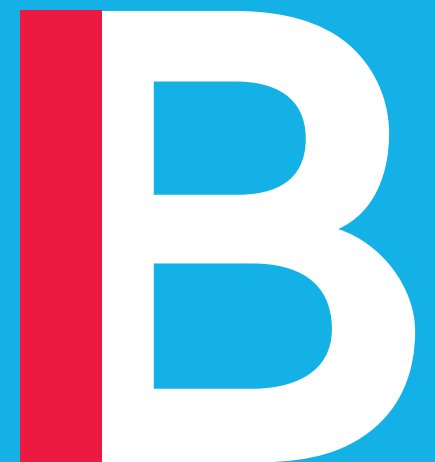


Lecture 9

Integrating SQL with other languages

Dr Fintan Nagle
f.nagle@imperial.ac.uk



Reading

Video lectures:

9.2 - Jupyter.mp4

9.3 - Pandas.mp4

9.4 - Graphing with Matplotlib.mp4

9.5.1 - Intro to ORMs in Python.mp4

9.5.2 - Flask.mp4

9.6 - ORMs in Ruby.mp4

History of pandas: <https://qz.com/1126615/the-story-of-the-most-important-tool-in-data-science/>

Reading

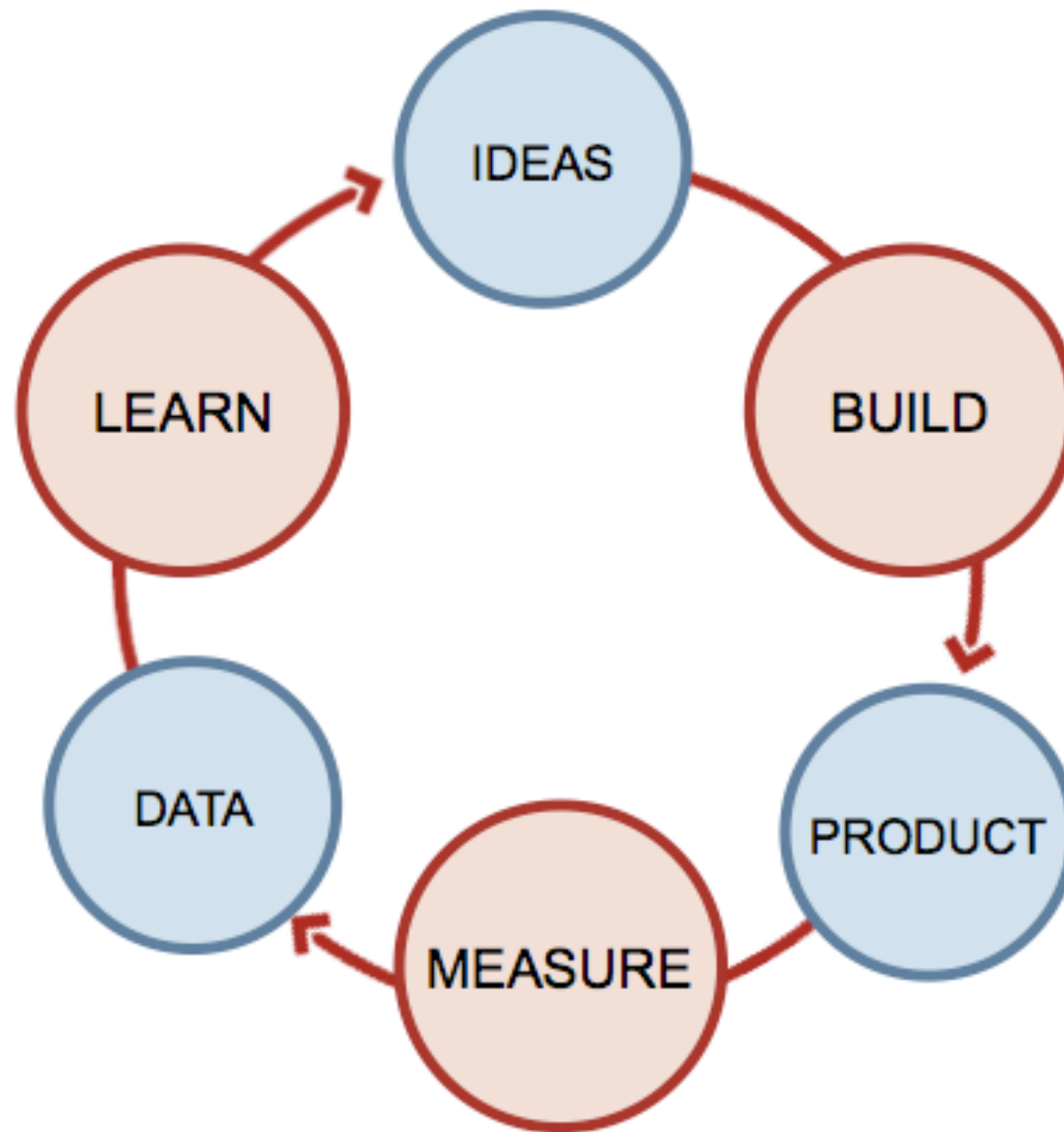
The XY problem

<https://xyproblem.info/>

The development loop

A person is sitting on a large blue beanbag chair in front of a modern glass building. The building has a grid-like pattern of windows and reflects the surrounding environment. The person is wearing a dark jacket and is looking down at something in their hands. The ground is paved with light-colored tiles. The overall scene is bathed in a blue light, giving it a modern and professional feel.

The development loop



The development loop

We often have several loops inside each other:

- Local testing (**development**)
Code in Sublime, run a server on your machine
Server reloads any changes near-instantly
Changes visible and testable in a few seconds
- Dry-run testing (**staging**)
Code in Sublime, do a git commit and deploy to the staging servers when ready
Changes visible and testable in a few minutes
- Real-world operation (**production**)
When ready, do code review, testing, security checks, then a deploy to the production servers
Changes visible and testable (by all users!) in a few hours

The development loop

Always keep the development loop as tight as possible.

Test everything in development first!

Text editors

A good text editor should:

- Not interfere with programmers' work (by adding extensions or refusing to save as different file types)
- Highlight syntax
- Show all files in the current project (not just folder), for easy browsing

Debugging

A person is sitting on a blue beanbag chair in front of a modern glass building. The building has a grid-like pattern of windows and reflects the surrounding environment. The person is wearing a dark jacket and is looking down at something in their hands. The ground is paved with light-colored tiles. The overall scene is bathed in a blue light, giving it a monochromatic appearance.



Debugging – classic pitfalls

- “It can’t be me, it must be a problem with the library.”
- “I’ll just file an angry bug report on Github...”
- “My environment must be OK.”

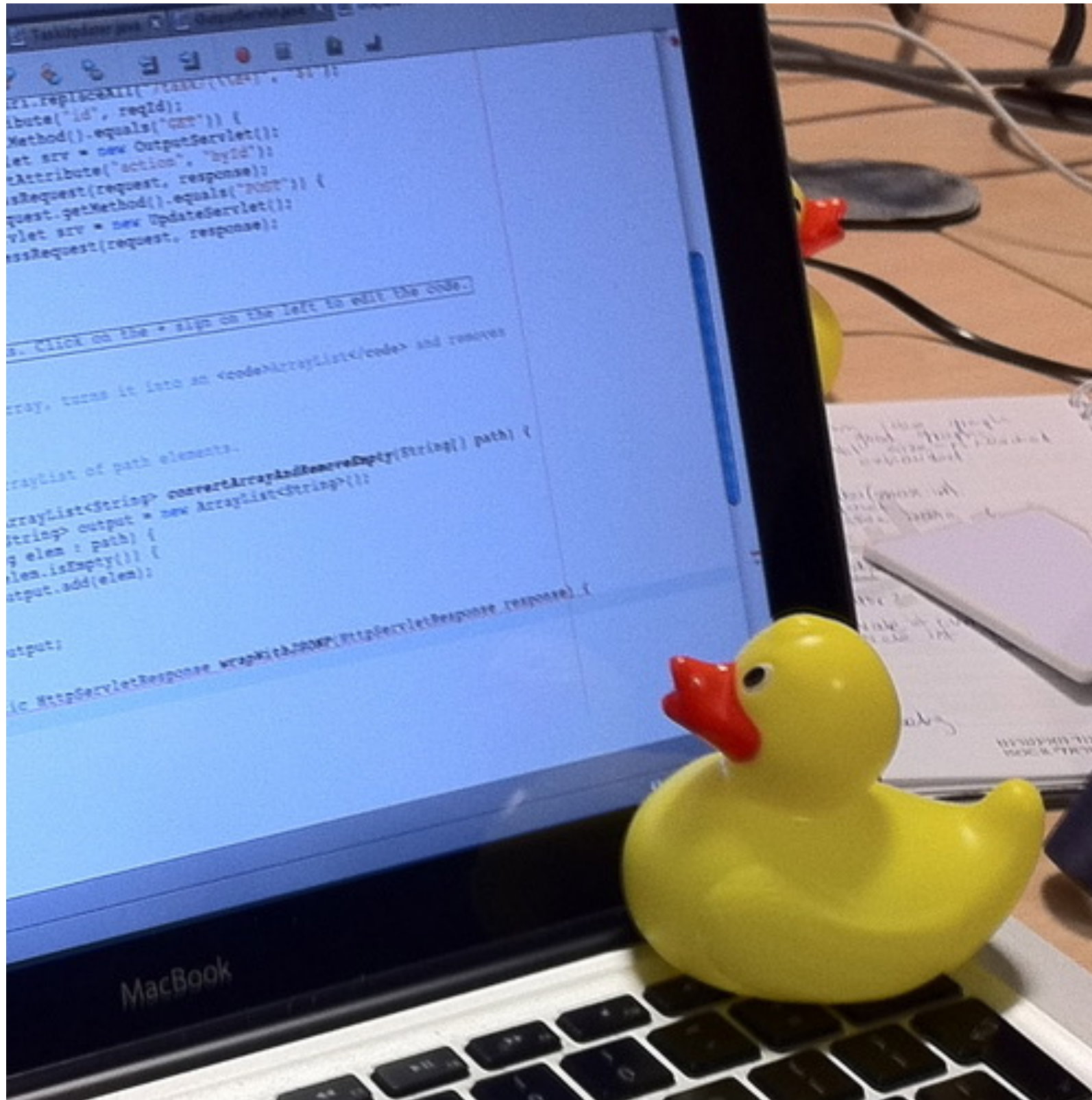
Difficulties in debugging are usually caused by false assumptions.

The bigger the assumption, the harder it is to question.

Always question your assumptions. If things aren’t working, you need to step back and question some larger assumptions.

This is why a good night’s sleep, a fresh mind and a second opinion are our best debugging tools.

Rubber duck debugging



The bridge method







Build a bridge between **where you are** and a **working solution**,
then cross it step by step.
(See PDF on the bridge method)

Debugging fallbacks

- Try things out
- Poke around in the debugger
- Look at examples/tutorials
- Look at documentation
- Do a Google search
(focus on Stack Overflow)
- Post on Stack Overflow
- Post on product forums
- Post on product mailing lists
- Open an issue on Github
- Use the rubber duck
- Ask a friend or colleague
- Contract the work out

Reporting bugs

Reporting bugs

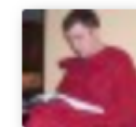
- 
-
- 
-
- 
-
- 
- Step-by-step instructions on how to recreate the bug
 - Make sure you've attempted to isolate the bug to what you are actually writing a bug against, instead of something else that could be the cause.
 - List attempts to isolate the bug to something other than the software you are writing a bug against
 - Make yourself available to answer questions and be available to help troubleshoot/recreate the bug

The bottom line is you have to engage some level of critical thinking when the bug is encountered. Once you've exhausted all possibilities that it could be your fault, write up a bug. If you find out it's your fault, but the software you are using/testing could have done something more usable to indicate it's your fault, still write a bug.

Also, to be a truly great bug-reporter, you must avail yourself to those testing the bug to help them recreate it. It's likely you've just "got the knack" for recreating that bug and there may be steps you are not conscious of. You can't just complain and walk away, participate in the process and help the team out by testing, recreating, and troubleshooting.

share edit flag

answered Oct 27 '08 at 15:39



Doug T.

47.3k ● 19 ● 109 ● 183

<https://stackoverflow.com/questions/240323/how-to-report-bugs-the-smart-way>

Reporting bugs

2

- Procedure used to re-create the bug including what was being done, what area of the application was being used and what event was happening at the time.
- Statement of reproduceability (reliably, not) - helps the developer know how hard it should be to reproduce so they don't give up too quickly
- Screen shots or documentation of error message / stack trace produced
- Criticality/Priority of the bug (can it be avoided, avoidance steps, is it catastrophic, does it have a business impact, what's the business risk, etc)
- Environment - which environment was the bug found in. Remote, local, etc.

Too often, our QA people think they can just put in a ticket saying, here's my exception without any backup documentation. It's near impossible to reproduce let alone fix the issue without more information.

share edit flag

answered Oct 27 '08 at 15:44



ScottCher

10.4k ● 5 ● 22 ● 25

<https://stackoverflow.com/questions/240323/how-to-report-bugs-the-smart-way>

Reporting bugs

<http://www.catb.org/~esr/faqs/smart-questions.html>

The Jupyter notebook

The Jupyter notebook

Interactive computing environment, supporting a variety of languages, which allows code, analysis and results to be saved in a “notebook.”

Spun off from the iPython project, a visual Python interpreter, in 2014; now supports other languages.

The logo references Galileo’s notebooks describing his discovery of the moons of Jupiter.



Jupyter keyboard commands

Shift and **enter**: execute current cell

Escape enters *command mode*

Escape, then **a**: add new cell above

Escape, then **b**: add new cell below

Escape, then **d**, then **d**: delete cell



A blue-tinted photograph of a modern glass building with a person sitting on a bench in the foreground. The text 'psycopg2' is overlaid on the left side of the image.

psycopg2

psycopg2



psycopg2: a Python interface to Postgres. It wraps libpq, the C interface to Postgres.

psycopg2

Probably named after psyco, an unmaintained JIT compiler for Python, and pg for postgres

Aims to:

- support heavy multi-threading and release the Python interpreter while connecting or exchanging data with the DB;
- have an aggressive caching policy for the physical database connections (opening a new connection has quite an high overhead);
- have a nice, non hard-coded way, of mapping PostgreSQL types to Python ones



[<https://www.free-soft.org/FSM/english/issue01/fog.html>]

psycopg2

Concepts in psycopg2:

- The connection: a link between you and the database server (can be open or closed)
- The cursor: a tool used to run queries and store our place when we are (gradually) loading results. One connection can have many cursors.

Operations in psycopg2:

- Execute a query
- Fetch results from a query (*this is done separately*)
- Commit a transaction
- Roll back a transaction



psycopg2

```
import psycopg2 as pg
```

Open connection:

```
connection =
```

```
pg.connect( 'postgres://postgres:postgres@localhost/dvdrental' )
```

Make a cursor:

```
cursor = connection.cursor()
```

Run a query:

```
query = 'SELECT * FROM film'
```

```
cursor.execute(query)
```

Fetch results:

```
results = cursor.fetchone()
```

or

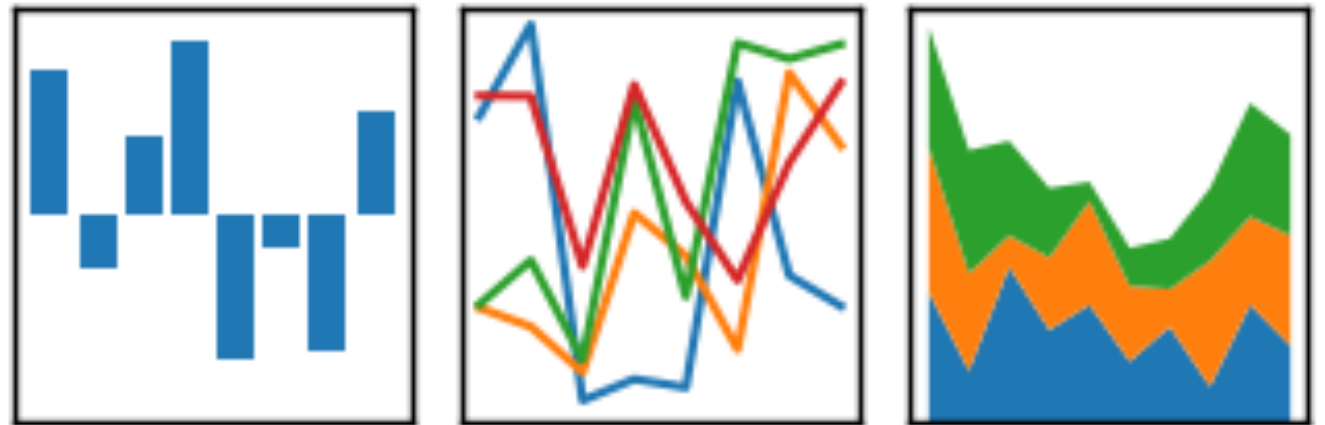
```
results = cursor.fetchall()
```

Pandas

A blue-tinted photograph of a modern glass building with a person sitting on a bench in the foreground. The building's glass facade reflects the surrounding environment. In the foreground, a person is sitting on a white, rounded bench, looking down at a device. The ground is paved with light-colored tiles. The overall scene is modern and urban.

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



A fast, popular Python analysis library, originally intended for financial time series data.

Developed in 2008 by Wes McKinney, working at a capital management firm at the time.

Pandas is from the econometric term **panel data** (time series)

The background of the slide is a photograph of a modern building with a glass facade, reflecting the surrounding environment. In the foreground, there is a paved plaza with several blue and white modular seating blocks. A person is sitting on one of these blocks, looking down at a device. The entire image is overlaid with a semi-transparent blue filter.

ORMs (Object-relational mappers)

ORMs

Object-relational mappers connect database tables with Python objects (or objects in another programming language).

A class is a definition which allows us to make objects. For example, we may define class Dog with name and breed:

```
Class Dog():  
    def __init__(self, name, breed):  
        self.name = name  
        self.breed = breed
```

ORMs

Once we have the class, we can define objects of class Dog:

```
dog_1 = Dog('Fido', 'Great Dane')  
dog_2 = Dog('Rover', 'Labrador')
```

```
print(dog_1.name)  
    Fido
```

```
print(dog_2.name)  
    Rover
```

ORMs

A class definition that corresponds to a database table is called a **model**.

For example, this is our model for Film:

```
class Film(db.Model):  
    __tablename__ = 'film'  
    film_id = db.Column(db.Integer, primary_key=True)  
    title = db.Column(db.String(255), index=True, unique=True)  
    description = db.column(db.Text())  copies = relationship('Inventory')
```


ORMs

Once we have this model,

- We can make Film objects and have SQLAlchemy add them to the database for us
- We can query the database and SQLAlchemy will automatically make Film objects for us

ORMs

```
CACHE Competence Load (0.0ms) SELECT "competences".* FROM "competences" WHERE "competences"."profile_id" = $1 AND "competences"."skill_id" = $2 LIMIT $3 [{"profile_id", 23}, {"skill_id", 93618}, {"LIMIT", 1}]
CACHE Competence Load (0.0ms) SELECT "competences".* FROM "competences" WHERE "competences"."profile_id" = $1 AND "competences"."skill_id" = $2 LIMIT $3 [{"profile_id", 23}, {"skill_id", 93629}, {"LIMIT", 1}]
↳ app/controllers/pathways_controller.rb:354:in `block in get_learnershape'
↳ app/controllers/pathways_controller.rb:354:in `block in get_learnershape'
CACHE Competence Load (0.0ms) SELECT "competences".* FROM "competences" WHERE "competences"."profile_id" = $1 AND "competences"."skill_id" = $2 LIMIT $3 [{"profile_id", 23}, {"skill_id", 93629}, {"LIMIT", 1}]
↳ app/controllers/pathways_controller.rb:355:in `block in get_learnershape'
Competence Load (1.7ms) SELECT "competences".* FROM "competences" WHERE "competences"."profile_id" = $1 AND "competences"."skill_id" = $2 LIMIT $3 [{"profile_id", 45}, {"skill_id", 93618}, {"LIMIT", 1}]
Competence Load (1.2ms) SELECT "competences".* FROM "competences" WHERE "competences"."profile_id" = $1 AND "competences"."skill_id" = $2 LIMIT $3 [{"profile_id", 23}, {"skill_id", 93628}, {"LIMIT", 1}]
↳ app/controllers/pathways_controller.rb:356:in `block in get_learnershape'
↳ app/controllers/pathways_controller.rb:351:in `block in get_learnershape'
CACHE Competence Load (0.0ms) SELECT "competences".* FROM "competences" WHERE "competences"."profile_id" = $1 AND "competences"."skill_id" = $2 LIMIT $3 [{"profile_id", 23}, {"skill_id", 93628}, {"LIMIT", 1}]
```

SQLAlchemy

The background of the slide is a photograph of a modern building with a glass facade, reflecting the surrounding environment. In the foreground, there is a paved plaza with several large, light-colored, rectangular concrete blocks arranged in a row. A person is sitting on one of these blocks, looking down at a device. The entire image is overlaid with a semi-transparent blue filter.

sqlalchemy



SQLAlchemy goes one step further than psycopg2: it is a full object-relational mapper, allowing database entities and operations to be manipulated in pure Python.

Python and the Flask web server



The Jinja templating language

- `{% ... %}` for Statements
- `{{ ... }}` for Expressions to print to the template output
- `{# ... #}` for Comments not included in the template output
- `# ... ##` for Line Statements



Flask: The simplest web server

Python and the Flask web server



Defining a model

```
class Inventory(db.Model):
```

Defining a model

```
class Inventory(db.Model):  
    film_id = db.Column(db.Integer(), primary_key=True)  
    inventory_id = db.Column(db.Integer(), ForeignKey('film.film_id'))  
    store_id = db.Column(db.Integer())  
  
    def __repr__(self): return str(self.inventory_id)
```

Setting up the database connection

```
class Config(object):  
    SQLALCHEMY_DATABASE_URI =  
        "postgres://imperial:imperial@imperial.csi1xfxus5vm.eu-west-  
        2.rds.amazonaws.com/dvdrental"  
  
from flask_sqlalchemy import SQLAlchemy  
  
app.config.from_object(Config)  
db = SQLAlchemy(app)
```

Making queries

```
filter_by(uuid=uuid).first()
```

```
.order_by(func.random())
```

```
db.session.add(c2)
```

```
db.session.commit()
```

SQLAlchemy

```
class Parent(Base):  
    __tablename__ = 'parent'  
    id = Column(Integer, primary_key=True)  
    children = relationship("Child")  
  
class Child(Base):  
    __tablename__ = 'child'  
    id = Column(Integer, primary_key=True)  
    parent_id = Column(Integer, ForeignKey('parent.id'))
```

<https://www.pythonsheets.com/notes/python-sqlalchemy.html>

SQLAlchemy

One To Many

A one to many relationship places a foreign key on the child table referencing the parent. `relationship()` is then specified on the parent, as referencing a collection of items represented by the child:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
```

<https://www.pythonsheets.com/notes/python-sqlalchemy.html>

SQLAlchemy

To establish a bidirectional relationship in one-to-many, where the “reverse” side is a many to one, specify an additional `relationship()` and connect the two using the `relationship.back_populates` parameter:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child", back_populates="parent")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
    parent = relationship("Parent", back_populates="children")
```

<https://www.pythonsheets.com/notes/python-sqlalchemy.html>

SQLAlchemy

Many To One

Many to one places a foreign key in the parent table referencing the child. `relationship()` is declared on the parent, where a new scalar-holding attribute will be created:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
```

<https://www.pythonsheets.com/notes/python-sqlalchemy.html>

SQLAlchemy

Bidirectional behavior is achieved by adding a second `relationship()` and applying the `relationship.back_populates` parameter in both directions:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", back_populates="parents")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parents = relationship("Parent", back_populates="child")
```

<https://www.pythonsheets.com/notes/python-sqlalchemy.html>

SQLAlchemy

One To One

One To One is essentially a bidirectional relationship with a scalar attribute on both sides. To achieve this, the **uselist** flag indicates the placement of a scalar attribute instead of a collection on the “many” side of the relationship. To convert one-to-many into one-to-one:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child = relationship("Child", uselist=False, back_populates="parent")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
    parent = relationship("Parent", back_populates="child")
```

<https://www.pythonsheets.com/notes/python-sqlalchemy.html>

SQLAlchemy

Or for many-to-one:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", back_populates="parent")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent = relationship("Parent", back_populates="child", uselist=False)
```

<https://www.pythonsheets.com/notes/python-sqlalchemy.html>

Transaction and Connect Object

```
from sqlalchemy import create_engine

db_uri = 'sqlite:///db.sqlite'
engine = create_engine(db_uri)

# Create connection
conn = engine.connect()
# Begin transaction
trans = conn.begin()
conn.execute('INSERT INTO "EX1" (name) '
             'VALUES ("Hello")')

trans.commit()
# Close connection
conn.close()
```

<https://www.pythonsneets.com/notes/python-sqlalchemy.html>

Many To Many

Many to Many adds an association table between two classes. The association table is indicated by the **secondary** argument to **relationship()**. Usually, the **Table** uses the **MetaData** object associated with the declarative base class, so that the **ForeignKey** directives can locate the remote tables with which to link:

```
association_table = Table('association', Base.metadata,
    Column('left_id', Integer, ForeignKey('left.id')),
    Column('right_id', Integer, ForeignKey('right.id'))
)

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
        secondary=association_table)

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

n-sqlalchemy.html

Inspect - Get Database Information

```
from sqlalchemy import create_engine
from sqlalchemy import inspect

db_uri = 'sqlite:///db.sqlite'
engine = create_engine(db_uri)

inspector = inspect(engine)

# Get table information
print inspector.get_table_names()

# Get column information
print inspector.get_columns('EX1')
```

<https://www.pythonsheets.com/notes/python-sqlalchemy.html>

Speeding up loading data

How do we insert thousands or hundreds of thousands of rows efficiently?

ORM insert methods are often slow.

There are special "bulk insert" methods but they are are also frequently slow.

In practice, the fastest method is often to handcraft a single SQL statement with *lots of rows inserted at once*. Python can be used to put the statement together.