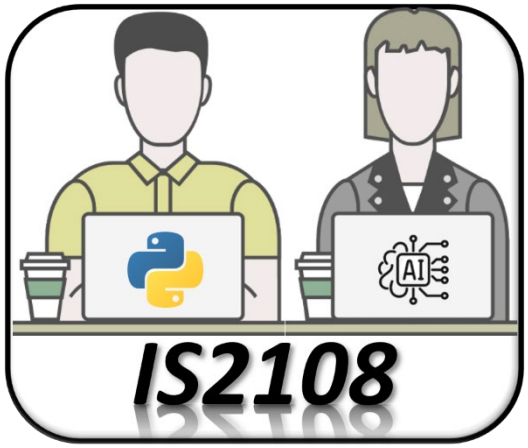


Lecture 6

Web Application Development with Django (II)

IS2108 – Full-stack Software Engineering for AI Solutions I
AY 2025/26 Semester 1

Lecturer: A/P TAN Wee Kek

Email: distwk@nus.edu.sg :: **Tel**: 6516 6731 :: **Office**: COM3-02-35

Consultation: Tuesday, 12 pm to 2 pm. Additional consultations by appointment are welcome.

Learning Objectives

- At the end of this lecture, you should understand:
  - Overview of object-oriented programming with Python.
  - Overview of object relational mapping (ORM).
  - ORM in Django.

Readings

- Required readings:
  - None.
- Suggested readings:
  - None

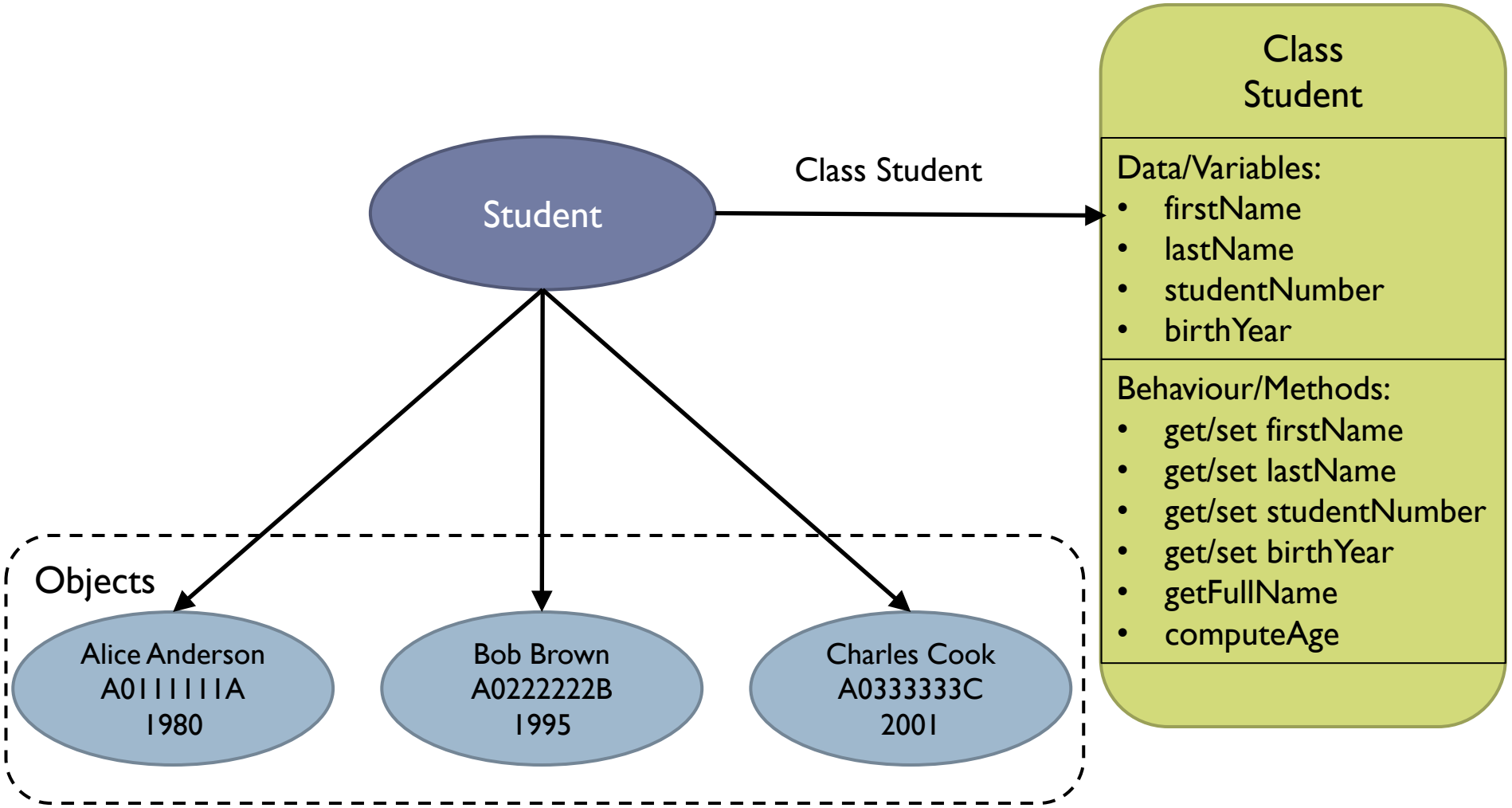
Object-Oriented Programming with Python

src

Object-oriented Programming

- **Objected-oriented programming (OOP)** is a programming paradigm based on the notion of **objects**:
  - Objects can contain data in the form of attributes or properties.
  - Objects can contain behaviour or code in the form of methods.
  - Similar objects are based on a common template known as **class**.
- OOP is useful for representing real-world objects.
- OOP is commonly used to build large-scale and complex software application involving objects interacting with one another

Object-oriented Programming (cont.)



Object-oriented Programming (cont.)

- Most methods in a class are getter/setter methods for accessing and modifying an object's attributes.
- There can also be non-getter/setter methods.

Classes in Python

- A class is defined using the `class` reserved word and the header convention:

```
class name:
    statements
```

- The name of the `class` must conform to the same identifier rules as variables and functions.
- The block of statements that makes up the class's body is indented to the right:
  - `statements` define class variables that are shared by all instances of the class.
  - **Class variables** can be attributes and methods.

Classes in Python (cont.)

- **Instance variables**:
  - Unlike class variables, instance variables are meant for data that are unique to each instance or object.
- The first argument of a class method always refers to itself:
  - The first argument is typically named as `self`.
  - This is nothing more than a convention.
  - The name `self` has no special meaning in Python.
  - The first argument can be named with any valid identifier.
- Creating an **instance** or object of a class:
  - Class instantiation uses the function notation.

Classes in Python (cont.)

- The class object is treated as function without parameter that returns a new instance of the class.

```
1   class Student:
2
3       kind = 'Undergraduate'
4
5
6
7   def main():
8
9       alice = Student()
10      print(alice.kind)
11
12      bob = Student()
13      print(bob.kind)
14
15
16
17  if __name__ == '__main__':
18
19      main()
```

```
Console 1/A
Undergraduate
Undergraduate
```

src01.py

- In `src01.py`, `kind` is a class attribute.
- Observe that both alice and bob are objects of Student `class` have the same value "Undergraduate" for `kind`.

Classes in Python (cont.)

- **Constructor**:
  - The instantiation operation creates an empty object by default.
  - In most cases, we would prefer to create objects with instances customised to a specific initial state.
  - A class may define a special method named `__init__` to perform this task.
  - In most other programming languages, this method is known as the constructor.
- The **instance attributes** of an object are defined via the `self` argument of the `__init__` method.

Classes in Python (cont.)

```
1   class Student:
2
3       kind = 'Undergraduate'
4
5
6
7       def __init__(self, firstName, lastName, studentNumber, birthYear):
8
9           self.firstName = firstName
10          self.lastName = lastName
11          self.studentNumber = studentNumber
12          self.birthYear = birthYear
13
14
15
16  def main():
17
18      alice = Student('Alice', 'Anderson', 'A0111111A', 1980)
19      print(alice.kind, alice.firstName, alice.lastName, alice.studentNumber, alice.birthYear, sep=', ')
20
21      bob = Student('Bob', 'Brown', 'A0222222B', 1995)
22      print(bob.kind, bob.firstName, bob.lastName, bob.studentNumber, bob.birthYear, sep=', ')
23
24
25
26  if __name__ == '__main__':
27
28      main()
```

- In `src02.py`, `firstName`, `lastName`, `studentNumber` and `birthYear` are instance attributes.
- Observe that both alice and bob objects of Student `class` have different values for these instance attributes.

src02.py

```
Console 1/A ×
Undergraduate, Alice, Anderson, A0111111A, 1980
Undergraduate, Bob, Brown, A0222222B, 1995
```

Classes in Python (cont.)

- So, what is the tangible difference between class attributes and instance attributes?
  - All objects of the same class share the same values for class attributes initially after instantiation.
  - But the values of class attributes for individual objects can certainly be changed thereafter.

```python
class Student:

    kind = 'Undergraduate'


    def __init__(self, firstName, lastName, studentNumber, birthYear):

        self.firstName = firstName
        self.lastName = lastName
        self.studentNumber = studentNumber
        self.birthYear = birthYear


def main():

    alice = Student('Alice', 'Anderson', 'A0111111A', 1980)
    bob = Student('Bob', 'Brown', 'A0222222B', 1995)

    print('alice.kind = {}'.format(alice.kind))
    print('bob.kind = {}'.format(bob.kind))
    print()

    bob.kind = 'Postgraduate'

    print('alice.kind = {}'.format(alice.kind))
    print('bob.kind = {}'.format(bob.kind))


if __name__ == '__main__':

    main()
```

```
Console 1/A ×
alice.kind = Undergraduate
bob.kind = Undergraduate

alice.kind = Undergraduate
bob.kind = Postgraduate
```

src03.py

More about Classes in Python

- Class methods:
  - Recall that class methods can be getter/setter methods or other behaviour methods.
- **Encapsulation** is an important OOP design pattern:
  - Data are visible only to semantically related methods to prevent misuse.
  - A class should not allow external calling code to access internal object data directly.
  - Internal object data should only be accessed through class methods.

More about Classes in Python (cont.)

- In most other programming languages:
  - A `class` enforces access restrictions to internal object data explicitly using the `private` reserved word.
  - Methods that are intended for use by code outside the `class` to access internal object data are designated with the `public` reserved word.
- In Python:
  - There is no `private` instance variables.
  - All instance variables can be accessed by external code.
  - However, a well-known convention is to prefix the name of instance variables with an underscore.
  - Class methods are then defined to access these underscored instance variables.

More about Classes in Python (cont.)

```python
import datetime


class Student:

    _kind = 'Undergraduate'


    def __init__(self, firstName, lastName, studentNumber, birthYear):

        self._firstName = firstName
        self._lastName = lastName
        self._studentNumber = studentNumber
        self._birthYear = birthYear

    def getkind(self):

        return self._kind

    def getFirstName(self):

        return self._firstName

    def setFirstName(self, firstName):

        self._firstName = firstName

    def getLastName(self):

        return self._lastName

    def setLastName(self, lastName):

        self._lastName = lastName

    def getStudentNumber(self):

        return self._studentNumber

    def setStudentNumber(self, studentNumber):

        self._studentNumber = studentNumber

    def getBirthYear(self):

        return self._birthYear

    def setBirthYear(self, birthYear):

        self._birthYear = birthYear

    def getFullName(self):

        return self._firstName + ' ' + self._lastName

    def computeAge(self):

        return datetime.datetime.now().year - self._birthYear


def main():

    alice = Student('Alice', 'Anderson', 'A0111111A', 1980)
    print(alice.getkind(), alice.getFirstName(), alice.getLastName(), alice.getStudentNumber(),
          alice.getBirthYear(), alice.getFullName(), alice.computeAge(), sep=', ')

    bob = Student('Bob', 'Brown', 'A0222222B', 1995)
    print(bob.getkind(), bob.getFirstName(), bob.getLastName(), bob.getStudentNumber(),
          bob.getBirthYear(), bob.getFullName(), bob.computeAge(), sep=', ')


if __name__ == '__main__':

    main()
```

- In `src04.py`, instance variables are accessed via their respective getter/setter methods.
- Observe that `getFullName()` and `computeAge()` are class methods.

```
Console 1/A ×
Undergraduate, Alice, Anderson, A0111111A, 1980, Alice Anderson, 42
Undergraduate, Bob, Brown, A0222222B, 1995, Bob Brown, 27
```

src04.py

Reusing Python Classes

- Recall that in Lecture P, we had introduced the notion of modules in Python:
  - Modules are source files containing Python code that can be imported into another source file.
  - We can define classes (and functions) in a module to more easily reuse them across different Python programs.
- See `src05.py` and `mylib.py` for a typical setup.

Reusing Python Classes (cont.)

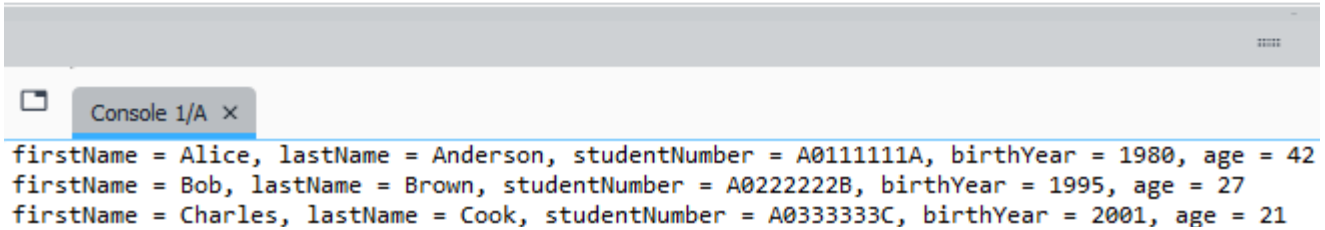
mylib.py

```
import datetime


class Student:

    kind = 'Undergraduate'


    def __init__(self, firstName, lastName, studentNumber, birthYear):

        self.firstName = firstName
        self.lastName = lastName
        self.studentNumber = studentNumber
        self.birthYear = birthYear

    def getFullName(self):

        return self.firstName + ' ' + self.lastName

    def computeAge(self):

        return datetime.datetime.now().year - self.birthYear


def print_student_data(students):

    for student in students:

        print('firstName = {}, lastName = {}, studentNumber = {}, birthYear = {}, age = {}'.
              format(student.firstName, student.lastName, student.studentNumber, student.birthYear, student.computeAge()))
```

Reusing Python Classes (cont.)

src05.py

```python
import mylib



def main():

    alice = mylib.Student('Alice', 'Anderson', 'A0111111A', 1980)
    bob = mylib.Student('Bob', 'Brown', 'A0222222B', 1995)
    charles = mylib.Student('Charles', 'Cook', 'A0333333C', 2001)

    students_list = list([])
    students_list.extend([alice, bob, charles])

    mylib.print_student_data(students_list)



if __name__ == '__main__':

    main()
```



Inheritance in Python

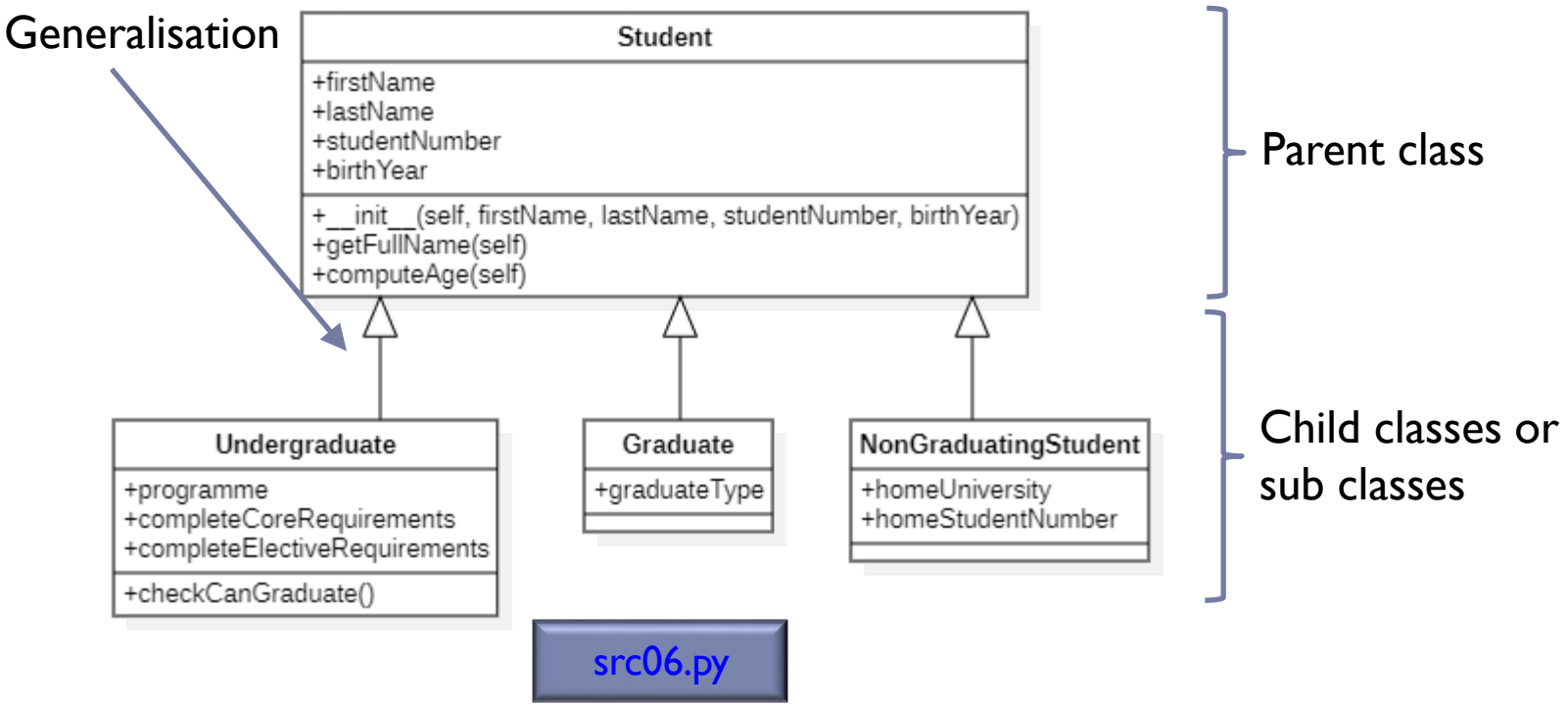
- A child class is similarly defined using the `class` reserved word and the header convention:
  - But the parent `class` name is enclosed in a pair of parenthesis brackets after the child `class` name.

```
class name(ParentClassName):
    statements
```

- The block of statements that makes up the child class's body defines the unique attributes and methods.
- It is possible to have no unique attribute and/or method:
  - Use the child class to denote a specialised kind of object.
  - Recall the class variable `kind` in the previous lecture.

Inheritance in Python (cont.)

- `super()` is used to reference the parent class.



Inheritance in Python (cont.)

```python
class Student:

    def __init__(self, firstName, lastName, studentNumber, birthYear):

        self.firstName = firstName
        self.lastName = lastName
        self.studentNumber = studentNumber
        self.birthYear = birthYear

    def getFullName(self):

        return self.firstName + ' ' + self.lastName

    def computeAge(self):

        return datetime.datetime.now().year - self.birthYear


class Undergraduate(Student):

    def __init__(self, firstName, lastName, studentNumber, birthYear,
                 programme, completeCoreRequirements, completeElectiveRequirements):

        super().__init__(firstName, lastName, studentNumber, birthYear)

        self.programme = programme
        self.completeCoreRequirements = completeCoreRequirements
        self.completeElectiveRequirements = completeElectiveRequirements

    def checkCanGraduate(self):

        return self.completeCoreRequirements and self.completeElectiveRequirements


class Graduate(Student):

    def __init__(self, firstName, lastName, studentNumber, birthYear,
                 graduateType):

        super().__init__(firstName, lastName, studentNumber, birthYear)

        self.graduateType = graduateType
```

```python
class NonGraduatingStudent(Student):

    def __init__(self, firstName, lastName, studentNumber, birthYear,
                 homeUniversity, homeStudentNumber):

        super().__init__(firstName, lastName, studentNumber, birthYear)

        self.homeUniversity = homeUniversity
        self.homeStudentNumber = homeStudentNumber




def main():

    sally = Student('Sally', 'Sandy', 'A0111111A', 1980)
    print(type(sally), sally.firstName, sally.lastName, sally.studentNumber, sally.birthYear, sep=', ')

    ursula = Undergraduate('Ursula', 'Underwood', 'A0222222B', 1995, 'Computing', False, False)
    print(type(ursula), ursula.firstName, ursula.lastName, ursula.studentNumber, ursula.birthYear,
          ursula.programme, ursula.completeCoreRequirements, ursula.completeElectiveRequirements, sep=', ')

    george = Graduate('George', 'Gordon', 'A0333333C', 1990, 'Master')
    print(type(george), george.firstName, george.lastName, george.studentNumber, george.birthYear,
          george.graduateType, sep=', ')

    nick = NonGraduatingStudent('Nick', 'Newton', 'A0444444D', 1991, 'Nanyang Technological University',
                                'E012345Z')
    print(type(nick), nick.firstName, nick.lastName, nick.studentNumber, nick.birthYear,
          nick.homeUniversity, nick.homeStudentNumber, sep=', ')



if __name__ == '__main__':

    main()
```

```
<class '__main__.Student'>, Sally, Sandy, A0111111A, 1980
<class '__main__.Undergraduate'>, Ursula, Underwood, A0222222B, 1995, Computing, False, False
<class '__main__.Graduate'>, George, Gordon, A0333333C, 1990, Master
<class '__main__.NonGraduatingStudent'>, Nick, Newton, A0444444D, 1991, Nanyang Technological University, E012345Z
```

src06.py

More about Inheritance in Python

- Notice that in the `Student` class hierarchy:
  - `Undergraduate`, `Graduate` and `NonGraduatingStudent` are different kinds of `Student`.
  - That is, the child classes are specialised kind of `Student`.
- It may sometime be useful to define **abstract** parent class:
  - In Python, an abstract class contains one or more abstract methods.
  - An abstract method is a method that is declared but contains no implementation.
  - It is possible to instantiate an object of an abstract base class in Python and no exception is raised.

More about Inheritance in Python (cont.)

- Python supports multiple inheritance:
  - A child class definition may contain multiple parent classes.

```
class name(ParentClass1, ParentClass2,
           ParentClass3, [...]):
    statement
```

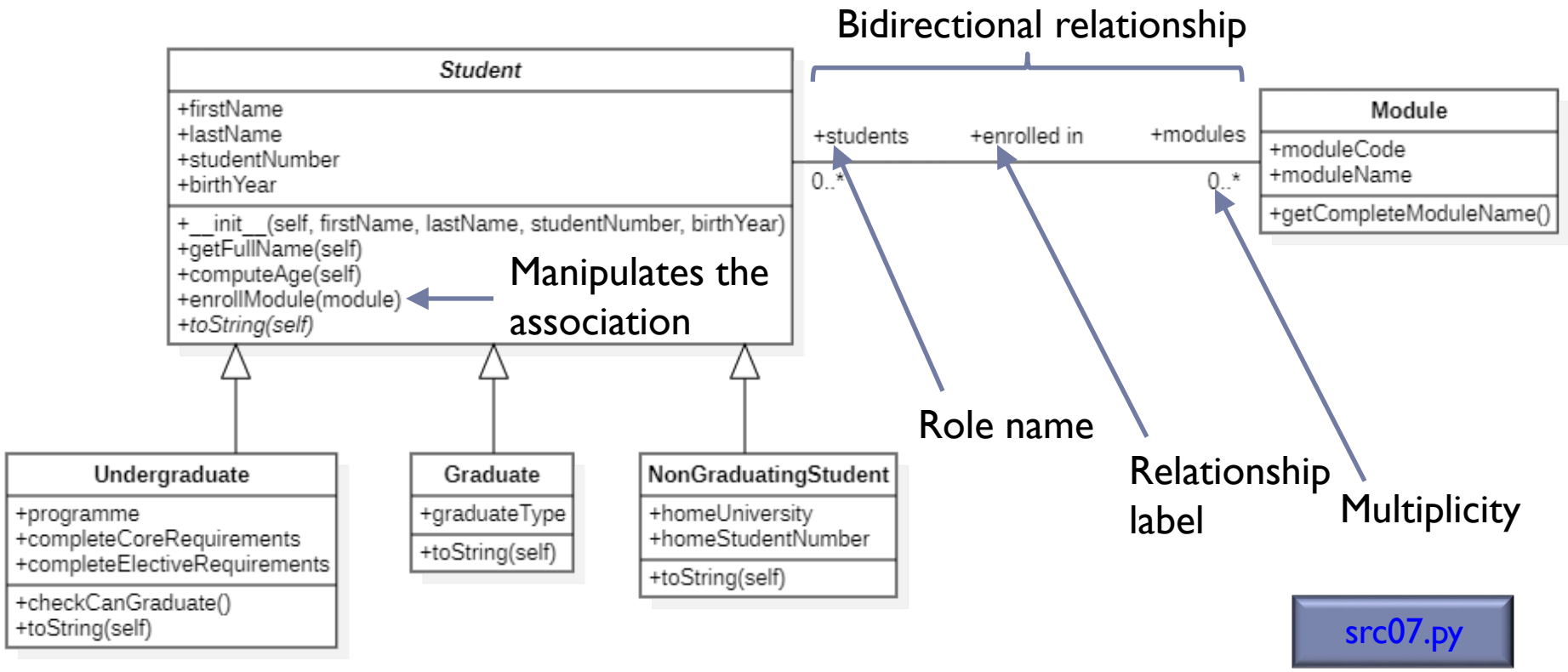
  - Common attributes and methods are search as depth-first, left-to-right.
  - In the real-word, such scenario is possible but rare and unnecessary.

Association in Python

- Recall that Python is dynamically typed:
  - Instance attributes representing an association are not typed to the corresponding `class`.
  - We simply instantiate them to an object of the corresponding `class`.
- If the multiplicity of an association variable is many:
  - E.g., 1..* or 0..*
  - We typically use a `list` data structure.
- There are different approaches to associate/dissociate relationships:
  - Manipulate the instance attributes directly.

Association in Python (cont.)

- Use getter/setter methods if defined with underscored attributes.
- Define dedicated class methods to manipulate the association.



Association in Python (cont.)

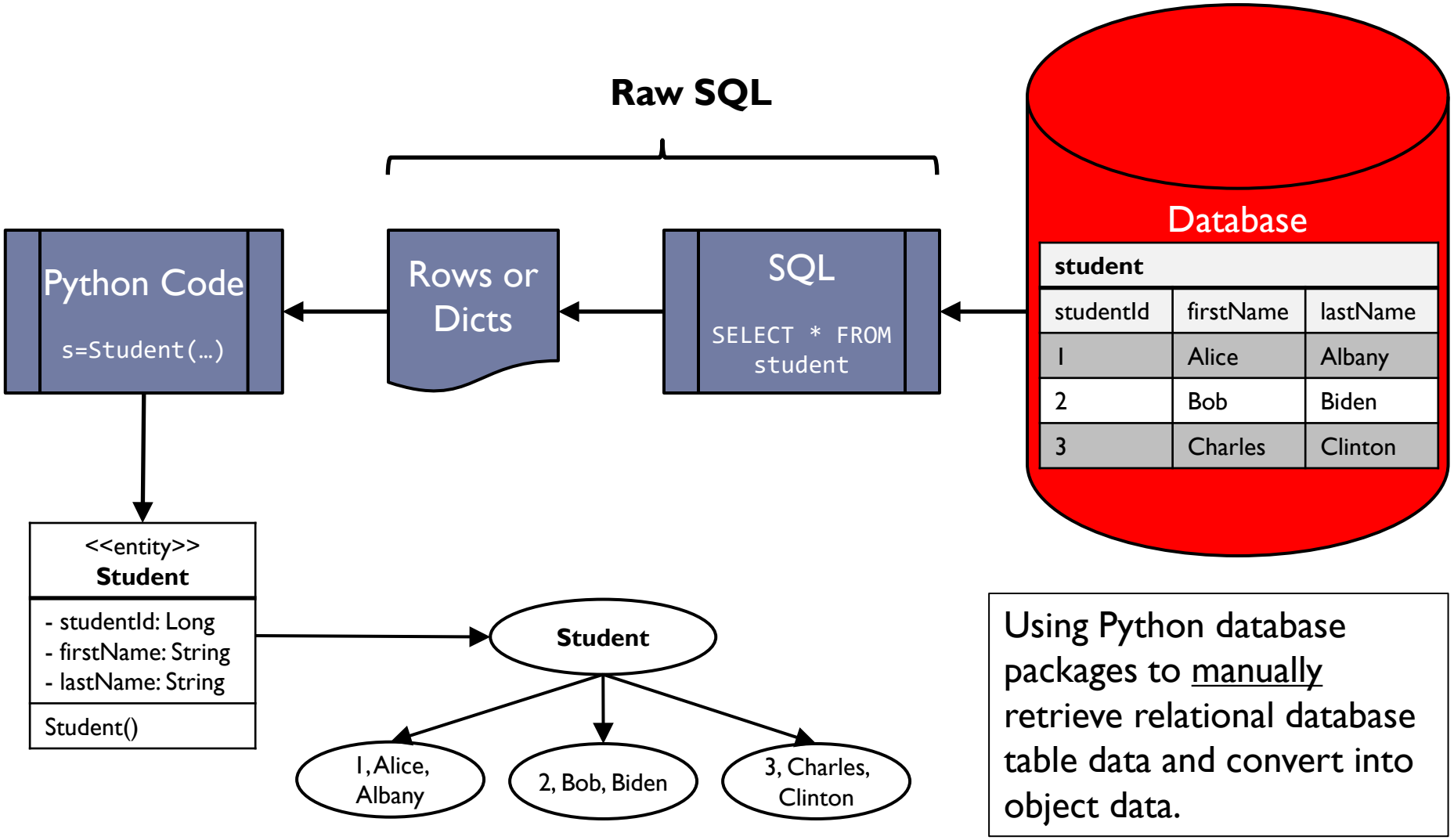
- `Student` enrolled in `Module`:
  - This is a bidirectional many-to-many relationship.
  - A student may enroll in many modules.
  - A module may be read by many students.
  - A new student may not enroll in any module and similarly a new module is not read by any student.
- The association is manipulated by `enrollModule()`:
- See sample code `src07.py`.

Object Relational Mapping (ORM)

Overview of Object Relational Mapping (ORM)

- Python can be used with object-oriented programming paradigm:
  - Works with data organised into objects.
  - Relational database works with data organised into tables.
- Traditionally, Python programmers use a suitable database package and raw SQL to:
  - Retrieve data from a relational database as rows or dicts then manually instantiate objects from the rows.
  - Write data into a database by using the objects to create the SQL statements manually and then executing them.
  - These mundane tasks are tedious, time-consuming and prone to errors.

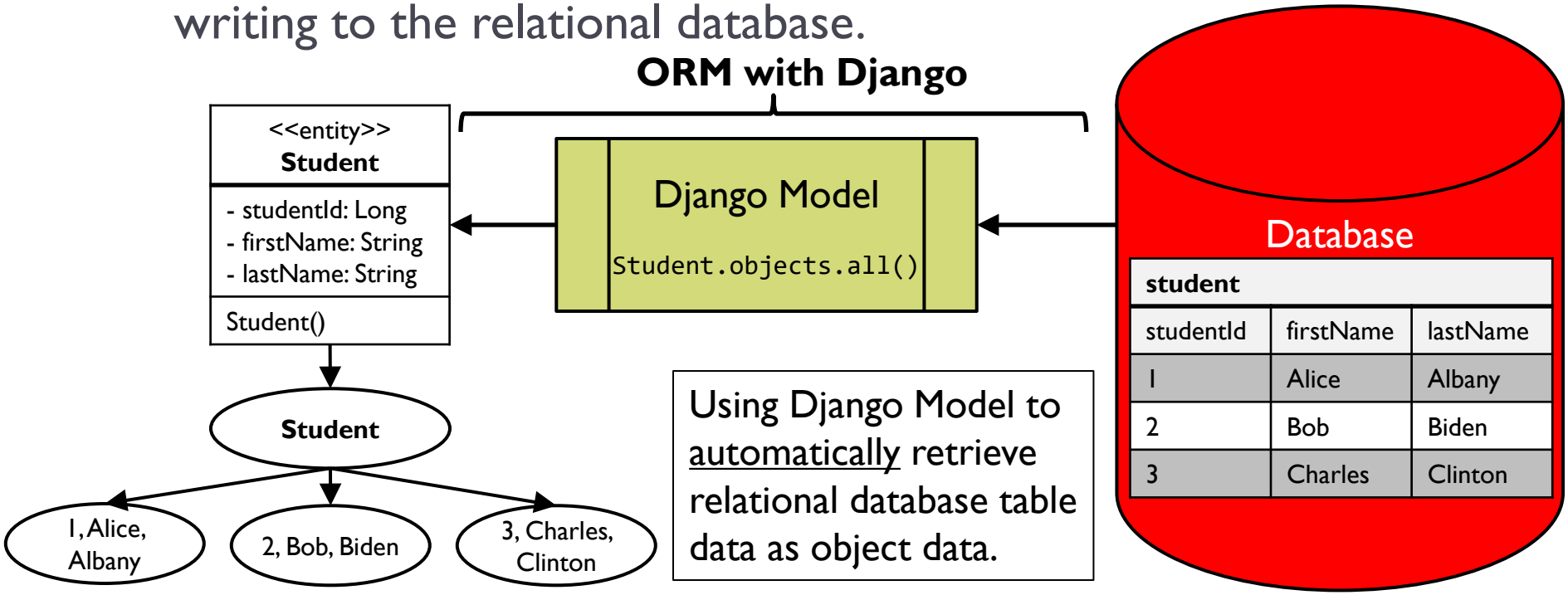
Overview of Object Relational Mapping (ORM) (cont.)



Using Python database packages to manually retrieve relational database table data and convert into object data.

Overview of Object Relational Mapping (ORM) (cont.)

- What is **object-relational mapping (ORM)**?
  - ORM is used for managing relational data in Python applications.
  - ORM automates the manual processes of retrieving from and writing to the relational database.



ORM in Django

lecture06

Models in Django

- **Models** in Django:
  - A model is the single, definitive source of information about data stored and managed by the application.
  - It contains the essential fields and behaviors of the data.
  - In general, each model maps to a single database table.
- Basic technical characteristics of a model:
  - Each model is a Python class that inherits from `django.db.models.Model`.
  - Each attribute of the model represents a database field.
  - Based on information in a model, Django provides database-access API that is automatically generated:
    - This will be discussed in the section on making queries.

Our First Django Model

- The following sample code defines a model class for `Student`:
  - `Student` is a subclass of `django.db.models.Model`.
  - Each attribute of the model is specified as a class variable:
    - Recall the difference between class variable and instance variable.
  - Each attribute is mapped to a column in the underlying database table.

```
7   class Student(models.Model):
8
9       firstName = models.CharField(max_length=32)
10      lastName = models.CharField(max_length=32)
11      studentNumber = models.CharField(max_length=9)
12      birthYear = models.IntegerField()
```

Our First Django Model (cont.)

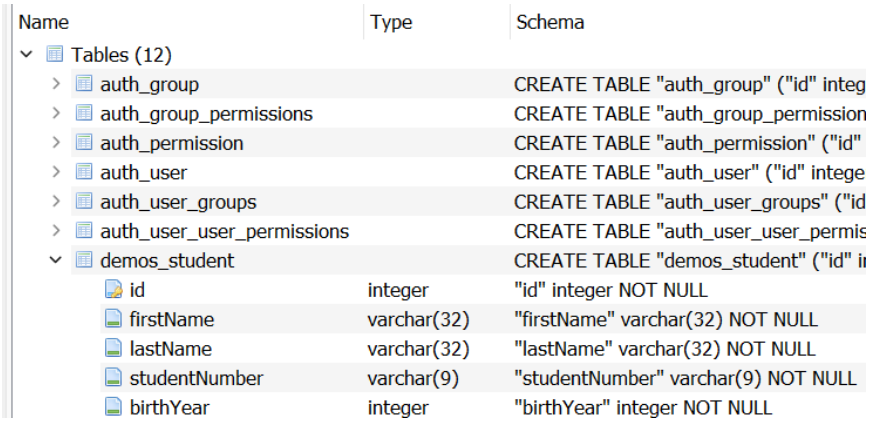
- To use a model:
  - Add the name of the app containing the required `models.py` into `settings.py` under `INSTALLED_APPS`.
    - This step is usually already done when creating a new app.
  - Run the following commands to create the database table:

```
python manage.py makemigrations
python manage.py migrate
```

  - The first command detects changes in `models.py` and generates the migration files (Python scripts) that describe those changes.
  - The second command applies the migration files to the actual database.

Our First Django Model (cont.)

- The `Student` model will create a database table resembling a typical `CREATE TABLE` statement in SQL.



Name	Type	Schema
Tables (12)		
auth_group		CREATE TABLE "auth_group" ("id" integ
auth_group_permissions		CREATE TABLE "auth_group_permission
auth_permission		CREATE TABLE "auth_permission" ("id"
auth_user		CREATE TABLE "auth_user" ("id" intege
auth_user_groups		CREATE TABLE "auth_user_groups" ("id
auth_user_user_permissions		CREATE TABLE "auth_user_user_permis
demos_student		CREATE TABLE "demos_student" ("id" i
id	integer	"id" integer NOT NULL
firstName	varchar(32)	"firstName" varchar(32) NOT NULL
lastName	varchar(32)	"lastName" varchar(32) NOT NULL
studentNumber	varchar(9)	"studentNumber" varchar(9) NOT NULL
birthYear	integer	"birthYear" integer NOT NULL

```
CREATE TABLE "demos_student" (
        "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
        "firstName" varchar(32) NOT NULL,
        "lastName" varchar(32) NOT NULL,
        "studentNumber" varchar(9) NOT NULL,
        "birthYear" integer NOT NULL
)
```

Our First Django Model (cont.)

- Additional points to note:
  - The name of the table, demos_student, is automatically derived from the app name and model name but can be overridden.
  - An `id` field is added automatically, but this behavior can be overridden:
    - `id` is known as an automatic primary key field.
  - The `CREATE TABLE` statement in this example is formatted using SQLite:
    - Django uses SQL tailored to the database backend specified in your settings file.

More About Fields

- Fields or attributes are the most important part of a model:
  - Fields are also the only required part of a model.
  - Field names should not conflict with the `models` API such as `clean`, `save` or `delete`.
- **Field types**:
  - Each field in a model should be an instance of the appropriate `Field` class.
  - Field class types determine many important information.
  - Column type:
    - The type of data to store.
    - E.g., `INTEGER`, `VARCHAR` and `TEXT`

More About Fields (cont.)

- Default HTML input element:
  - Used when rendering a form field.
  - E.g., `<input type="text">`, `<select>`
- Minimal validation requirements:
  - Used in Django's admin and in automatically-generated forms.
- Django supports dozens of built-in field types:
  - Refer to the official documentation page here.
- Custom model fields can also be created:
  - Refer to the official documentation page here.

More About Fields (cont.)

- **Field options**:
  - `null`:
    - Determines whether empty values are stored as NULL in the database.
    - Default is `False`.
  - `blank`:
    - Determines whether the field is allowed to be blank.
    - Default is `False`.
  - `choices`:
    - A sequence of 2-value tuples, a mapping, an enumeration type, or a callable (that expects no arguments and returns any of the previous formats).
    - These are use as choices for the field.

More About Fields (cont.)

- If provided, the default form input element will be a select box instead of the standard text field, which limits choices to those given.

- `default`:
  - The default value for the field.
  - This can be a value or a callable object:
    - If callable it will be called every time a new object is created.
- `primary_key`:
  - If `True`, this field is the primary key for the model.
  - If `primary_key=True` is not specified for any fields in your model, Django will automatically add a field to hold the primary key.
- `unique`:
  - If `True`, this field must be unique throughout the table.

More About Fields (cont.)

- In the following example of the `BetterStudent` model:

```python
class BetterStudent(models.Model):

    GENDER = [
        ('M', 'Male'),
        ('F', 'Female'),
        ('O', 'Other')
    ]

    firstName = models.CharField(max_length=32)
    lastName = models.CharField(max_length=32)
    studentNumber = models.CharField(max_length=9)
    birthYear = models.IntegerField()
    nationality = models.CharField(max_length=32, default='Singaporean')
    address = models.CharField(max_length=128, blank=True, null=True)
    gender = models.CharField(max_length=1, choices=GENDER)
```

demos_betterstudent		CREATE TABLE "demos_betterstudent" (
id	integer	"id" integer NOT NULL
firstName	varchar(32)	"firstName" varchar(32) NOT NULL
lastName	varchar(32)	"lastName" varchar(32) NOT NULL
studentNumber	varchar(9)	"studentNumber" varchar(9) NOT NULL
birthYear	integer	"birthYear" integer NOT NULL
nationality	varchar(32)	"nationality" varchar(32) NOT NULL
address	varchar(128)	"address" varchar(128)
gender	varchar(1)	"gender" varchar(1) NOT NULL

```
CREATE TABLE "demos_betterstudent" (
        "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
        "firstName" varchar(32) NOT NULL,
        "lastName" varchar(32) NOT NULL,
        "studentNumber" varchar(9) NOT NULL,
        "birthYear" integer NOT NULL,
        "nationality" varchar(32) NOT NULL,
        "address" varchar(128) NULL,
        "gender" varchar(1) NOT NULL
)
```

More About Fields (cont.)

- Observe that the SQL `CREATE TABLE` statement is quite different from the code defining the Django model.
- Field options:
  - `nationality` – Specifies a default value
  - `address` – Allows blank with blank value stored as NULL.
  - `gender` – Specifies 1 character length with choices.

Database Access API

- Recall that Django provides a set of database-access API based on the models defined:
  - These API methods make it easy to perform create, retrieve, update and delete (CRUD) operations on database records.
  - There is no need to write SQL statements explicitly.
- **Creating objects or model instances**:
  - Instantiate a new object using keyword arguments to the model class.
  - Then call the `save()` method to save the object into the database as a new record or row in the underlying table.
  - See `demo01` for an example with the `Student` model.
  - This is equivalent to an SQL `INSERT INTO` statement.

Database Access API (cont.)

- **Retrieving model instances**:
  - In general, objects can be retrieved from a database by constructing a `QuerySet` via a `Manager` on the required model class.
  - A `QuerySet` represents a collection of objects from a database.
  - It can have zero, one or many filters.
  - Filters are used to narrow down the query results based on the given parameters.
  - In SQL terminologies, a `QuerySet` equates to a `SELECT` statement, and a filter is a limiting clause such as `WHERE` or `LIMIT`.

Database Access API (cont.)

- A `QuerySet` can be obtained using the model's `Manager`.
- Each model has at least one `Manager`, and it's called `objects` by default.
- See `demo02` for an example with the `Student` model:
  - Observe the use of the `filter()` method in `Manager`.
  - In this example, we retrieve a single `Student` object based on `studentNumber`.
- **Retrieving all model instances**:
  - Use the `all()` method on a `Manager`.
  - See `demo03` for an example.

Database Access API (cont.)

- **Retrieving a single model instance**:
  - The `get()` method on a `Manager` can be used if it is known that there is only one object that matches a query.
  - See `demo04` for an example.
  - Observe the exception handling with `DoesNotExist` and `MultipleObjectsReturned`.
- **The `pk` lookup shortcut**:
  - Django provides the `pk` lookup shortcut where `pk` stands for primary key.
  - In the `Student` model example, the primary is the `id` field.
  - See `demo05` for the use of `pk` lookup.

Database Access API (cont.)

- **Saving changes to objects**:
  - The `save()` method can be used to save changes to an object that is already in the database.
  - Updating of objects is typically done after retrieval.
  - This is equivalent to an SQL `UPDATE` statement.
  - See `demo06` for an example.
- **Deleting objects**:
  - The `delete()` method immediately deletes an object.
  - It also returns the number of objects deleted and a dictionary with the number of deletions per object type.
  - This is equivalent to an SQL `DELETE FROM` statement.
  - See `demo07` for an example.

Relationships

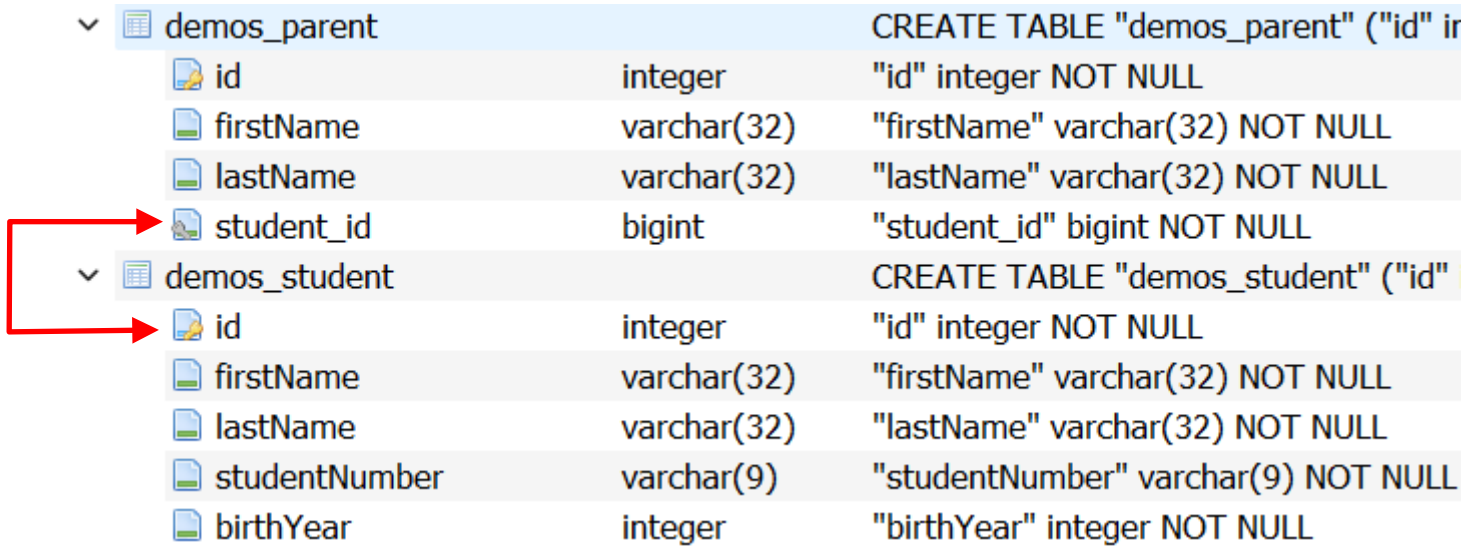
- One of the most powerful features of relational database is the relationships among tables:
  - Enforces referential constraints to preserve the integrity of data across related tables when performing create, update and delete operations.
  - Enables the querying of data across related tables.
- Django provides support to define the three most common types of database relationships:
  - many-to-one
    - How about one-to-many?
  - many-to-many
  - one-to-one

Relationships (cont.)

- **Many-to-one relationships**:
  - `django.db.models.ForeignKey` is used to define a many-to-one relationship.
  - It is used just like any other `Field` type by including it as a class attribute of the model.
  - `ForeignKey` requires a positional argument, which is the class to which the model is related.
  - Recall that in a many-to-one relationship, the foreign key is placed in the many side.
  - See `demo` for an example of one student has many parents.
    - This will create a foreign key `student_id` in the `demos_parent` table.

Relationships (cont.)

```python
class Parent(models.Model):

    firstName = models.CharField(max_length=32)
    lastName = models.CharField(max_length=32)

    student = models.ForeignKey(Student, on_delete=models.RESTRICT, related_name='parents', default=None)
```



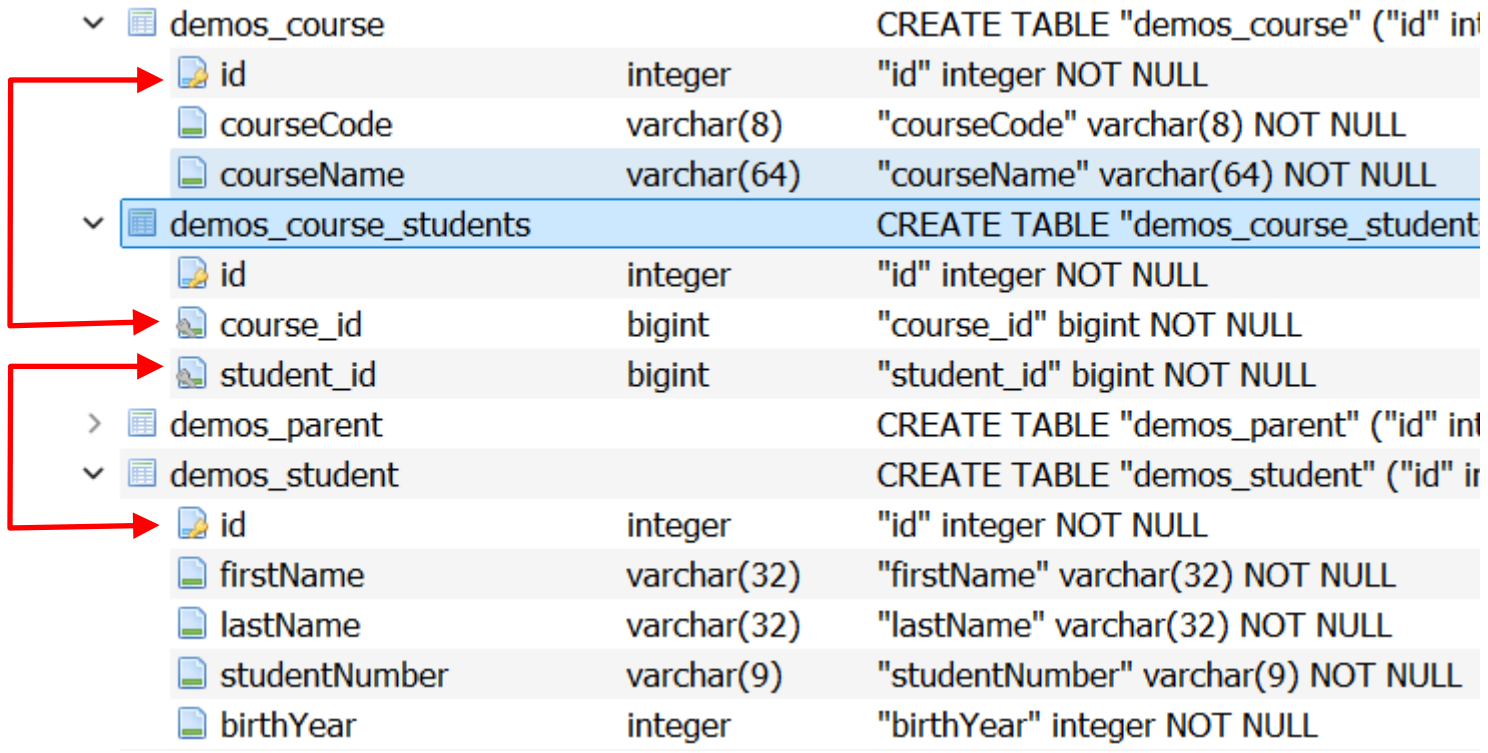

```
CREATE TABLE "demos_parent" (
        "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
        "firstName" varchar(32) NOT NULL,
        "lastName" varchar(32) NOT NULL,
        "student_id" bigint NOT NULL REFERENCES "demos_student" ("id")
)
```

Relationships (cont.)

- **Many-to-many relationships**:
  - `ManyToManyField` is used to define a many-to-many relationship.
  - Similar to `ForeignKey`, it is used just like any other `Field` type by including it as a class attribute of the model.
  - `ManyToManyField` requires a positional argument, which is the class to which the model is related.
  - See `demo` for an example of many students have many courses.
    - This will create a join table `demos_course_students`.
    - Recall that a many-to-many relationship requires a join table.
    - The foreign keys are inside the join table

Relationships (cont.)

```python
class Course(models.Model):

    courseCode = models.CharField(max_length=8)
    courseName = models.CharField(max_length=64)

    students = models.ManyToManyField(Student, related_name='courses_students')
```

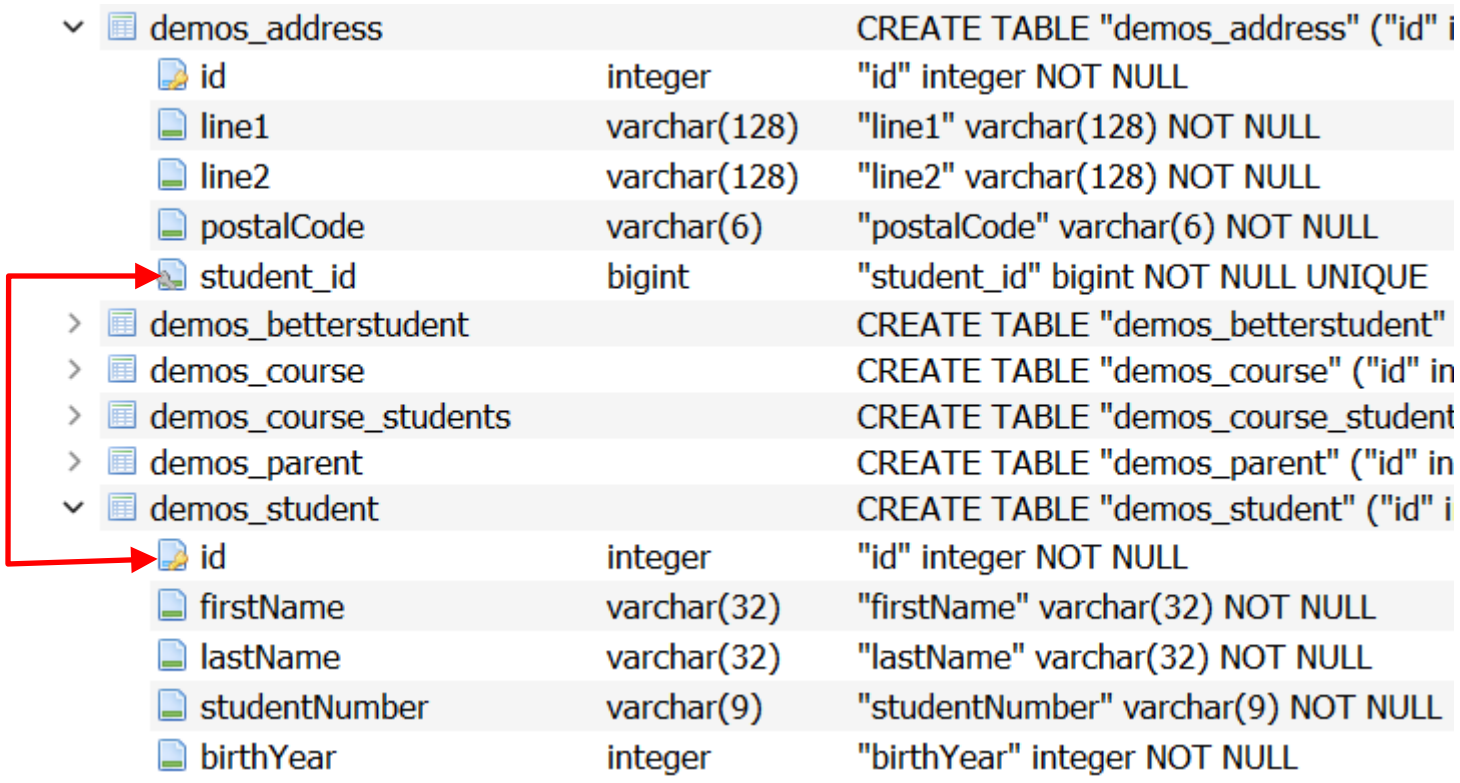


Relationships (cont.)

- **One-to-one relationships**:
  - `OneToOneField` is used to define a one-to-one relationship.
  - It is used just like any other `Field` type by including it as a class attribute of the model.
  - `OneToOneField` requires a positional argument, which is the class to which the model is related.
  - See `demo` for an example of one student has one address:
    - This will create a foreign key `student_id` in the `demos_address` table.

Relationships (cont.)

```
52  class Address(models.Model):
53
54      line1 = models.CharField(max_length=128)
55      line2 = models.CharField(max_length=128)
56      postalCode = models.CharField(max_length=6)
57
58      student = models.OneToOneField(Student, on_delete=models.CASCADE, related_name='address')
```



Associating and Disassociating Objects

- Once the relationships among objects have been defined, objects can be associated and dissociated through the model instances.
- The exact association and disassociation depends on the relationship type.
- **Many-to-one** and **one-to-one** relationships**:**
  - Association – Assign a model instance to the relationship field.
  - Disassociation – Assign `None` to the relationship field.
  - Use the `save()` method to update the database.
- **Many-to-many** relationships:
  - Association – Use the `add()` method of the relationship field.

Associating and Disassociating Objects (cont.)

- Disassociation – Use the `remove()` method of the relationship field.
- To remove all associations – Use the `clear()` method of the relationship field.
- To replace associations – Use the `set()` method of the relationship field.

- See `demo08` for the following association examples:
  - One student has many parents:
    - This is mandatory and is associated when the objects are created.
  - Many students have many courses.
  - One student has one address.
    - This is mandatory and is associated when the objects are created.

Associating and Disassociating Objects (cont.)

- See `demo09` for the following disassociation examples:
  - Many students have many courses.
  - The mandatory relationships are not dissociated but can be replaced if need to.

Retrieval of Relationship Fields

- In Django, relationship fields are retrieved lazily by default:
  - Django does not immediately fetch related objects when you retrieve a model instance.
  - The first query for the main object is executed.
  - The second query for the related objects is deferred until you actually access the relationship.
- There are two methods can be used for eager loading:
  - In this case, Django execute a single join query for both the main object and related objects together.
  - `select_related()` – Is used for single-valued relationship
  - `prefetch_related()` – Is used for multi-valued relationship

Retrieval of Relationship Fields (cont.)

- See `demo10` for retrieval of the following relationship:
  - Many students have many courses.
- In general, lazy loading is more efficient but requires explicit accessing of relationship fields.
- Eager fetching is useful when you are returning the main objects directly without explicit accessing of relationship fields.

Lookups that Span Relationships

- Django provides a powerful and intuitive way to "follow" relationships in lookups:
  - This mechanism automates SQL `JOIN` queries with filtering on related fields.
  - To span a relationship, use the field name of related fields across models, separated by double underscores.
  - Repeat the convention until you have reached the field that you need.
- See `demo11` for lookups spanning the following relationships:
  - One student has many parents.
  - Many students have many courses.

Model Inheritance

- Model inheritance in Django works almost identically to the way normal class inheritance works in Python:
  - However, the base class should still inherit from `django.db.models.Model`.
- We will defer the discussion of Django model inheritance to the lab.

Summary

- Objects contain data attributes and behavioural methods that manipulate them.
- Classes are templates that are defined to enable the instantiation of objects that share similar data attributes and behavioural methods.
- The methods in classes are defined as Python functions.

Q&A

Next Lecture…

- Learn about:
  - Advanced concepts on web application development with Django.

In-class Formative Quiz for Lecture 06