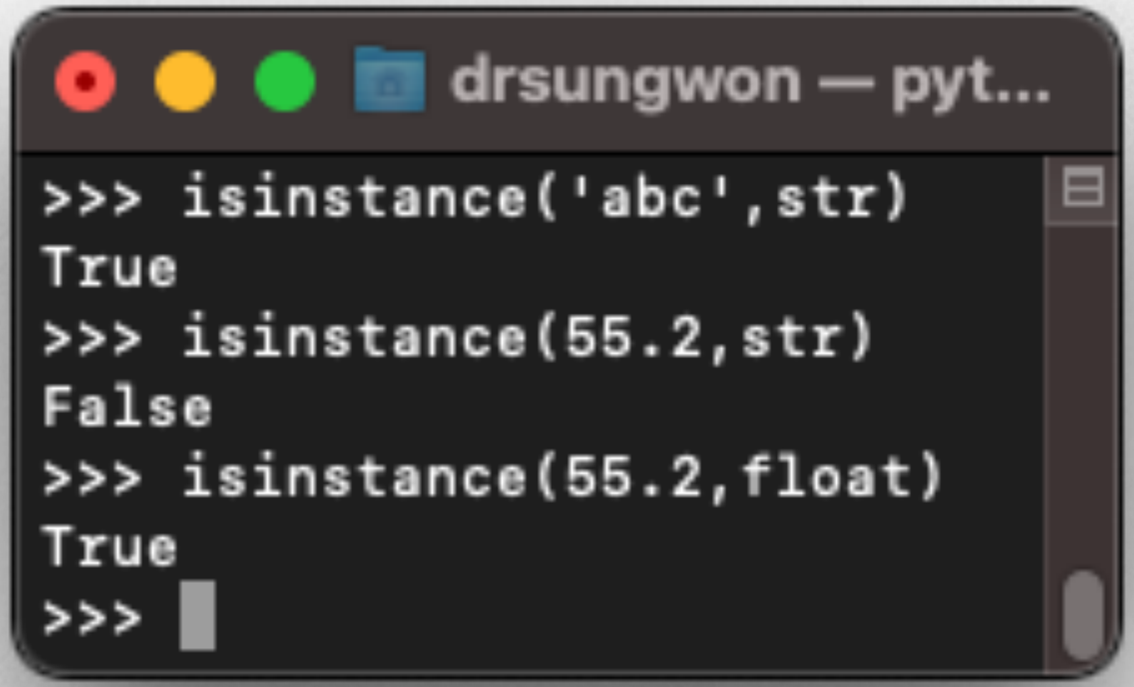
- Every location in the computer's memory has a memory address
- Object: a value (or thing) at a memory address with a type

26.0 **id1** **float**

- Variable contains the memory address of the object **degrees_celsius**

Function isinstance

- Function isinstance reports whether an object is an instance of a class – whether an object has a particular type:



- 'abc' is an instance of str, but 55.2 is not.
- 55.2 is an instance of float.

What is Class?

- https://en.wikipedia.org/wiki/Class_(computer_programming)



Class Student

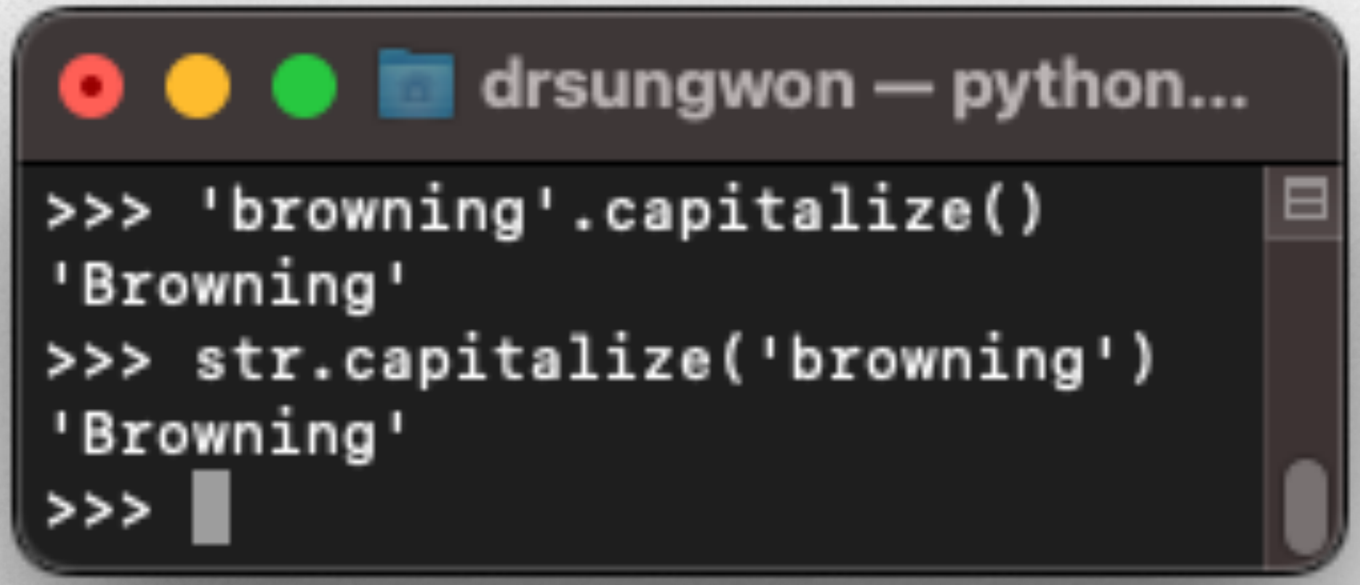
- Student is a class.
- a, b, and c are the instances of the class Student.
- name, id, gender~~, birthyear~~ are instance variables (or member variables).
- a = Student() creates a Student object and then assigns that object to variable a.

```
class Student:
    name = ""
    id = 0
    gender = "female"

a = Student()
b = Student()
c = Student()

a.name = "Harry Potter"
a.id = 2017103709
a.gender = "male"

b.name = "Hermione Granger"
b.id = 2018103722

c.name = "Ron Weasley"

print(a.name)
print(a.id)
print(a.gender)

print(b.name)
print(b.id)
print(b.gender)

print(c.name)
print(c.id)
print(c.gender)
```

```
Harry Potter
2017103709
male
Hermione Granger
2018103722
female
Ron Weasley
0
female
```

Methods

- Functions inside Class



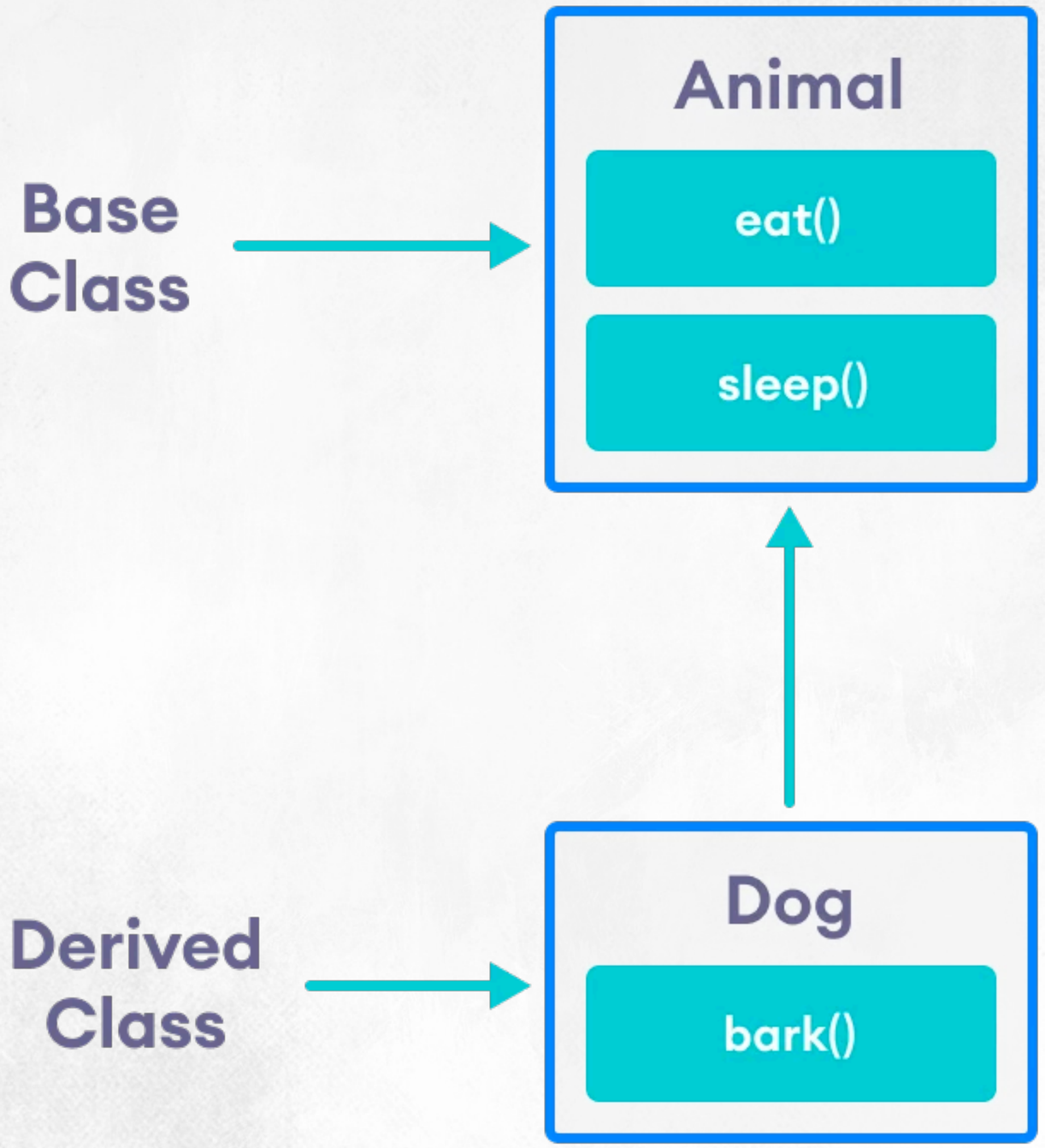
Methods in Class Student

```python
class Student:
    name = ""
    id = 0
    gender = "female"
    course = []

    def num_courses(self):
        return len(self.course)

a = Student()

a.name = "Harry Potter"
a.id = 2017103709
a.gender = "male"
a.course = ["English", "Programming"]

print(a.num_courses())
print(Student.num_courses(a))
```

self
self.___

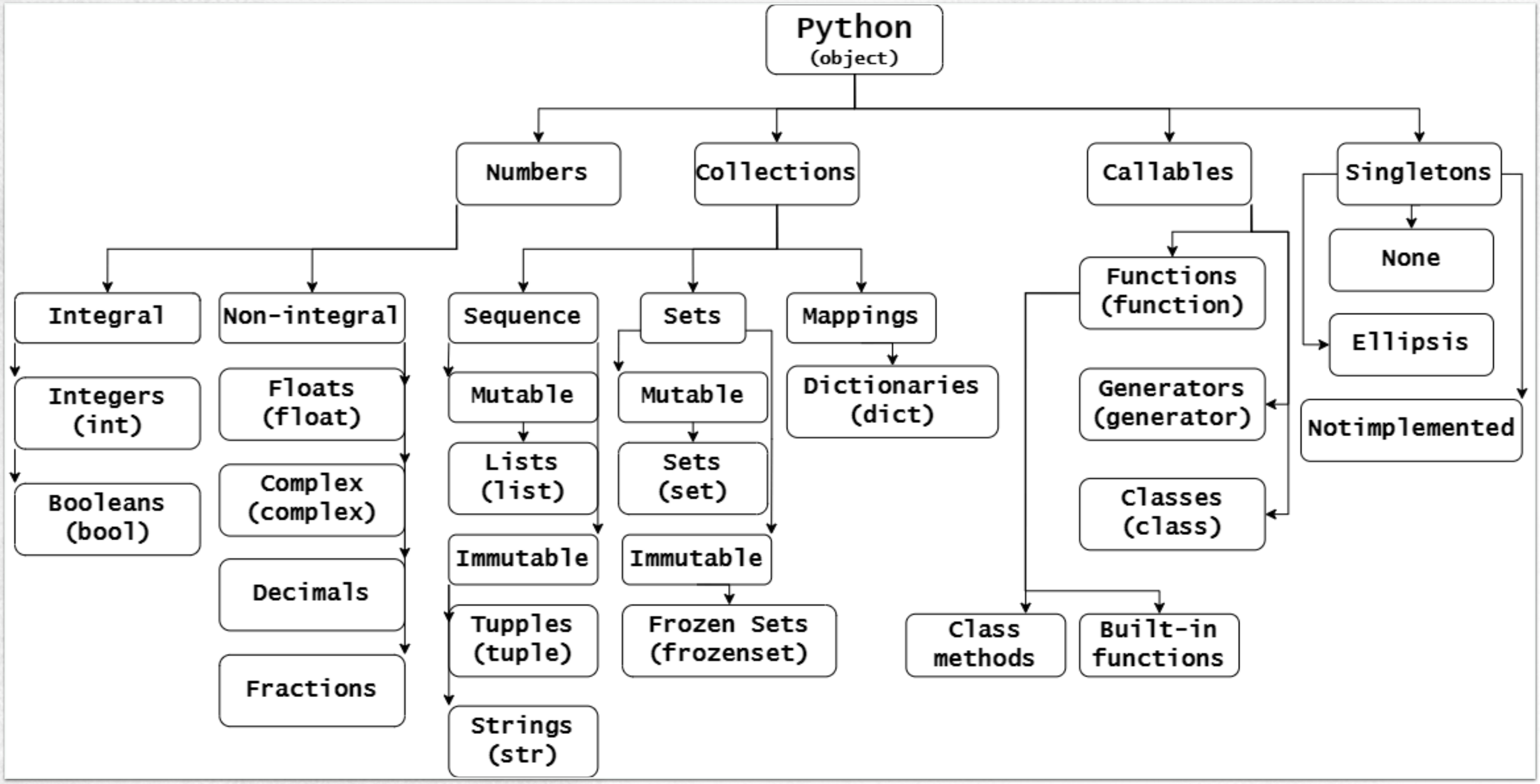
2
2

Inheritance



Reference: https://www.programiz.com/cpp-programming/inheritance

Inheritance



Reference: https://python.plainenglish.io/python-type-hierarchy-378a049f2f81

Inheritance

```
>>> isinstance('abc',str)
True
>>> isinstance(55.2,str)
False
>>> isinstance(55.2,float)
True
```

```
>>> help(object)
Help on class object in module builtins:

class object
 |  The most base type
```

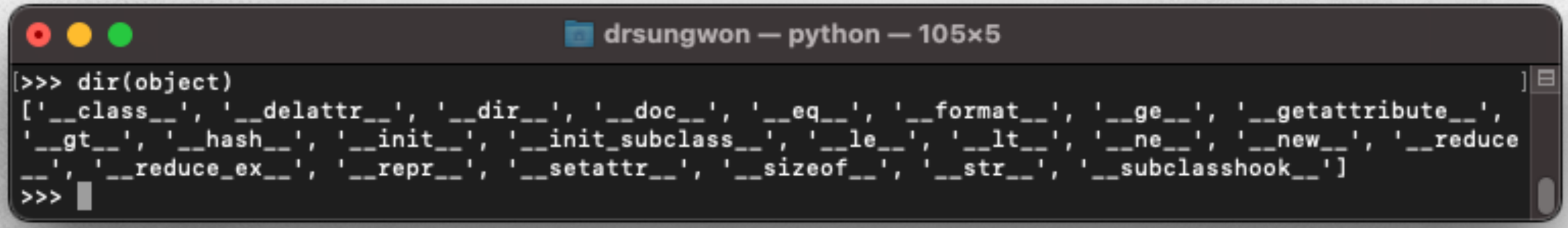
```
>>> isinstance(55.2,object)
True
>>> isinstance('abc',object)
True
```

```
>>> isinstance(str, object)
True
>>> isinstance(max, object)
True
```

- 'abc' is an instance of class str
- 55.2 is an instance of class float
- 'abc' and 55.2 are instances of class object
- Classes and functions are instances of class object
- Every class in Python is derived from class object, so every instance of every class is an derived classes of class object.
- Class object is the superclass of class str, and class str is a subclass of class object.

Inheritance

- `dir` shows a list of attributes (attributes are variables inside a class that refer to methods, functions, variables, or other classes)



- Every class in Python inherits these attributes from class object: they are automatically part of every class.
- Every subclass inherits the features of its superclass.
- It helps avoid a lot of duplicate code and makes interactions between related types consistent.

Inheritance

- Every class in Python inherits these attributes from class object: they are automatically part of every class.
- Every subclass inherits the features of its superclass.

drsungwon — python — 105×17

```
>>> dir(object)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce
__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> class Book:
...     """No code"""
...
>>> type(Book)
<class 'type'>
>>> type(str)
<class 'type'>
>>> dir(Book)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getat
tribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__n
e__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__su
bclasshook__', '__weakref__']
>>>
```

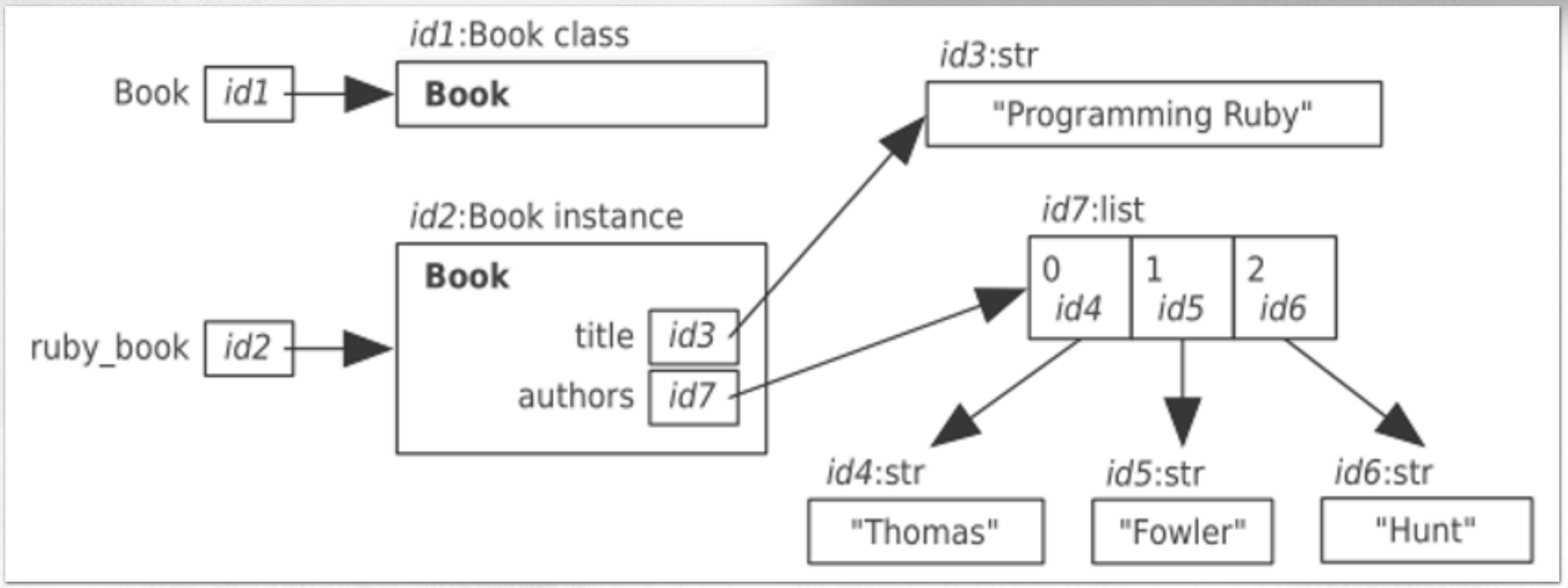
Book class

drsungwon — python — 53×11

```
>>> class Book:
...     """No Code"""
...
>>> ruby_book = Book()
>>> ruby_book.title = "Programming Ruby"
>>> ruby_book.authors = ["Thomas", "Fowler", "Hunt"]
>>> ruby_book.title
'Programming Ruby'
>>> ruby_book.authors
['Thomas', 'Fowler', 'Hunt']
>>>
```

drsungwon — less ◂ python — 59×15

```
Help on class Book in module __main__:

class Book(builtins.object)
 |  No Code
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
(END)
```



Book class

book.py

```python
class Book:
    """Description about a book"""

    title = ""
    authors = ""

    def num_authors(self):
        """(Book) -> int
        Return the number of authors of this book.
        """
        return len(self.authors)
```

```
다운로드 — python — 53×10
>>> import book
>>> ruby_book = book.Book()
>>> ruby_book.title = "Programming Ruby"
>>> ruby_book.authors = ["Thomas", "Fowler", "Hunt"]
>>>
>>> ruby_book.num_authors()
3
>>> book.Book.num_authors(ruby_book)
3
>>>
```

- book is the imported module
- In that module, there is class Book
- Inside Book, there is method num_authors.
- The argument to the call, ruby_book, is passed to parameter self.

__init__ method (Constructor)

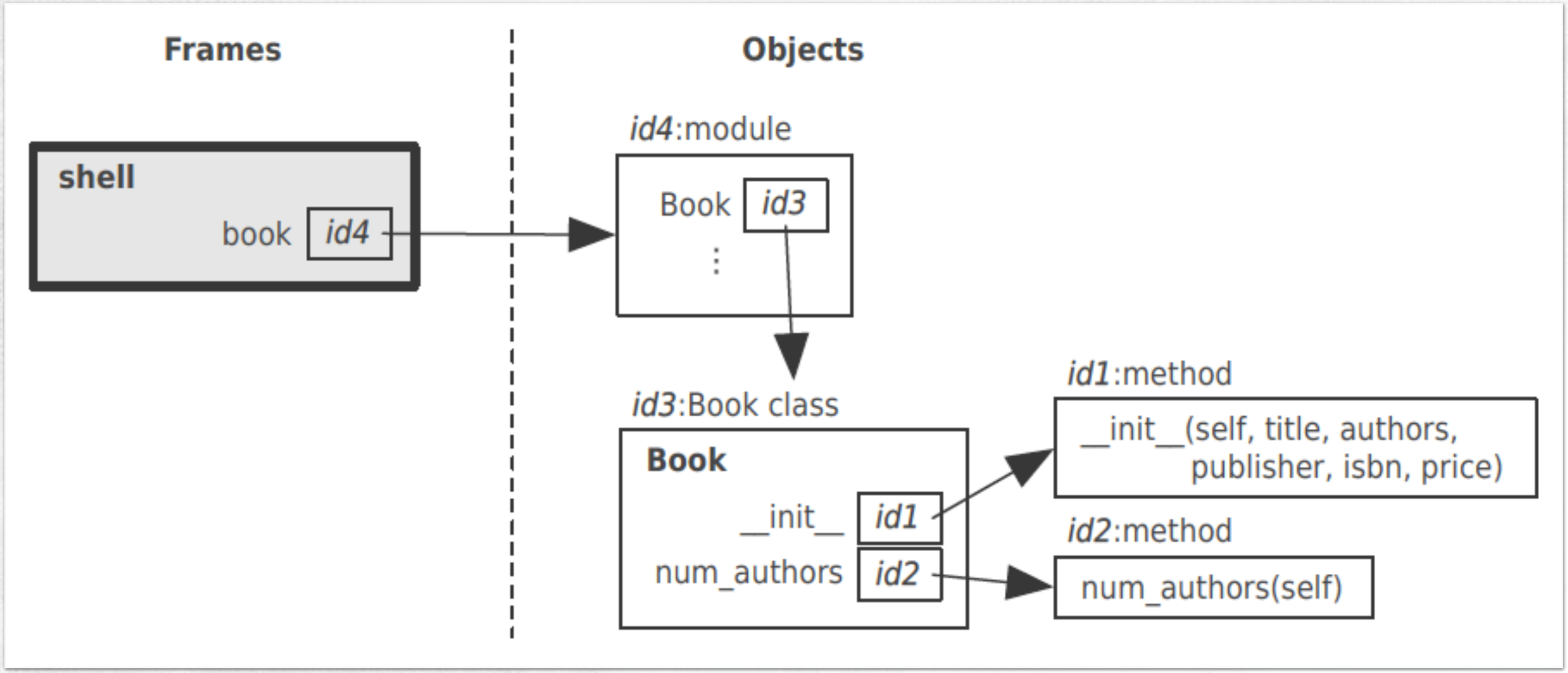
book.py

```
class Book:
    """Information about a book"""

    def __init__(self, title, authors, publisher, isbn, price):
        """(Book, str, list of str, str, str, number) -> NoneType
        Create a new book entitled title, written by the people in authors,
        published by publisher, with ISBN isbn and costing price dollars.
        """
        self.title = title
        self.authors = authors[:]
        self.publisher = publisher
        self.ISBN = isbn
        self.price = price


    def num_authors(self):
        """(Book) -> int

        Return the number of authors of this book.
        """
        return len(self.authors)
```

```
>>> import book
>>> ruby_book = book.Book()
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    ruby_book = book.Book()
TypeError: __init__() missing 5 required positional
arguments: 'title', 'authors', 'publisher', 'isbn',
and 'price'
>>> python_book = book.Book('Practical Programming',
['Campbell','Gries','Montojo'],'Pragmatic Bookshelf'
,'978-1-93778-545-1',25.0)
>>> python_book.title
'Practical Programming'
>>> python_book.authors
['Campbell', 'Gries', 'Montojo']
>>> python_book.price
25.0
>>> python_book.num_authors()
3
```

__init__ method (Constructor)

- The module book contains a single statement: the class definition.

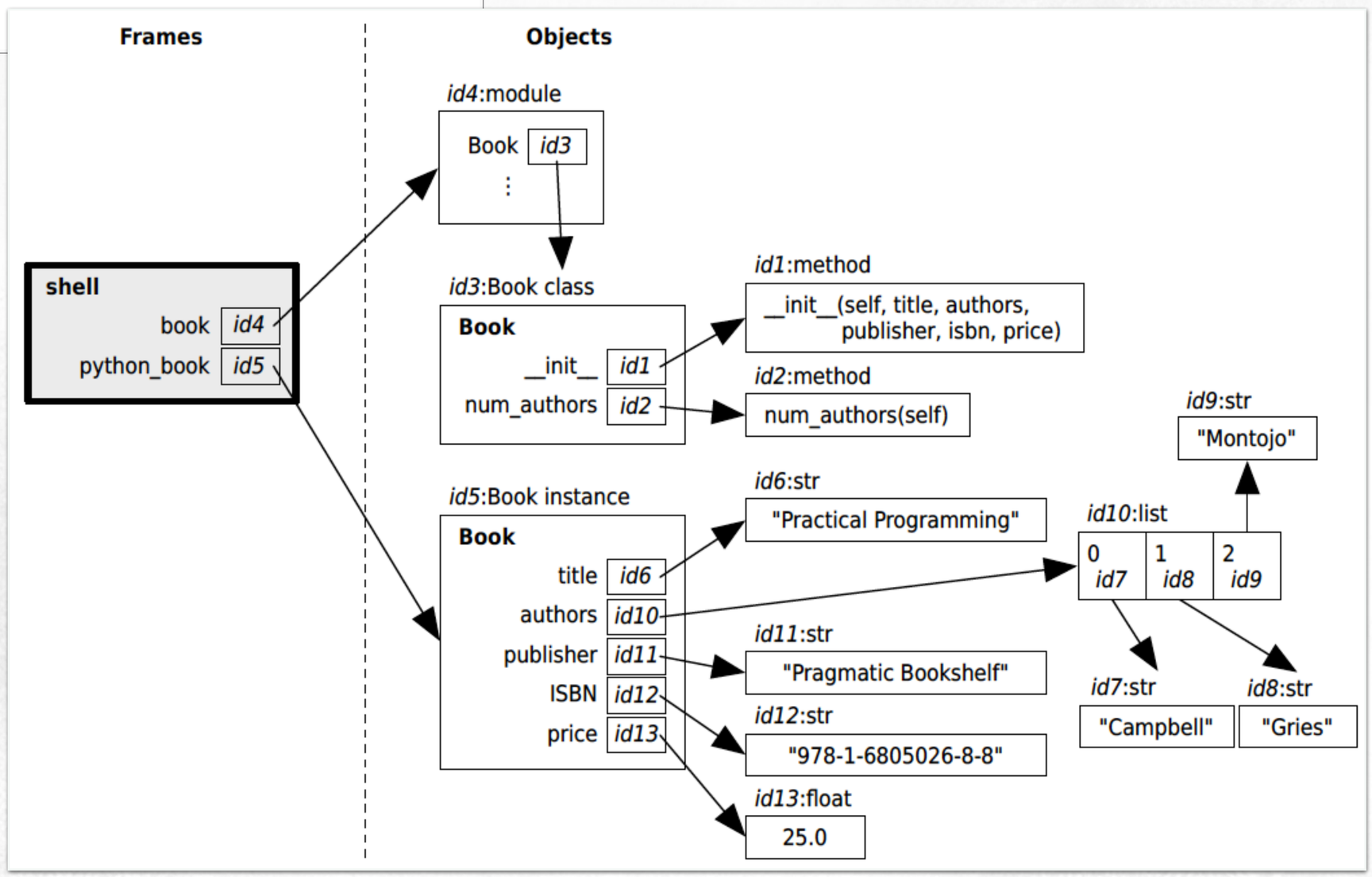


- Method __init__ is called whenever a Book object is created, to initialize the new object. (=a constructor)

__init__ method (Constructor)

- The steps that Python follows when creating an object:
  - It creates an object at a particular memory address.
  - It calls method __init__, passing in the new object into the parameter self.
  - It produces that object's memory address.

```
>>> import book
>>> python_book = book.Book('Practical Programming',
['Campbell','Gries','Montojo'],'Pragmatic Bookshelf'
,'978-1-93778-545-1',25.0)
```



Class variable vs Instance variable

```python
class Book:
    """Description about a book"""

    title = "none"
    authors = "none"

    def num_authors(self):
        """(Book) -> int
        Return the number of authors of this book.
        """
        return len(self.authors)

a = Book()
b = Book()

print(a.title) #none
print(b.title) #none

Book.title = "NONE"
print(Book.title) #NONE

print(a.title) #NONE
print(b.title) #NONE
```

```
none
none
NONE
NONE
NONE
```

```python
class Book:
    """Description about a book"""

    title = "none"

    def __init__(self):
        self.title = "____"
        self.authors = "____"

    def num_authors(self):
        """(Book) -> int
        Return the number of authors of this book.
        """
        return len(self.authors)

a = Book()
b = Book()

print(a.title) #____
print(b.title) #____

Book.title = "NONE"
print(Book.title) #NONE

print(a.title) #____
print(b.title) #____
```

```
____
____
NONE
____
____
```

__init__ method : reload (if modified)

```python
class Book:
    """Information about a book"""

    def __init__(self, title, authors, publisher, isbn, price):
        """(Book, str, list of str, str, str, number) -> NoneType
        Create a new book entitled title, written by the people in authors,
        published by publisher, with ISBN isbn and costing price dollars.
        """
        self.title = title
        self.authors = authors[:]
        self.publisher = publisher
        self.ISBN = isbn
        self.price = price


    def num_authors(self):
        """(Book) -> int

        Return the number of authors of this book.
        """
        return len(self.authors)
```

book.py

```
>>> import imp
>>> imp.reload(book)
<module 'book' from 'C:\\Users\\jiyoung\\AppData\\Lo
cal\\Programs\\Python\\Python35\\Scripts\\class\\boo
k.py'>
>>> ruby_book = book.Book()
>>> ruby_book.title
''
>>> python_book = book.Book('Practical Programming',
['Campbell','Gries','Montojo'],'Pragmatic Bookshelf'
,'978-1-93778-545-1',25.0)
>>> python_book.title
'Practical Programming'
```

Add __eq__ method into Book class

```
def __eq__(self, other):
    """ (Book, Book) -> bool
    Return True iff this book and other have the same ISBN.
    """
    return self.ISBN == other.ISBN
```

```
>>> python_book_1 = book.Book(
...     'Practical Programming',
...     ['Campbell', 'Gries', 'Montojo'],
...     'Pragmatic Bookshelf',
...     '978-1-93778-545-1',
...     25.0)
>>> python_book_2 = book.Book(
...     'Practical Programming',
...     ['Campbell', 'Gries', 'Montojo'],
...     'Pragmatic Bookshelf',
...     '978-1-93778-545-1',
...     25.0)
>>> python_book_1 == python_book_2
False
>>> python_book_1 == python_book_1
True
>>> python_book_2 == python_book_2
True
```

```
>>> python_book_1 = book.Book(
...     'Practical Programming', ['Campbell', 'Gries', 'Montojo'],
...     'Pragmatic Bookshelf', '978-1-93778-545-1', 25.0)
>>> python_book_2 = book.Book(
...     'Practical Programming', ['Campbell', 'Gries', 'Montojo'],
...     'Pragmatic Bookshelf', '978-1-93778-545-1', 25.0)
>>> survival_book = book.Book(
...     "New Programmer's Survival Manual", ['Carter'],
...     'Pragmatic Bookshelf', '978-1-93435-681-4', 19.0)
>>> python_book_1 == python_book_2
True
>>> python_book_1 == survival_book
False
```

Override vs Overload

- We can **override** an inherited method by defining a new version in our subclass. This replaces the inherited method so that it is no longer used.

```
def __eq__(self, other):
    """ (Book, Book) -> bool
    Return True iff this book and other have the same ISBN.
    """
    return self.ISBN == other.ISBN
```

- **Overloading** occurs when two or more methods in one class have the **same method name but different parameters**.

```
class Book:
    """Information about a book"""

    def __init__(self, title="", authors=[], publisher="", isbn="0", price=10.0):
        """(Book, str, list of str, str, str, number) -> NoneType
        Create a new book entitled title, written by the people in authors,
        published by publisher, with ISBN isbn and costing price dollars.
        """
        self.title = title
        self.authors = authors[:]
        self.publisher = publisher
        self.ISBN = isbn
        self.price = price
```

Lookup rules for a method call

- Look in the current object's class. If we find a method with the right name, use it.
- If we didn't find it, look in the superclass. Continue up the class hierarchy until the method is found.

Classes and objects

- Classes and objects are two of programming's power tools!
- They let good programmers do a lot in very little time.
- But with them, bad programmers can create a real mess…

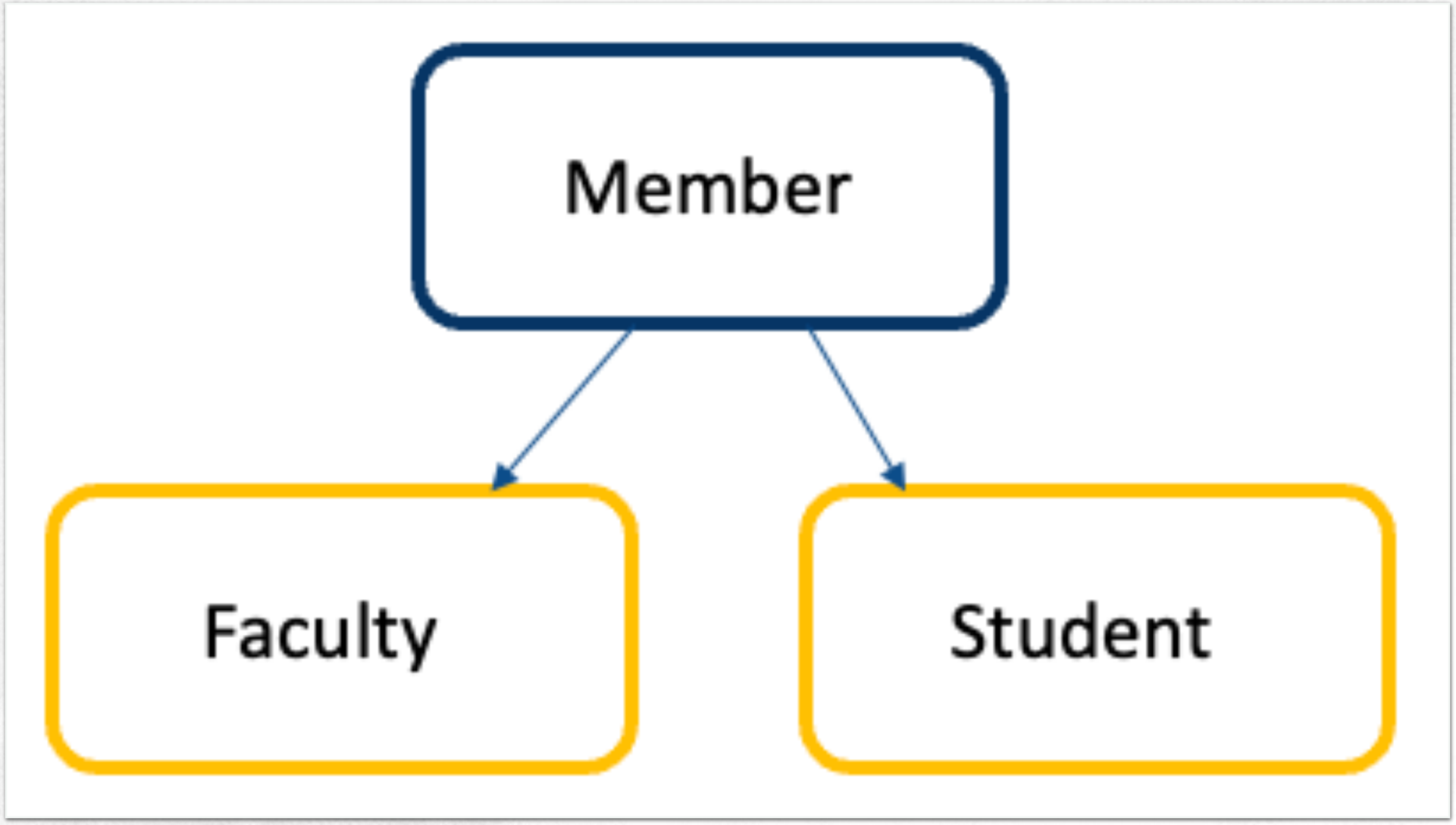
- Some concepts that will help you design reliable, reusable object-oriented software.

Inheritance (상속, 유전의 법칙)

- Whenever you create a class, you are using inheritance: your new class automatically inherits all of the attributes of class object.
- Just like a child inherits attributes from his/her parents.
- You can also declare that your new class is a subclass of some other class.

Inheritance: example

- Managing people at a university.



Inheritance: example

- Managing people at a university.

```
class Member:
    """A member of a university"""

    def __init__(self, name, address, email):
        """(Member, str, str, str) -> NoneType
        Create a new member named name,
        with home address and email address
        """
        self.name = name
        self.address = address
        self.email = email
```

```
class Faculty(Member):
    """A faculty member at a university"""


    def __init__(self, name, address, email, faculty_num):
        """(Faculty, str, str, str, str) -> NoneType
        Create a new faculty named name,
        with home address and email address,
        faculty number faculty_num and an empty list of courses.
        """
        super().__init__(name, address, email)
        self.faculty_number = faculty_num
        self.courses_teaching = []
```

```
class Student(Member):
    """A Student member at a university"""


    def __init__(self, name, address, email, student_num):
        """(Faculty, str, str, str, str) -> NoneType
        Create a new student named name,
        with home address and email address,
        student number student_num and an empty list of courses taken,
        and an empty list of current courses.
        """
        super().__init__(name, address, email)
        self.student_number = student_num
        self.courses_taken = []
        self.courses_taking = []
```

Inheritance: example

- Managing people at a university.

```
class Member:
    """A member of a university"""

    def __init__(self, name, address, email):
        """(Member, str, str, str) -> NoneType
        Create a new member named name,
        with home address and email address
        """
        self.name = name
        self.address = address
        self.email = email
```

```
class Faculty(Member):
    """A faculty member at a university"""


    def __init__(self, name, address, email, faculty_num):
        """(Faculty, str, str, str, str) -> NoneType
        Create a new faculty named name,
        with home address and email address,
        faculty number faculty_num and an empty list of courses.
        """
        super().__init__(name, address, email)
        self.faculty_number = faculty_num
        self.courses_teaching = []
```

Inheritance: example

- Managing people at a university.

```
class Member:
    """A member of a university"""

    def __init__(self, name, address, email):
        """(Member, str, str, str) -> NoneType
        Create a new member named name,
        with home address and email address
        """
        self.name = name
        self.address = address
        self.email = email
```

```
class Student(Member):
    """A Student member at a university"""


    def __init__(self, name, address, email, student_num):
        """(Faculty, str, str, str, str) -> NoneType
        Create a new student named name,
        with home address and email address,
        student number student_num and an empty list of courses taken,
        and an empty list of current courses.
        """
        super().__init__(name, address, email)
        self.student_number = student_num
        self.courses_taken = []
        self.courses_taking = []
```

Inheritance: example

```
>>> snape = Faculty('Severus Snape','Seoul','snape@k
hu.ac.kr','1234')
>>> snape.name
'Severus Snape'
>>> snape.email
'snape@khu.ac.kr'
>>> snape.faculty_number
'1234'
```

```
>>> harry = Student('Harry Potter','London','hpotter
@khu.ac.kr','4321')
>>> harry.name
'Harry Potter'
>>> harry.email
'hpotter@khu.ac.kr'
>>> harry.student_number
'4321'
```

Add features to the superclass

```
>>> snape = Faculty('Severus Snape','Seoul','snape@k
hu.ac.kr','1234')
>>> harry = Student('Harry Potter','London','hpotter
@khu.ac.kr','4321')
>>> print(snape)
Severus Snape
Seoul
snape@khu.ac.kr
>>> print(harry)
Harry Potter
London
hpotter@khu.ac.kr
```

```
>>> str(harry)
'Harry Potter\nLondon\nhpotter@khu.ac.kr'
```

```
class Member:
    """A member of a university"""

    def __init__(self, name, address, email):
        """(Member, str, str, str) -> NoneType
        Create a new member named name,
        with home address and email address
        """
        self.name = name
        self.address = address
        self.email = email

    def __str__(self):
        """(Member) -> str
        Return a string representation of this Member
        """
        rep = "{}\n{}\n{}".format(self.name, self.address, self.email)
        return rep
```

Add features to the subclass

```
>>> snape = Faculty('Severus Snape',
'Seoul','snape@khu.ac.kr','1234')
>>> print(snape)
Severus Snape
Seoul
snape@khu.ac.kr
1234
Courses:[]
```

```
class Faculty(Member):
    """A faculty member at a university"""


    def __init__(self, name, address, email, faculty_num):
        """(Faculty, str, str, str, str) -> NoneType
        Create a new faculty named name,
        with home address and email address,
        faculty number faculty_num and an empty list of courses.
        """
        super().__init__(name, address, email)
        self.faculty_number = faculty_num
        self.courses_teaching = []

    def __str__(self):
        """(Faculty) -> str
        Return a string representation of this Faculty
        """
        member_string = super().__str__()
        rep = "{}\n{}\nCourses:{}".format(
            member_string, self.faculty_number, self.courses_teaching)
        return rep
```

Private variables and methods

- Public is default
  - no access limitation
  - open to public
- Private is hidden from outside of class
  - supported via “__” prefix
  - Getter and Setter methods are recommended

```python
class MyClass:
    def __init__(self):
        self.__privateVariable = 1

    def __privateMethod(self):
        self.__privateVariable = 2

    def getMethod(self):
        return self.__privateVariable

    def setMethod(self, givenVariable):
        self.__privateVariable = givenVariable

myObject = MyClass()
print(myObject.getMethod()) # 1

myObject.setMethod(100)
print(myObject.getMethod()) # 100

print(myObject.__privateVariable) # ERROR
```

Private variables and methods

- Guarantees Information Hiding (or encapsulation) design principle of object oriented programming
  - Information hiding is the principle of segregation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed.

Polymorphism (다형성)

- Polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

```python
class Member:
    def __init__(self, name, address, email):
        self.name = name
    def saySomething(self):
        print("[Member] saySomething()")

class Faculty(Member):
    def __init__(self, name, address, email, faculty_num):
        super().__init__(name, address, email)
        self.faculty_number = faculty_num
        self.courses_teaching = []
    def saySomething(self):
        print("[Faculty] saySomething() of {}", self.name)

class Student(Member):
    def __init__(self, name, address, email, student_num):
        super().__init__(name, address, email)
        self.student_number = student_num
        self.courses_taken = []
        self.courses_taking = []
    def saySomething(self):
        print("[Student] saySomething() of {}", self.name)

member1 = Faculty('Paul Gries','Ajax','pgries@cs.toronto.edu','1234')
member2 = Student('Jen Campbell','Toronto','campbell@cs.toronto.edu','4321')
member3 = Faculty('Sungwon Lee','Python','drsungwon@khu.ac.kr','1357')

members = [member1,member2,member3]

for member in members:
    member.saySomething()
```

```
[Faculty] saySomething() of {} Paul Gries
[Student] saySomething() of {} Jen Campbell
[Faculty] saySomething() of {} Sungwon Lee
```

Summary

- In object-oriented languages, new types are defined by creating classes. Classes support encapsulation; in other words, they combine data and the operations on it so that other parts of the program can ignore implementation details.
- New classes can be defined by inheriting features from existing ones. The new class can override the features of its parent and/or add new features.
- When a method is defined in a class, its first argument must be a variable that represents the object the method is being called on. By convention, this argument is called `self`.
- Some methods have special predefined meanings in Python; to signal this, their names begin and end with two underscores. Some of these methods are called when constructing objects (__init__) or converting them to strings (__str__ and __repr__); others, like __add__ and __sub__, are used to imitate arithmetic.

Thank you

경희대학교
KYUNG HEE UNIVERSITY