PYTHON PROGRAMMING

2023/10/04 12:00 PM

CONTENTS

EXCEPTION HANDLING

- Exception
- Try-Except
- Exception Handling

Exception

- something that does not conform to the norm

- Unhandled exception raised → program terminates/crashes
  - ✓ IndexError
  - ✓ TypeError
  - ✓ NameError
  - ✓ ValueError

- Ex) IndexError

```
test = [1, 2, 3]
test[3]
```

→ : list index out of range

Exception – ZeroDivisionError (ex.)

```
num_success = 10
num_failure = int(input("Number of failures: "))
my_ratio = num_success / num_failure
print('My ratio is', my_ratio)
```

- If `num_failure` gets `0` for input, **ZeroDivisionError** is raised. (line 3)

Exception – ValueError (ex.)

```
val = int(input('Enter an integer: '))
print('The square of', val, 'is', val**2)
```

- If `val` gets a string for input, **ValueError** is raised. (line 1)

Try – Except

```
try:
    block of code to do when no error
except ErrorType:
    block of code when error happens
```

- Exception handling by using `try-except` block.
- Exceptions must be one or more.
- `try` cannot be used without `except`

Try – Except Examples

- ZeroDivisionError

```
try:
        my_ratio = num_success / num_failures
        print('My ratio is', my_ratio)
except ZeroDivisionError:
        print('No failures! The ratio is undefined')
```

Try – Except Examples (cont.)

- ValueError

```
while True:
    val = input('Enter an integer: ')
    try:
        val = int(val)
        print('The square of', val, 'is', val**2)
    except ValueError:
        print(val, 'is not an integer!')
```

Try – Except Examples (cont.)

```
def readInt( ):
     while True:
          val = input('Enter an integer: ')
          try:
               return(int(val))
          except ValueError:
               print(val, 'is not an integer!')
```

Try – Except Examples (cont.)

```
def readVal(valType, requestMsg, errorMsg):
    while True:
        val = input(requestMsg + ' ')
        try:
            return(valType(val))
        except ValueError:
            print(val, errorMsg)

readVal(int, 'Enter an integer: ', 'is not an integer')
```

⇒ function `readVal( )` is **polymorphic**; works for **many types** of arguments

Exception Handling

```
raise exceptionName(argument)
```

- Using exceptions for control flow.
- `exceptionName`: usually one of the built-in exceptions (e.g., `ValueError`)
- `argument`: usually a string to print error messages.

Exception Handling (cont.)

1. Can be implemented by `try-except` block.

```
def getRatios(vect1, vect2):
        ratios = []
        for index in range(len(vect1)):
            try:
                ratios.append(vect1[index]/vect2[index])
            except ZeroDivisionError:
                ratios.append(float('nan'))
            except:
                raise ValueError('wrong arguments!')
        return ratios
```

Exception Handling (cont.)

2. Without using a try-except block.

```
def getRatios(vect1, vect2):
    ratios = []
    if len(vect1) != len(vect2):
        raise ValueError('Wrong arguments')
    for index in range(len(vect1)):
        vect1E = vect1[index]
        vect2E = vect2[index]
        if (type(vect1E) not in (int, float)) or (type(vect2E) not in (int, float)):
                raise ValueError('Wrong arguments')
        if vect2Elem == 0.0:
                ratios.append(float('NaN')) #NaN: Not a Number
        else:
                ratios.append(vect1E/vect2E)
    return ratios
```

CLASSES

- Object
- Class
- Class Instantiation and Attribute Reference
- Special Methods in Classes

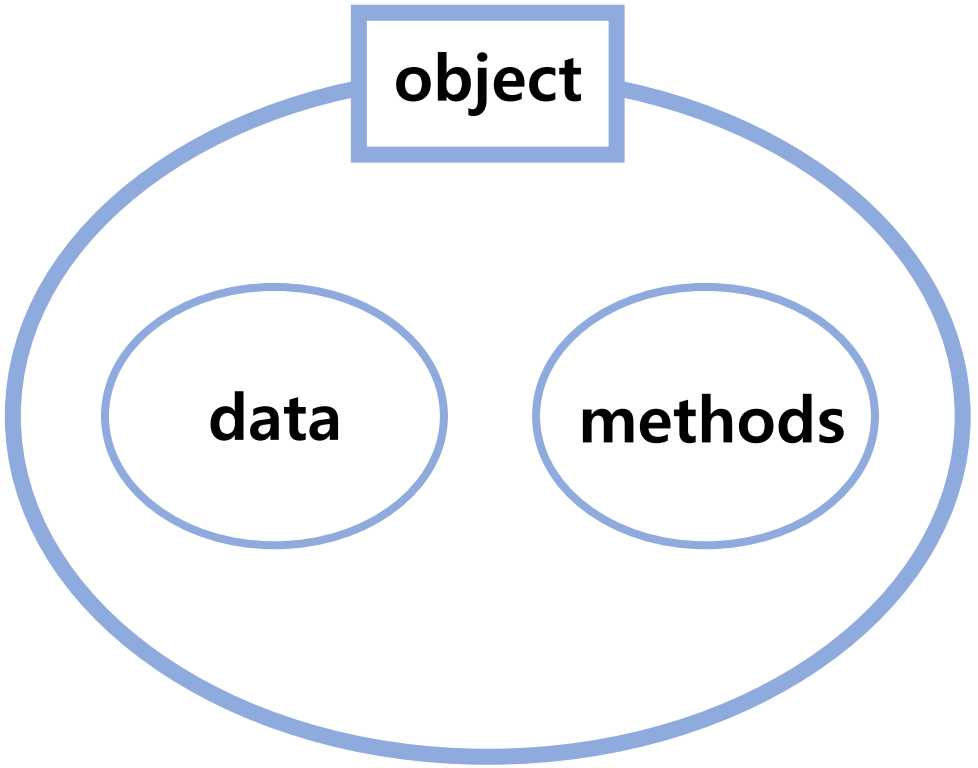
Object

- Objects are the core things that Python programs manipulate.

- Every object has a **type** that defines the kinds of things that programs can do with that object.

- The key to **object-oriented programming** is thinking about objects as collections of both *data* and the *methods that operate on that data.*

Object (cont.)



- An object consists of data and methods.
- A function defined within a class is called a **method**.

Object (cont.)

- An **abstract data type** is a set of objects and the operations on those objects.

- These are bound together so that one can pass an object from one part of a program to another.

- In doing so, provide access not only to the data attributes of the object but also to operations that make it easy to manipulate that data.

Object (cont.)

- The specifications of those operations define an **interface** between the abstract data type and the rest of the program.

- The **interface** defines the behavior of the operations

  — *what they do*, but not how they do it.

- The interface provides an **abstraction barrier** that isolates the rest of the program from the data structures, algorithms, and code involved in providing a *realization* of the type abstraction.

Class

`class` *`classname`*

- In Python, data abstraction is implemented using **classes**.
- Has data attributes and method attributes.
- A class definition creates an object of type `type` and associates with that class object a set of objects of type `instancemethod`.

Class – Example

```
class IntSet(object):
    """An intSet is a set of integers"""
    #Information about the implementation (not the abstraction)
    #Value of the set is represented by a list of ints, self.vals.
    #Each int in the set occurs in self.vals exactly once.

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if e not in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
           Returns True if e is in self, and False otherwise"""
        return e in self.vals
```

```
    def remove(self, e):
        """Assumes e is an integer and removes e from self
           Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def getMembers(self):
        """Returns a list containing the elements of self.
           Nothing can be assumed about the order of the elements"""
        return self.vals[:]

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}' #-1 omits trailing comma
```

- Expression `IntSet.insert` refers to the method `insert` defined in the definition of the class `IntSet`.
- `print(type(IntSet), type(IntSet.insert))`

  → `<class 'type'> <class 'function'>`

Class – Example (cont.)

```
def insert(self, e):
    """Assumes e is an integer and inserts e into self"""
    if e not in self.vals:
        self.vals.append(e)
```

```
s = IntSet()
s.insert(3)
```

- `insert` method has two **formal parameters**, but it is called with only one **actual parameter**. ⇒ artifact of the dot notation (.)

- The object associated with the expression preceding the dot (`s`) is passed as the first parameter to the method (=`self`).

Class – Operations

`class` *`classname`*

- supports 2 operations:

  ① Instantiation: used to create instances of the class

  ex) `s = IntSet()`

  ② Attribute references: use **dot notation** (.) to access attributes within the class

  ex) `s.member(1)`

Class Instantiation and Attribute Reference

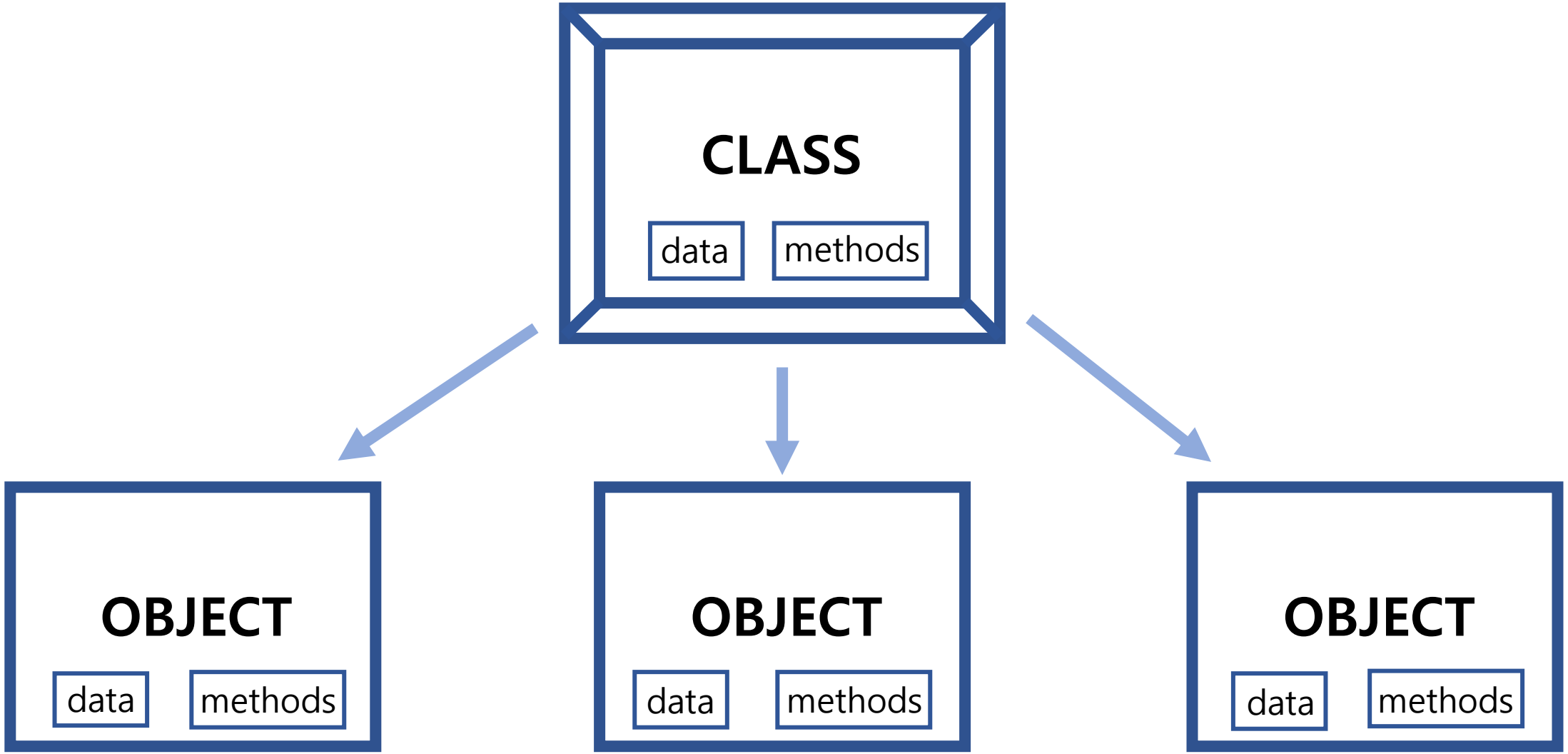
- Instantiation: *instancename* = *classname*()

- Method reference: *instancename*.method()

- Data reference: *instancename*.data

- Ex)
```
s = IntSet()
s.insert(3)
print(s.member(3))
```
→ True

Class Instantiation and Attribute Reference (cont.)



Class Instantiation and Attribute Reference – Example

```
import datetime

class Person(object):

    def __init__(self, name):
        """Create a person"""
        self.name = name
        try:
            lastBlank = name.rindex(' ')
            self.lastName = name[lastBlank+1:]
        except:
            self.lastName = name
        self.birthday = None

    def getName(self):
        """Returns self's full name"""
        return self.name

    def getLastName(self):
        """Returns self's last name"""
        return self.lastName

    def setBirthday(self, birthdate):
        """Assumes birthdate is of type datetime.date
           Sets self's birthday to birthdate"""
        self.birthday = birthdate
```

`Person` is instantiated with the argument `(name)`

→ supplied to the `__init__` method

- ✓ `name`
- ✓ `lastName`
- ✓ `birthday`

```
    def getAge(self):
        """Returns self's current age in days"""
        if self.birthday == None:
            raise ValueError
        return (datetime.date.today() - self.birthday).days

    def __lt__(self, other):
        """Returns True if self precedes other in alphabetical
           order, and False otherwise. Comparison is based on last
           names, but if these are the same full names are
           compared."""
        if self.lastName == other.lastName:
            return self.name < other.name
        return self.lastName < other.lastName

    def __str__(self):
        """Returns self's name"""
        return self.name
```

Class Instantiation and Attribute Reference – Example (cont.)

```
me = Person('Michael Guttag')
him = Person('Barack Hussein Obama')
her = Person('Madonna')

print(him.getLastName())
him.setBirthday(datetime.date(1961, 8, 4))
her.setBirthday(datetime.date(1958, 8, 16))
print(him.getName(), 'is', him.getAge(), 'days old')
```

```
Obama
Barack Hussein Obama is 20747 days old
```

- When instantiating a class, look at the specification of the `__init__` function for that class to know what arguments to supply and what properties those arguments should have.

Class Instantiation and Attribute Reference – Example (cont.)

- After the above code is executed, there will be three instances of class `Person`.

- One can then access information about these instances using the methods associated with them.
  - Ex) `him.getLastName()` returns `'Obama'`. `him.lastName` also returns `'Obama'`; However, writing expressions that directly access instance variables is considered poor form, and should be avoided.

Special Methods in Classes

① `__init__` : Called whenever a class is instantiated

② `__str__` : When the `print` command is used on the object, this method associated with the it is automatically invoked.

③ `__lt__` : Called when class instances are compared using "<" or ">"

Ex) `self.name < other.name  ==  self.name.__lt(other.name)`

Special Methods in Classes (cont.)

```
pList = [me, him, her]

for p in pList:
        print(p)
```

```
Michael Guttag
Barack Hussein Obama
Madonna
```

```
pList.sort()

for p in pList:
        print(p)
```

```
Michael Guttag
Madonna
Barack Hussein Obama
```

- `pList.sort()` will sort that list using the `__lt__` method defined in class `Person`.
  - Ex) "Guttag" < "Madonna" < "Obama"

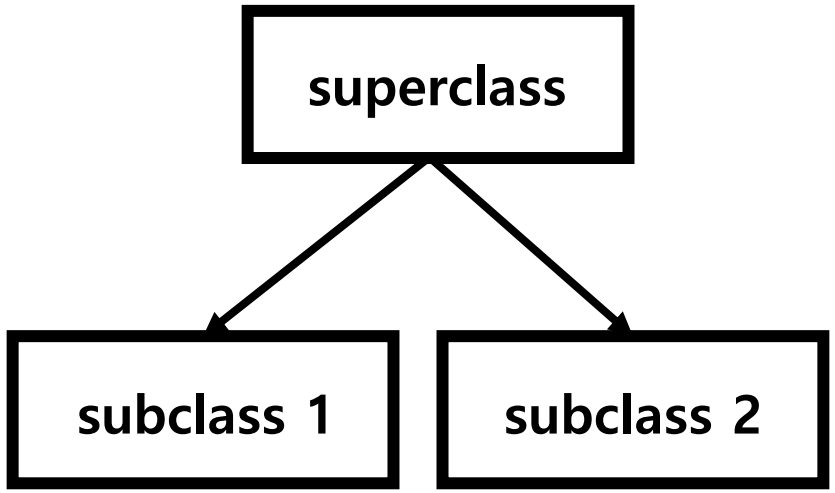
INHERITANCE

- Inheritance
- Variables
- Class Instantiation and Data Attributes Instantiation
- Multiple Levels of Inheritance

Inheritance

- Provides a convenient mechanism for building groups of related abstractions.
- Creates a type hierarchy.
- Each type inherits attributes from the types above it in the hierarchy.
- class **Object** is at the top of the hierarchy.

Inheritance (cont.)



- Subclass inherits all attributes of its superclass.
- Subclass can add new attributes.
- Subclass can **override**, i.e., replace attributes of its superclass.
- The version of the method that is executed is based on the object that is used to invoke the method.

Inheritance – Example

```
class MITPerson(Person):

    nextIdNum = 0 #identification number

    def __init__(self, name):
        Person.__init__(self, name)
        self.idNum = MITPerson.nextIdNum
        MITPerson.nextIdNum += 1

    def getIdNum(self):
        return self.idNum

    def __lt__(self, other):
        return self.idNum < other.idNum
```

- `MITPerson` is a subclass of `Person`, therefore inherits the attributes from its superclass, `Person`.
  - ✓ Add new attributes: class variable `nextIdNum`, instance variable `idNum`, method `getIdNum`.
  - ✓ Override: `__init__` , `__lt__`

Variables

- **Class variables**: data attributes associated with a class

  ex) `nextIdNum (class MITPerson)`

- **Instance variables** : data attributes associated with an instance

  ex) `idNum(`instance `MITPerson)`

Class Instantiation and Data Attributes Instantiation

```
class MITPerson(Person):

    nextIdNum = 0 #identification number

    def __init__(self, name):
        Person.__init__(self, name)
        self.idNum = MITPerson.nextIdNum
        MITPerson.nextIdNum += 1

    def getIdNum(self):
        return self.idNum

    def __lt__(self, other):
        return self.idNum < other.idNum
```

```
p1 = MITPerson("Choco Kim")
```

① `Person.__init__` → `self.name` 생성

② `MITPerson.__init__` → `self.idNum` 생성

(initialized by `nextIdNum`)

③ `nextIdNum`에 `1` 더함

- When an instance of `MITPerson` is created, a new instance of `nextIdNum` is NOT created. ⇒ It is incremented by 1.
- Each instance of `MITPerson` has a unique `idNum`.

Class Instantiation and Data Attributes Instantiation – Example

```
p1 = MITPerson('Mark Guttag’)
p2 = MITPerson('Billy Bob Beaver’)
p3 = MITPerson('Billy Bob Beaver’)
p4 = Person('Billy Bob Beaver')
```

p1 (MITPerson)

name = Mark Guttag
lastName = Guttag
birthday = None
idNum = 0

p2 (MITPerson)

name = Billy Bob Beaver
lastName = Beaver
birthday = None
idNum = 1

p3 (MITPerson)

name = Billy Bob Beaver
lastName = Beaver
birthday = None
idNum = 2

p4 (Person)

name = Billy Bob Beaver
lastName= Beaver
birthday = None

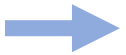
Class Instantiation and Data Attributes Instantiation – Example (cont.)

```
print(p3.name)
print(p3.idNum)
print(p4.name)
print(p4.idNum)
```

```
Billy Bob Beaver
2
Billy Bob Beaver
AttributeError: ‘Person’ object has no attribute ‘idNum’
```

Class Instantiation and Data Attributes Instantiation – Example (cont.)

```
print('p1 < p2 =', p1 < p2)
print('p3 < p2 =', p3 < p2)
print('p4 < p1 =', p4 < p1)
```



```
p1 < p2 = True
p3 < p2 = False
p4 < p1 = True
```

- `p1 = MITPerson, p2 = MITPerson`

  ⇒ uses the `__lt__` method defined in `class MITPerson`

- `p2 = MITPerson, p3 = MITPerson`

  ⇒ uses the `__lt__` method defined in `class MITPerson`

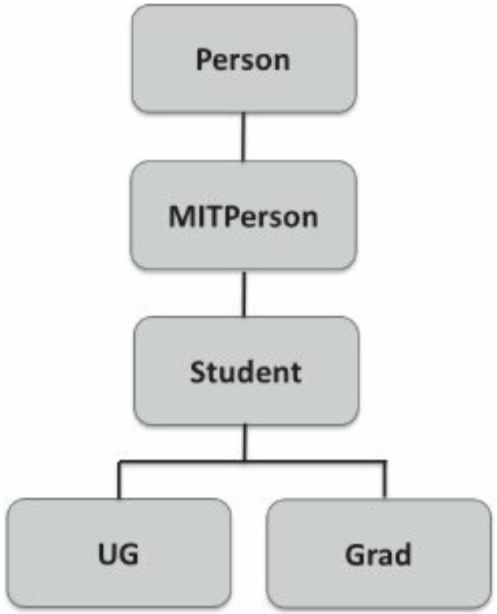
- `p1 = MITPerson, p4 = Person`

  ⇒ first argument of the expression is used to determine which `__lt__` method to invoke : `p4 < p1 == p4.__lt__(p1)`

  ⇒ uses the `__lt__` method defined in `class Person`

Multiple Levels of Inheritance

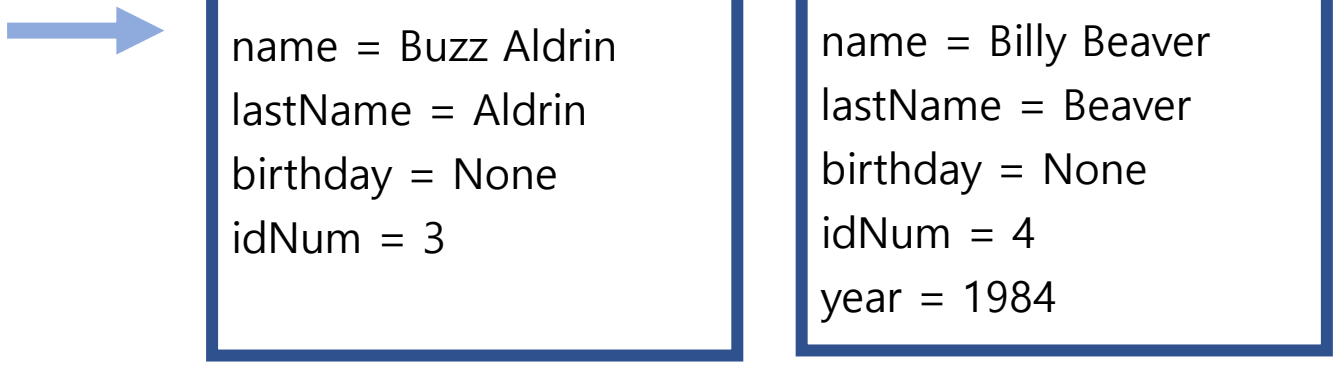
```
class Student(MITPerson):
    pass

class UG(Student):
    def __init__(self, name, classYear):
        MITPerson.__init__(self, name)
        self.year = classYear
    def getClass(self):
        return self.year

class Grad(Student):
    pass
```



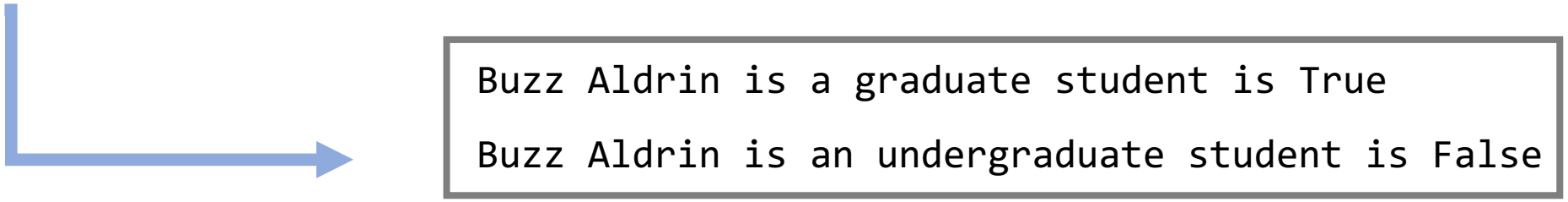
- class `Student` has no attributes other than those inherited from its superclass.
- By making the class `Student`
  ⇒ can distinguish students (`Student`) and non-students. (`MITPerson`)
- By making the class `Grad` ⇒ can distinguish two different kinds of students. (`UG` and `Grad`)

Multiple Levels of Inheritance - Example

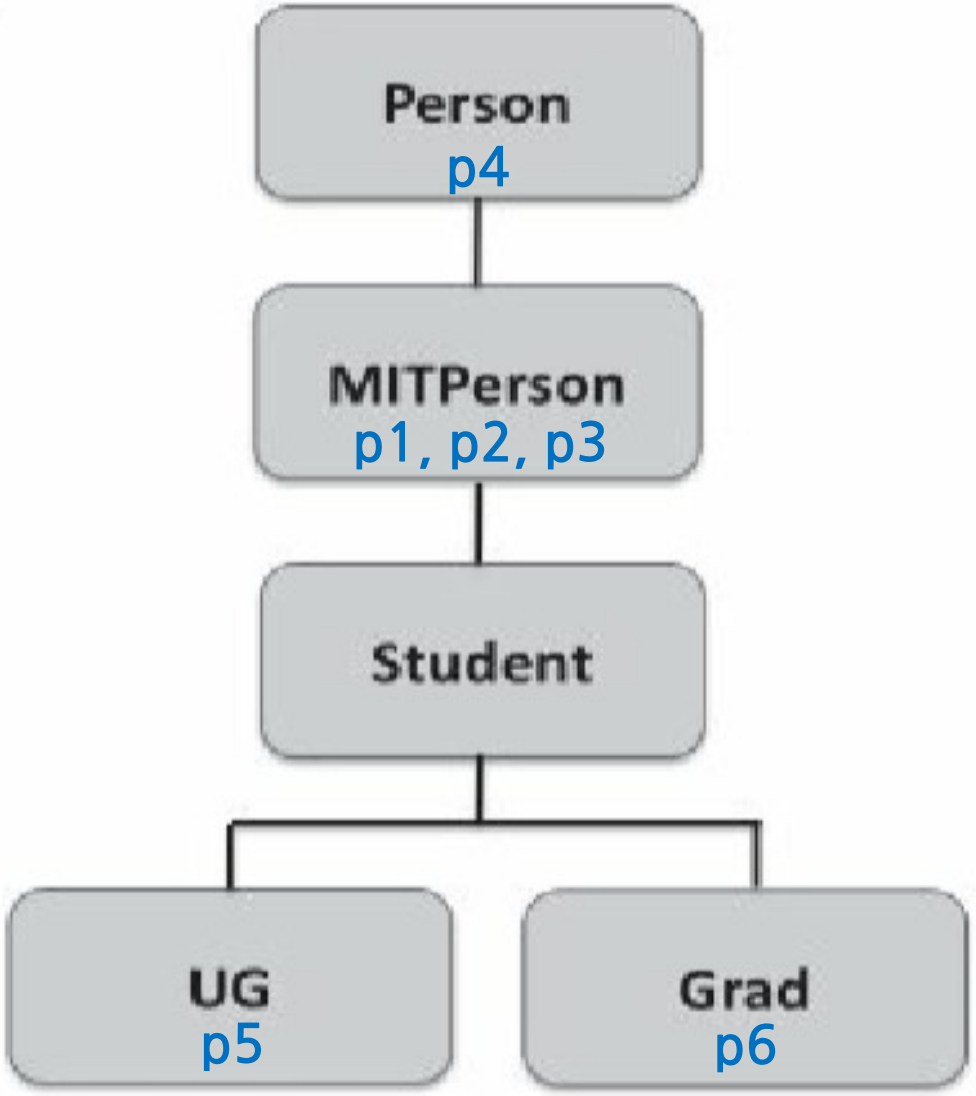
```
p5 = Grad('Buzz Aldrin')
p6 = UG('Billy Beaver', 1984)
```




```
print(p5, 'is a graduate student is', type(p5) == Grad)
print(p5, 'is an undergraduate student is', type(p5) == UG)
```



Multiple Levels of Inheritance – Example (cont.)



Multiple Levels of Inheritance – Example (cont.)

```
print(p5, 'is a student is', p5.isStudent())
print(p6, 'is a student is', p6.isStudent())
print(p3, 'is a student is', p3.isStudent())
```

```
Buzz Aldrin is a student is True
Billy Beaver is a student is True
Billy Bob Beaver is a student is False
```

ENCAPSULATION

- Encapsulation
- Information Hiding
- Accessing Data Attributes

Encapsulation

- Two important concepts at the heart of object-oriented programming.

1. **Encapsulation**
   - Bundling together of data attributes and the methods for operating on them.
2. **Information hiding**
   - One of the keys to modularity.
   - If those parts of the program that use a class rely only on the specifications of the methods in the class, a programmer implementing the class is free to change the implementation of the class (e.g., to improve efficiency) without worrying that the change will break code that uses the class.

Encapsulation – Example 1

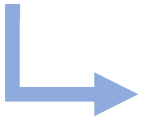
```
class Grades(object):

    def __init__(self):
        """Create empty grade book"""
        self.students = []
        self.grades = {}
        self.isSorted = True

    def addStudent(self, student):
        """Assumes: student is of type Student
           Add student to the grade book"""
        if student in self.students:
            raise ValueError('Duplicate student')
        self.students.append(student)
        self.grades[student.getIdNum()] = []
        self.isSorted = False
```

```
    def addGrade(self, student, grade):
        """Assumes: grade is a float
           Add grade to the list of grades for student"""
        try:
            self.grades[student.getIdNum()].append(grade)
        except:
            raise ValueError('Student not in mapping')

    def getGrades(self, student):
        """Return a list of grades for student"""
        try: #return copy of list of student's grades
            return self.grades[student.getIdNum()][:]
        except:
            raise ValueError('Student not in mapping')

    def getStudents(self):
        """Return a sorted list of the students in the grade book"""
        if not self.isSorted:
            self.students.sort()
            self.isSorted = True
        return self.students[:] #return copy of list of students
```

- `getGrades()` returns a copy of the list of grades associated with a student.
- `getStudents()` returns a copy of the list of students.

Encapsulation – Example 1 (cont.)

```
allStudents = course1.getStudents()
allStudents.extend(course2.getStudents())
```



```
allStudents.students = [course1 students, course2 students]
```

- i.e. By returning the **copy** of the list, the original list of students **cannot be altered from the outside!**

Encapsulation – Example 2

```
def gradeReport(course):
    """Assumes course is of type Grades"""
    report = ''
    for s in course.getStudents():
        tot = 0.0
        numGrades = 0
        for g in course.getGrades(s):
            tot += g
            numGrades += 1
        try:
            average = tot/numGrades
            report = report + '\n'\
                     + str(s) + '\'s mean grade is ' + str(average)
        except ZeroDivisionError:
            report = report + '\n'\
                     + str(s) + ' has no grades'
    return report
```

- Uses class `Grades` to produce a grade report for students taking a course.
- `gradeReport` gives the average of all grades of the student.

Encapsulation – Example 2 (cont.)

- Class Instantiation

```
ug1 = UG('Jane Doe', 2014)
ug2 = UG('John Doe', 2015)
ug3 = UG('David Henry', 2003)
g1 = Grad('Billy Buckner')
g2 = Grad('Bucky F. Dent')
```

ug1 (UG)

name = Jane Doe
lastName = Doe
birthday = None
idNum = 5
year = 2014

ug2 (UG)

name = John Doe
lastName = Doe
birthday = None
idNum = 6
year = 2015

ug3 (UG)

name = David Henry
lastName = Henry
birthday = None
idNum = 7
year = 2003

g1 (Grad)

name = Billy Buckner
lastName = Bukner
birthday = None
idNum = 8

g2 (Grad)

name = Bucky F.Dent
lastName = Dent
birthday = None
idNum = 9

Encapsulation – Example 2 (cont.)

```
sixHundred = Grades()
sixHundred.addStudent(ug1)
sixHundred.addStudent(ug2)
sixHundred.addStudent(g1)
sixHundred.addStudent(g2)
for s in sixHundred.getStudents():
    sixHundred.addGrade(s, 75)
sixHundred.addGrade(g1, 25)
sixHundred.addGrade(g2, 100)
sixHundred.addStudent(ug3)
print(gradeReport(sixHundred))
```

```
students =
  [ug1 (Jane Doe), ug2 (John Doe),
   g1 (Billy Buckner), g2 (Bucky F. Dent),
   ug3 (David Henry)]

grades = {5: [75], 6: [75], 8: [75, 25],
          9: [75, 100], 7: []}
```

```
Jane Doe's mean grade is 75.0
John Doe's mean grade is 75.0
David Henry has no grades
Billy Buckner's mean grade is 50.0
Bucky F. Dent's mean grade is 87.5
```

- function `addStudent()` uses class `Grades` to produce a grade preport for some students taking a course named sixHundred.

Information Hiding

- Some programming languages (e.g., Java, C++) provide mechanisms for enforcing information hiding.

- Programmers can make the attributes of a class **private**, so that clients of the class can access the data only through the object's methods.

- Python 3 uses a naming convention to make attributes invisible outside the class.
  - When the name of an attribute starts with ___ but does not end with ___, that attribute is not visible outside the class.

Information Hiding – Example

```
class infoHiding(object):
    def __init__(self):
        self.visible = 'Look at me'
        self.__alsoVisible__ = 'Look at me too'
        self.__invisible = 'Don\'t look at me directly'

    def printVisible(self):
        print(self.visible)

    def printInvisible(self):
        print(self.__invisible)

    def __printInvisible(self):
        print(self.__invisible)

    def __printInvisible__(self):
        print(self.__invisible)
```

```
test = infoHiding()
print(test.visible)
print(test.__alsoVisible__)
print(test.__invisible)
```

```
Look at me
Look at me too
Error: 'infoHiding' object has no attribute '__invisible'
```

Information Hiding – Example (cont.)

```
class infoHiding(object):
    def __init__(self):
        self.visible = 'Look at me'
        self.__alsoVisible__ = 'Look at me too'
        self.__invisible = 'Don\'t look at me directly'

    def printVisible(self):
        print(self.visible)

    def printInvisible(self):
        print(self.__invisible)

    def __printInvisible(self):
        print(self.__invisible)

    def __printInvisible__(self):
        print(self.__invisible)
```

```
test = infoHiding()
test.printInvisible()
test.__printInvisible__()
test.__printInvisible()
```

```
Don't look at me directly
Don't look at me directly
Error: 'infoHiding' object has no attribute '__printInvisible'
```

Accessing Data Attributes

- Accessing (private) data attributes of superclass in subclass;

```
class subClass(infoHiding):
    def __init__(self):
        print('from subclass', self.__invisible)
```

```
testSub = subClass()
```

```
Error: 'subClass' object has no attribute '_subClass__invisible'
```

- When a subclass attempts to use a hidden attribute of its superclass, an `AttributeError` occurs.