Object oriented programming

- classes, objects
- self
- construction
- encapsulation

Object Oriented Programming

- **Programming paradigm**, other paradigms are e.g.
  - *functional programming* where the focus is on functions, lambda's and higher order functions, and
  - *imperative programming* focusing on sequences of statements changing the state of the program
- Core concepts are **objects**, **methods** and **classes**,
  - allowing one to construct *abstract data types*, i.e. user *defined types*
  - objects have states
  - methods manipulate objects, defining the interface of the object to the rest of the program
- OO supported by many programming languages, including Python

Object Oriented Programming - History
(selected programming languages)

Mid 1960's **Simula 67**

(Ole-Johan Dahl and Kristen Nygaard, Norsk Regnesentral Oslo)
Introduced classes, objects, virtual procedures

1970's **Smalltalk** (Alan Kay, Dan Ingalls, Adele Goldberg, Xerox PARC)

Object-oriented programming, fully dynamic system
(opposed to the static nature of Simula 67)

1985 **Eiffel** (Bertrand Meyer, Eiffel Software)

Focus on software quality, capturing the full software cycle

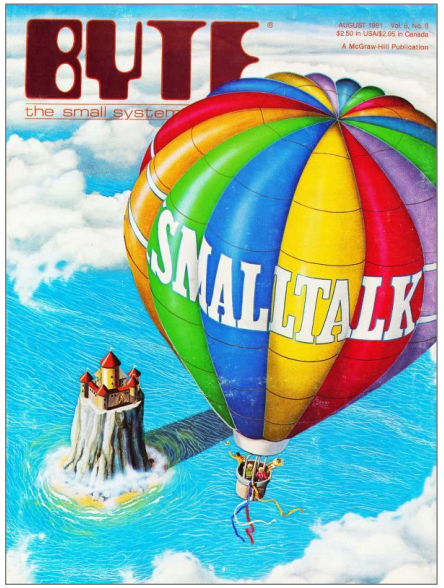
1985 **C++** (Bjarne Stroustrup [MSc Aarhus 1975], AT&T Bell Labs)

1995 **Java** (James Gosling, Sun)

2000 **C#** (Anders Hejlsberg (studied at DTU) et al., Microsoft)

1991 **Python** (Guido van Rossum)

Multi-paradigm programming language, fully dynamic system

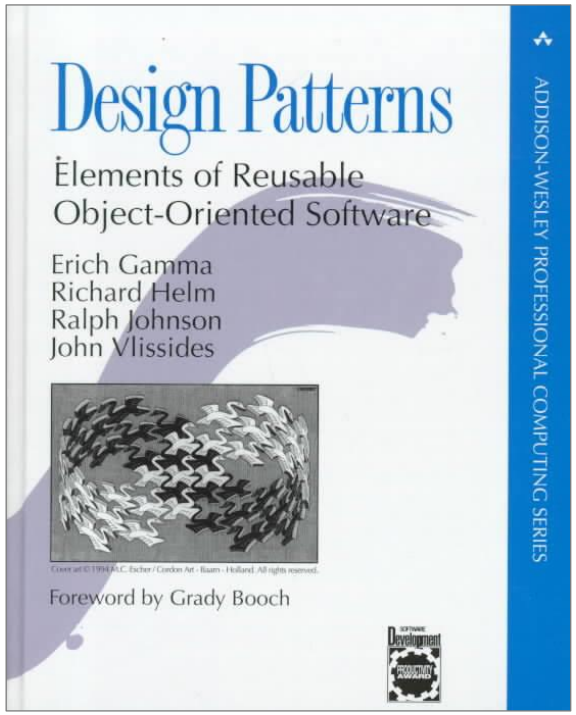


Byte Magazine, August 1981

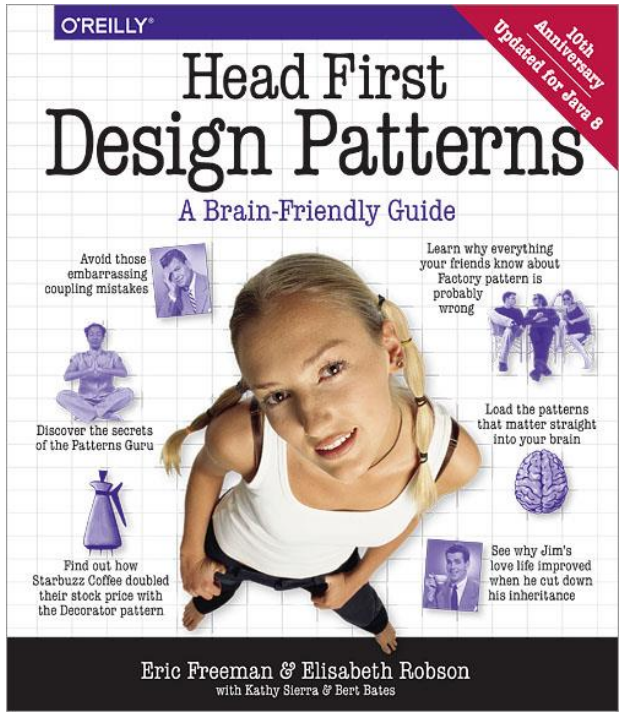
Note: Java, C++, Python, C# are among Top 5 on TIOBE January 2023 index of popular languages (only non OO language among Top 5 was C)

Design Patterns (not part of this course)

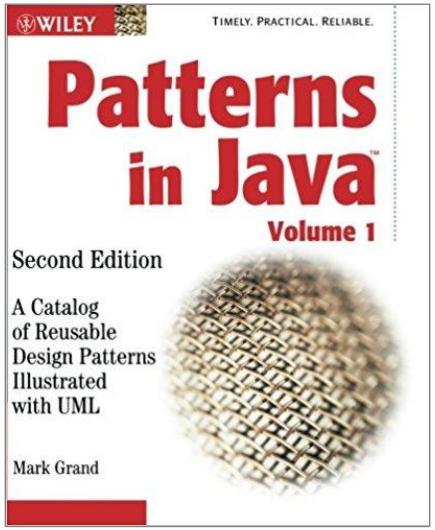
reoccuring patterns in software design



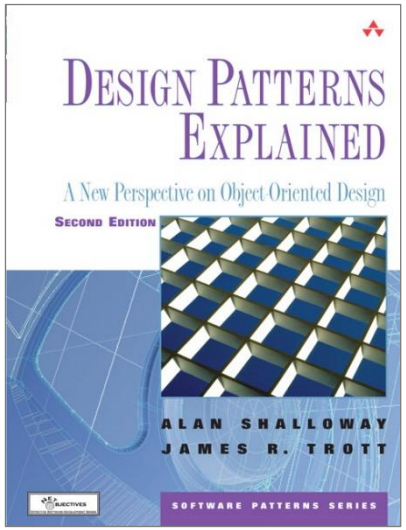
The Classic book 1994
(C++ cookbook)



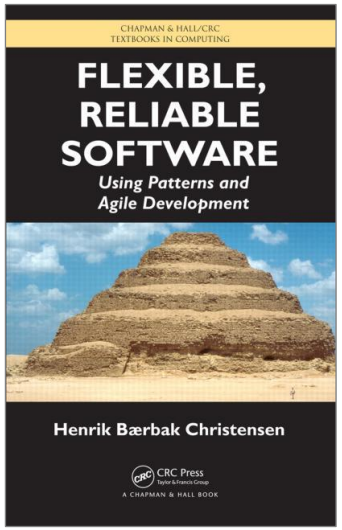
A very alternative book 2004
(Java, very visual)



Java cookbook 2003



Java textbook 2004



Java textbook 2010

...and many more books on the topic of Design Patterns, also with Python

Some known classes, objects, and methods

Type / class	Objects	Methods (examples)
int	0 -7 42 1234567	.__add__(x), .__eq__(x), .__str__()
str	"" 'abc' '12_ a'	.isdigit(), .lower(), .__len__()
list	[] [1,2,3] ['a', 'b', 'c']	.append(x), .clear(), .__mul__(x)
dict	{'foo' : 42, 'bar' : 5}	.keys(), .get(), .__getitem__(x)
NoneType	None	.__str__()

Example:

The function `str(obj)` calls the methods `obj.__str__()` or `obj.__repr__()`, if `obj.__str__` does not exist.

`print` calls `str`.

Python shell

```
> 5 + 7  # + calls .__add__(7)
| 12
> (5).__add__(7)  # eq. to 5 + 7
| 12
> (7).__eq__(7)  # eq. to 7 == 7
| True
> 'aBCd'.lower()
| 'abcd'
> 'abcde'.__len__()
  # .__len__() called by len(...)
| 5
> ['x', 'y'].__mul__(2)
| ['x', 'y', 'x', 'y']
> {'foo' : 42}.__getitem__('foo')
  # eq. to {'foo' : 42}['foo']
| 42
> None.__str__()  # used by str(...)
| 'None'
> 'abc'.__str__(), 'abc'.__repr__()
| ('abc', "'abc'")
```

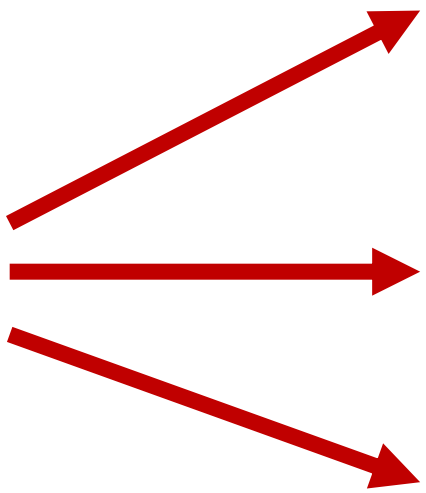
Classes and Objects

class
(type)

```
class Student
```

class methods

```
set_name(name)
set_id(student_id)

get_name()
get_id()
```



creating **instances** of class `Student` using **constructor** `Student()`

objects
(instances)

```
student_DD
name = 'Donald Duck'
id = '107'
```

data attributes

```
student_MM
name = 'Mickey Mouse'
id = '243'
```

```
student_SM
name = 'Scrooge McDuck'
id = '777'
```

docs.python.org/3/tutorial/classes.html

Using the `Student` class

student.py

```
student_DD = Student()
student_MM = Student()
student_SM = Student()

student_DD.set_name('Donald Duck')
student_DD.set_id('107')

student_MM.set_name('Mickey Mouse')
student_MM.set_id('243')

student_SM.set_name('Scrooge McDuck')
student_SM.set_id('777')

students = [student_DD, student_MM, student_SM]

for student in students:
    print(student.get_name(),
          "has id",
          student.get_id())
```

Python shell

```
Donald Duck has id 107
Mickey Mouse has id 243
Scrooge McDuck has id 777
```

Call **constructor** for class `Student`. Each call returns a new `Student` object.

Call class methods to set data attributes

Call class methods to read data attributes

class `Student`

name of class

docstring containing documentation for class

class definitions start with the keyword `class`

student.py

```
class Student:
    '''Documentation of class'''

    def set_name(self, name):
        self.name = name

    def set_id(self, student_id):
        self.id = student_id

    def get_name(self):
        return self.name

    def get_id(self):
        return self.id
```

the first argument to all class methods is a reference to the object called upon, and by convention the first argument should be named `self`.

use `self.` to access an attribute of an object or class method (attribute reference)

often called **mutator methods**, since they change the state of an object

often called **accessor methods**, since they only read the state of an object

class method definitions start with keyword `def` (like normal function definitions)

Note In other OO programming languages the explicit reference to `self` is not required (in Java and C++ `self` is the keyword `this`)

When are object attributes initialized ?

Python shell

```
> x = Student()
> x.set_name("Gladstone Gander")
> x.get_name()
| 'Gladstone Gander'
> x.get_id()
| AttributeError: 'Student' object has no attribute 'id'
```

- Default behaviour of a class is that instances are created with no attributes defined, but has access to the attributes / methods of the class
- In the previous class `Student` both the `name` and `id` attributes were first created when set by `set_name` and `set_id`, respectively

Class construction and `__init__`

- When an object is created using *`class_name`*`()` it's initializer metod `__init__` is called.
- To initialize objects to contain default values, (re)define this function.

student.py

```
class Student:
    def __init__(self):
        self.name = None
        self.id = None

    ... previous method definitions ...
```

Question – What is printed ?

Python shell

```
> class C:
      def __init__(self):
            self.v = 0
      def f(self):
            self.v = self.v + 1
            return self.v
> x = C()
> print(x.f() + x.f())
```

a) 1

b) 2


c) 3

d) 4

e) 5

f) Don’t know

`__init__` with arguments

- When creating objects using *`class_name(args)`* the initializer method is called as `__init__(`*`args`*`)`
- To initialize objects to contain default values, (re)define this function to do the appropriate initialization

student.py

```
class Student:
    def __init__(self, name=None, student_id=None):
        self.name = name
        self.id = student_id

    ... previous method definitions ...
```

Python shell

```
> p = Student("Pluto")
> print(p.get_name())
| Pluto
> print(p.get_id())
| None
```

Are accessor and mutator methods necessary ?

No - but good programming style

Python shell

```
> p = Pair(3, 5)
> p.sum()
| 8
> p.set_a(4)
> p.sum()
| 9
> p.a      # access object attribute
| 4
> p.b = 0 # update object attribute
> p.sum()
| 9        # the_sum not updated
```

pair.py

```
class Pair:
    """ invariant: the_sum = a + b """
    def __init__(self, a, b):
        self.a = a
        self.b = b
        self.the_sum = self.a + self.b
    def set_a(self, a):
        self.a = a
        self.the_sum = self.a + self.b
    def set_b(self, b):
        self.b = b
        self.the_sum = self.a + self.b
    def sum(self):
        return self.the_sum
```

constructor

mutator

accessor

Defining order on instances of a class (sorting)

- To define an order on objects, define the “<“ operator by defining `__lt__`
- When ”<” is defined a list `L` of students can be sorted using `sorted(L)` and `L.sort()`

student.py

```
class Student:
    def __lt__(self, other):
        return self.id < other.id

    ... previous method definitions ...
```

Python shell

```
> student_DD < student_MM
| True
> [x.id for x in students]
| ['243', '107', '777']
> [x.id for x in sorted(students)]
| ['107', '243', '777']
```

Converting objects to `str`

- To be able to convert an object to a string using `str(`*`object`*`)`, define the method `__str__`
- `__str__` is e.g. used by `print`

Student_constructor.py

```
class Student:
    def __str__(self):
        return ("Student('%s', '%s')"
                % (self.name, self.id))

    ... previous method definitions ...
```

Python shell

```
> print(student_DD)  # without __str__
| <__main__.Student object at 0x03AB6B90>
> print(student_DD)  # with __str__
| Student('Donald Duck', '107')
```

Nothing is private in Python

- Python does not support **hiding information** inside objects
- Recommendation is to start attributes with underscore, if these should be used only locally inside a class, i.e. be considered ”private”
- **PEP8:** “Use one leading underscore only for non-public methods and instance variables“

private_attributes.py

```
class My_Class:
    def set_xy(self, x, y):
        self._x = x
        self._y = y

    def get_sum(self):
        return self._x + self._y

obj = My_Class()
obj.set_xy(3, 5)

print("Sum =", obj.get_sum())
print("_x =", obj._x)
```

Python shell

```
| Sum = 8
| _x = 3
```

C++ private, public

C++ vs Python

1. argument types
2. return types
3. `void` = `NoneType`
4. `private` / `public` access specifier
5. types of data attributes
6. data attributes must be defined in class
7. object creation
8. no `self` in class methods

private_attributes.cpp

```
#include <iostream>

using namespace std;

class My_Class {
private:④
 ⑤int x, y;⑥
public:④      ⑧①       ①
②③void set_xy(int a, int b) {
     x = a;
     y = b
   };           ⑧
② int get_sum() {
     return x + y;
   };
};

main() {
⑦ My_Class obj;
   obj.set_xy(3, 5);
   cout << "Sum = " << obj.get_sum() << endl;
   cout << "x = " << obj.x << endl;
}
```

invalid reference

Java private, public

Java vs Python

1. argument types
2. return types
3. `void` = `NoneType`
4. `private` / `public` access specifier
5. types of data attributes
6. data attributes must be defined in class
7. object creation
8. no `self` in class methods

private_attributes.java

```
class My_Class {
④ private int x, y;
④ public void set_xy(int a, int b) {
    x = a; y = b;
  }
④ public int get_sum() { return x + y; };
};
class private_attributes {
  public static void main(String args[]){
  ⑦ My_Class obj = new My_Class();
    obj.set_xy(3, 5);
    System.out.println("Sum = " + obj.get_sum());
    System.out.println("x = " + obj.x);
  }
}
```

⑤ ⑥ ⑧ ① ① ② ③ ② ⑧

invalid reference

Name mangling (partial privacy)

- Python handles references to class attributes inside a class definition with *at least two leading underscores and at most one trailing underscore* in a special way: `__attribute` is textually replaced by `_classname__attribute`

- Note that [Guttag, p. 200] states ”that attribute is not visible outside the class” – which only is partially correct (see example)

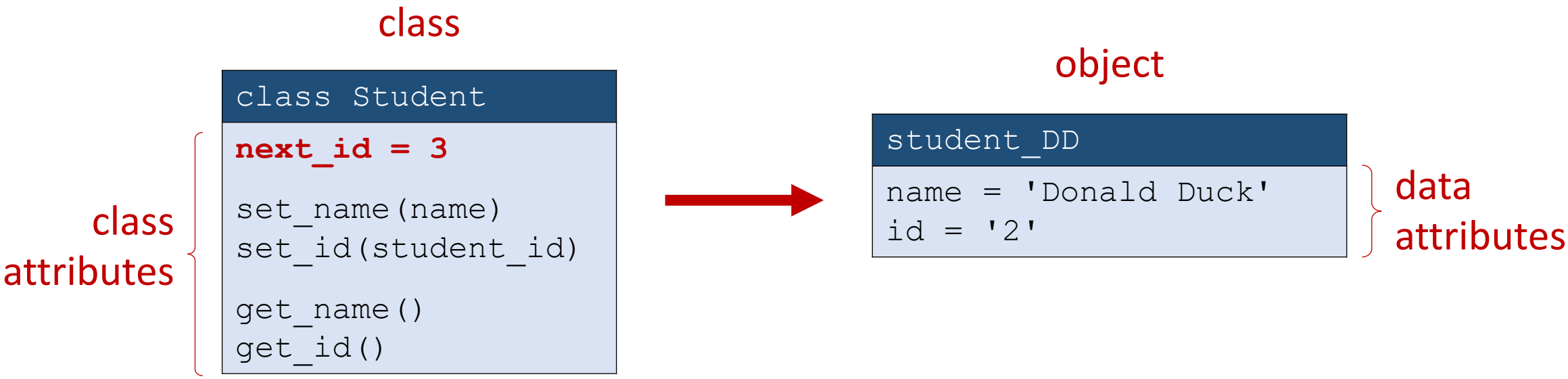
name_mangeling.py

```
class MySecretBox:
    def __init__(self, secret):
        self.__secret = secret
```

Python shell

```
> x = MySecretBox(42)
> print(x.__secret)
| AttributeError: 'MySecretBox'
  object has no attribute
  '__secret'
> print(x._MySecretBox__secret)
| 42
```

Class attributes



- *obj.attribute* first searches the objects attributes to find a match, if no match, continuous to search the attributes of the class
- Assignments to *obj.attribute* are always to the objects attribute (possibly creating the attribute)
- Class attributes can be accesed directly as *class.attribute* (or *obj.__class__.attribute*)

Class data attribute

- **`next_id`** is a class attribute
- Accessed using **`Student.next_id`**
- The lookup ① can be replaced with **`self.next_id`**, since only the class has this attribute, looking up in the object will be propagated to a lookup in the class attributes
- In the update ② it is crucial that we update the class attribute, since otherwise the incremented value will be assigned as an object attribute

  (What will the result be?)

student_auto_id.py

```
class Student:
    next_id = 1  # class attribute
    def __init__(self, name):
        self.name = name
        self.id = str(Student.next_id)  ①
        Student.next_id += 1  ②
    def get_name(self):
        return self.name
    def get_id(self):
        return self.id

students = [Student('Scrooge McDuck'),
            Student('Donald Duck'),
            Student('Mickey Mouse')]
for student in students:
    print(student.get_name(),
          "has student id",
          student.get_id())
```

Python shell

```
Scrooge McDuck has student id 1
Donald Duck has student id 2
Mickey Mouse has student id 3
```

Question – What does `obj.get()` return ?

Python shell

```
> class MyClass:
      x = 2

      def get(self):
          self.x = self.x + 1
          return MyClass.x + self.x

> obj = MyClass()
> print(obj.get())
| ?
```

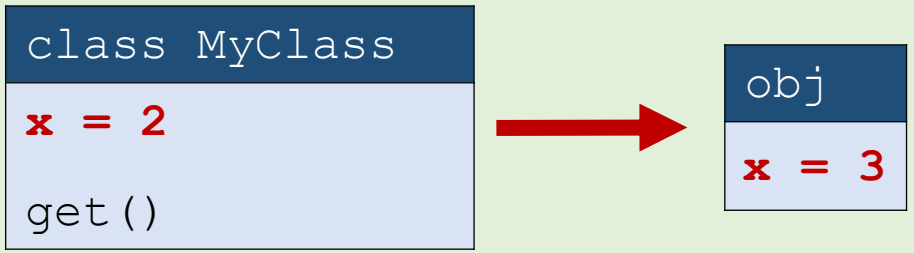
a) 4

b) 5

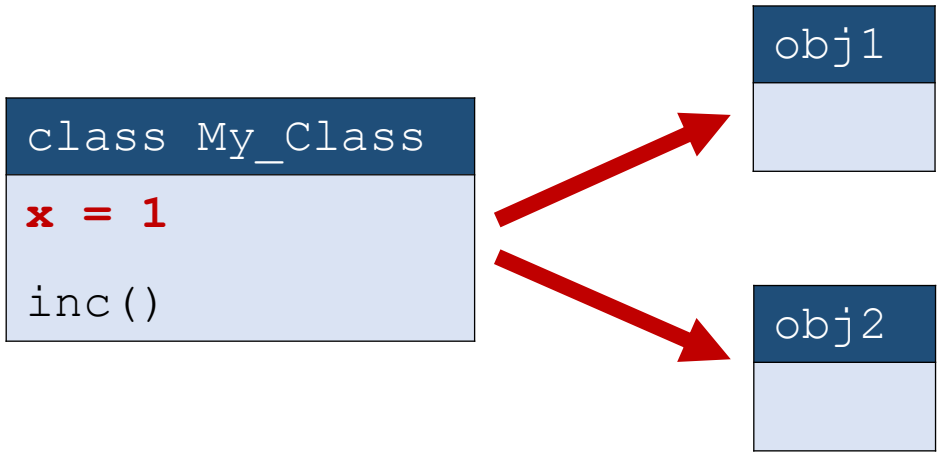
c) 6

d) UnboundLocalError

e) Don't know



Class data attribute example (in Python)




```
class_attributes.py
class My_Class:
    x = 1  # class attribute
    def inc(self):
        My_Class.x = self.x + 1
obj1 = My_Class()
obj2 = My_Class()
obj1.inc()
obj2.inc()
print(obj1.x, obj2.x)
```

```
Python shell
| 3 3
```

- Note that `My_Class.x` and `self.x` refer to the same class attribute (since `self.x` has never been assigned a value)

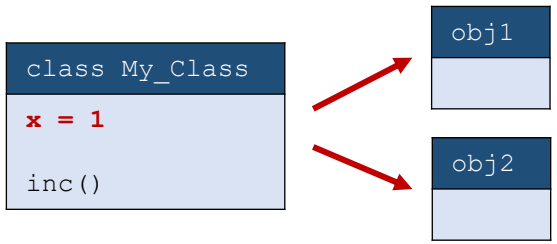
__dict__, __name__ and __class__

Python shell

```
> MM = Student('Mickey Mouse')
> MM.__dict__                                  # objects attributes
| {'name': 'Mickey Mouse', 'id': '1'}
> MM.__class__                                 # objects class (reference to object of type class)
| <class '__main__.Student'>
> Student.__name__                             # class name (string)
| 'Student'
> Student.__dict__                             # class attributes
| mappingproxy({
  '__module__': '__main__',                                          # module where class defined
  'next_id': 2,                                                      # class data attriute
  '__init__': <function Student.__init__ at 0x000002831344CD30>,     # class method
  'get_name': <function Student.get_name at 0x000002831344CE50>,     # class method
  'get_id': <function Student.get_id at 0x000002831344CEE0>,         # class method
  '__dict__': <attribute '__dict__' of 'Student' objects>,           # attributes of class
  '__weakref__': <attribute '__weakref__' of 'Student' objects>,     # (for garbage collecting)
  '__doc__': None                                                    # docstring
  })
```

Java `static`

- In Java *class attributes,* i.e. attribute values shared by all instances, are labeled **`static`**
- Python allows both class and instance attributes with the same name – in Java at most one of them can exist



static_attributes.java

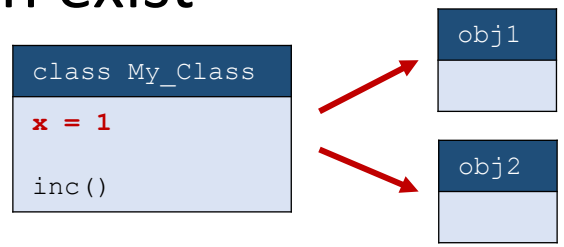
```
class My_Class {
    public static int x = 1;
    public void inc() { x += 1; };
}

class static_attributes {
    public static void main(String args[]){
        My_Class obj1 = new My_Class();
        My_Class obj2 = new My_Class();
        obj1.inc();
        obj2.inc();
        System.out.println(obj1.x);
        System.out.println(obj2.x);
    }
}
```

Java output

```
3
3
```

C++ `static`

- In C++ *class attributes,* i.e. attribute values shared by all instances, are labeled **`static`**
- ISO C++ forbids in-class initialization of non-const static member
- Python allows both class and instance attributes with the same name – in C++ at most one of them can exist



static_attributes.cpp

```
#include <iostream>
using namespace std;
class My_Class {
public:
  static int x; // "= 1" is not allowed
  void inc() { x += 1; };
};
int My_Class::x = 1; // class initialization
int main(){
  My_Class obj1;
  My_Class obj2;
  obj1.inc();
  obj2.inc();
  cout << obj1.x << endl;
  cout << obj2.x << endl;
}
```

C++ output

```
3
3
```

Constants

- A simple usage of class data attributes is to store a set of constants (but there is nothing preventing anyone to chance these values)

Python shell

```
> class Color:
      RED   = "ff0000"
      GREEN = "00ff00"
      BLUE  = "0000ff"
> Color.RED
| 'ff0000'
```

PEP8 Style Guide for Python Code (some quotes)

- Class names should normally use the CapWords convention.
- Always use self for the first argument to instance methods.
- Use one leading underscore only for non-public methods and instance variables.
- For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods.
- Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

www.python.org/dev/peps/pep-0008/

Some methods many classes have

Method	Description
`__eq__(self, other)`	Used to test if two elements are equal Two elements where __eq__ is true must have equal __hash__
`__str__(self)`	Used by `str` and `print`
`__repr__(self)`	Used by `repr`, e.g. for printing to the IDE shell (usually something that is a valid Python expression for `eval()`)
`__len__(self)`	Length (integer) of object, e.g. lists, strings, tuples, sets, dictionaries
`__doc__(self)`	The docstring of the class
`__hash__(self)`	Returns hash value (integer) of object Dictionary keys and set values must have a __hash__ method
`__lt__(self, other)`	Comparison (less than, <) used by `sorted` and `sort()`
`__init__(self,...)`	Class initializer

Overloading operators: www.python-course.eu/python3_magic_methods.php