Coding Bootcamp Code in Python

DEBUGGING PYTHON

Errors & warnings: pylint, flake8

- Static cod analysis, errors, warnings, code quality suggestions

```
$ pylint  add.py
************* Module add
C:  1, 0: Missing module docstring (missing-docstring)
E:  4,10: Undefined variable 'x' (undefined-variable)

Report
======
3 statements analysed.
…
```

```
#!/usr/bin/env python

if __name__ == '__main__':
    print(x + 3)
```

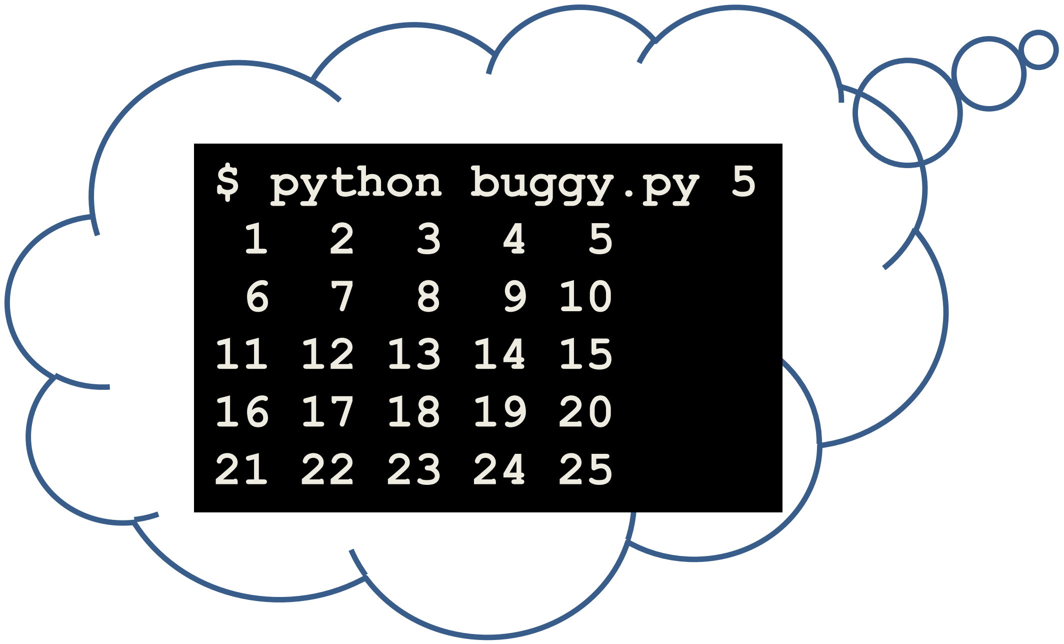
- flake8 can be invoked from vim, as git hook

Use classic debugger

- Bugs are ubiquitous…
- Debugging by `print`?
  - easy to do
  - takes a long time for complex situations
  - unstructured process
  - pollutes code
- Use debugger (pdb for Python): it can
  - step through code, statement by statement
  - inspect variable values
  - …

Okay, what's this?!?

```
def main():
    n = int(sys.argv[1])
    matrix = [[0] * n] * n
    for i in range(n):
        for j in range(n):
            matrix[i][j] = i*n + j + 1
    print('\n'.join([' '.join(['{:2d}'.format(e) for e in row])
                             for row in matrix]))
    return 0
```



```
$ python buggy.py 5
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

In your dreams!

```
$ python buggy.py 5
21 22 23 24 25
21 22 23 24 25
21 22 23 24 25
21 22 23 24 25
21 22 23 24 25
```

Starting & viewing source

- Starting the debugger

```
$ python -m pdb buggy.py 5
> ./buggy.py(3)<module>()
-> import sys
(Pdb)
```

Statement about to be executed

debugger prompt

- Listing source code: `l [<line-nr>]` (list)

```
(Pdb) l
  1     #!/usr/bin/env python
  2
  3  ->         import sys
  4
  5     def main():
  6         n = int(sys.argv[1])
```

Stepping

- Execute statement: `n` (next)

```
(Pdb) n
> ./buggy.py(5)<module>()
-> def main():
(Pdb)
> ./buggy.py(14)<module>()
-> if __name__ == '__main__':
(Pdb)
> ./buggy.py(15)<module>()
-> status = main()
```

- Step into function: `s` (step)

```
(Pdb) s
> ./buggy.py(5)<module>()
-> def main():
```

`r` (return): run until current function returns

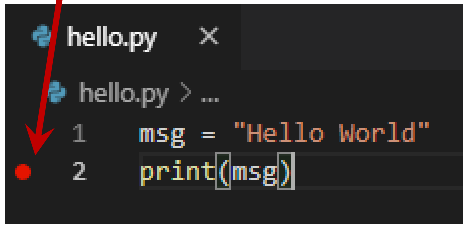
Printing values: variables

- Print variable values: `p <var>` (print)

```
(Pdb) n
> ./buggy.py(6)main()
-> n = int(sys.argv[1])
(Pdb)
> ./buggy.py(7)main()
-> matrix = [[0] * n] * n
(Pdb) p n
5
(Pdb) n
-> for i in range(n):
(Pdb) p matrix
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

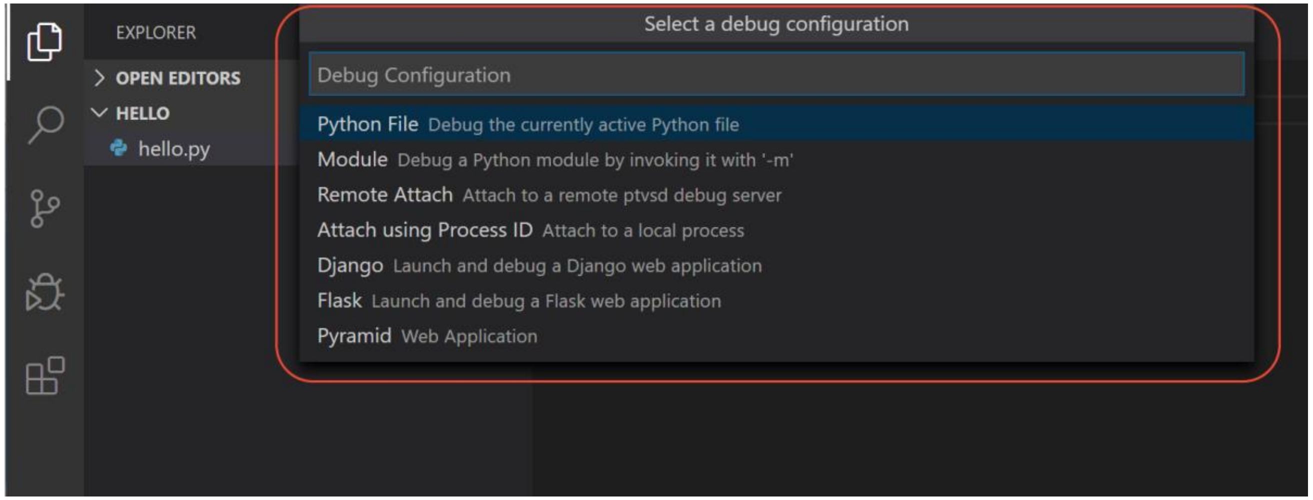
Mouse Click or F9

VSCode Debugger



```
msg = "Hello World"
print(msg)
```

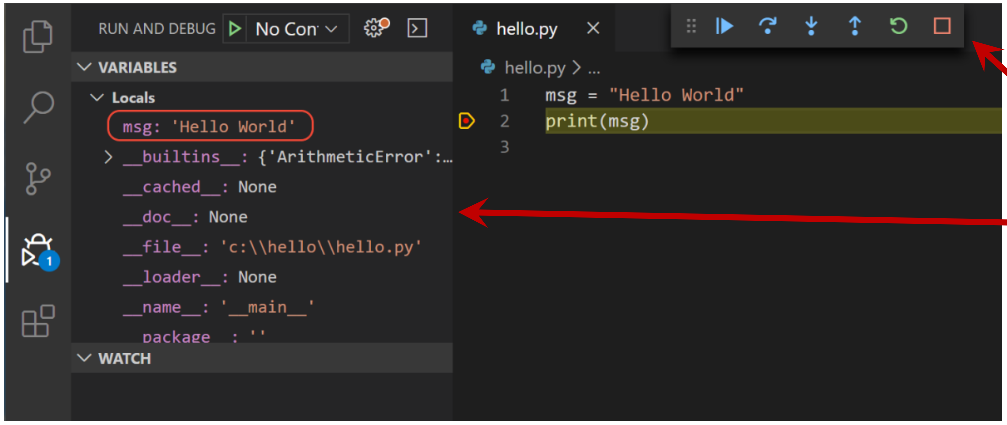
F5



Select a debug configuration

Debug Configuration

Python File	Debug the currently active Python file
Module	Debug a Python module by invoking it with '-m'
Remote Attach	Attach to a remote ptvsd debug server
Attach using Process ID	Attach to a local process
Django	Launch and debug a Django web application
Flask	Launch and debug a Flask web application
Pyramid	Web Application



```
msg: 'Hello World'
> __builtins__: {'ArithmeticError':…
__cached__: None
__doc__: None
__file__: 'c:\\hello\\hello.py'
__loader__: None
__name__: '__main__'
__package__: ''
```

```
msg = "Hello World"
print(msg)
```

Control your execution and the variables

Code Pack 20

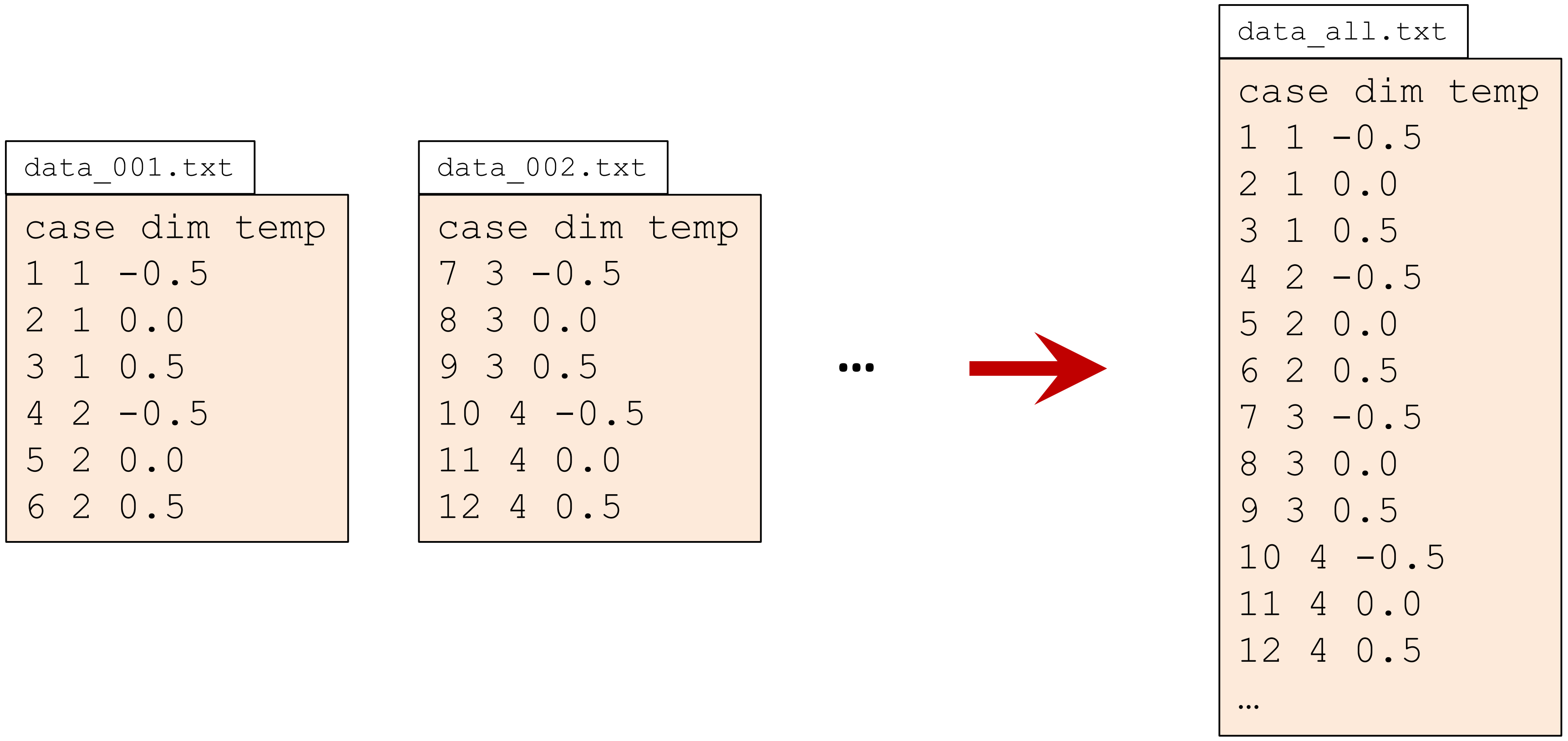
A. Debug your code

Coding Bootcamp Code in Python

FILE SYSTEM OPERATIONS: HANDLING FILES AND DIRECTORIES

Working with files in directories

- Directory contains files `data_001.txt`, `data_002.txt`,…



`data_001.txt`

```
case dim temp
1 1 -0.5
2 1 0.0
3 1 0.5
4 2 -0.5
5 2 0.0
6 2 0.5
```

`data_002.txt`

```
case dim temp
7 3 -0.5
8 3 0.0
9 3 0.5
10 4 -0.5
11 4 0.0
12 4 0.5
```

…

`data_all.txt`

```
case dim temp
1 1 -0.5
2 1 0.0
3 1 0.5
4 2 -0.5
5 2 0.0
6 2 0.5
7 3 -0.5
8 3 0.0
9 3 0.5
10 4 -0.5
11 4 0.0
12 4 0.5
…
```

Using glob

```
from argparse import ArgumentParser, FileType
from pathlib import Path
…
def main():
    arg_parser = ArgumentParser(description='…')
    arg_parser.add_argument('-o', dest='output_file',
                            type=FileType('w'), help='…')
    arg_parser.add_argument('-p', dest='pattern', help='…')
    options = arg_parser.parse_args()
    is_header_printed = False
    path = Path('.')
    for file_name in path.glob(options.pattern):
        with open(file_name, 'r') as input_file:
            header = input_file.readline()
            if not is_header_printed:
                options.output_file.write(header)
                is_header_printed = True
            for line in input_file:
                if line.strip():
                    options.output_file.write(line)
    return 0
```

Same as in Bash shell

```
$ python concat_data.py  -o data.txt  -p 'data_*.txt'
```

Path operations

- Many operations in `pathlib` package
  - Current working directory: `Path.cwd()`
  - Create path:

```
path = Path.cwd() / 'data' / 'output.txt'
    path == '/home/gjb/Tests/data/output.txt'
```

Will do the right thing for each OS

  - Dissecting paths:

```
• filename = path.name
      name == 'test.txt'
• dirname = path.parent
      dirname == '/home/gjb/data'
• parts = path.parts
      parts == ('/', 'home', 'gjb', 'data', 'output.txt')
• ext = path.suffix
      ext == '.txt'
• dirname = Path('/home/gjb/Tests').name
      dirname == 'Tests'
• ext = Path('/home/gjb/Tests/').suffix
      ext == ''
```

File system tests

- File tests:
  - `path.exists()`: `True` if `path` exists
  - `path.is_file()`: `True` if `path` is file
  - `path.is_dir()`: `True` if `path` is directory
  - `path.is_symlink()`: `True` if `path` is link
  - `pathlib.os.access(path,`
    `pathlib.os.R_OK)`:
    `True` if `path` can be read
    - `pathlib.os.R_OK`: read permission
    - `pathlib.os.W_OK`: write permission
    - `pathlib.os.X_OK`: execute permission

However: ask forgiveness, not permission!

Copying, moving, deleting

- Functions in `os` and `shutil` modules
  - copy file: `shutil.copy(source, dest)`
  - copy file, preserving ownership, timestamps: `shutil.copy2(source, dest)`
  - move file: `path.replace(dest)`
  - delete file: `path.unlink()`
  - remove non-empty directory: `path.rmdir()`
  - remove directory: `shutil.rmtree(directory)`
  - create directory: `path.mkdir()`

Temporary files

- Standard library `tempfile` package
  - Creating file with guaranteed unique name: `tempfile.NamedTemporaryFile(…)`

```
import tempfile
…
tmp_file = tempfile.NamedTemporaryFile(mode='w', dir='.',
                                       suffix='.txt', delete=False)
print("created temp file '{0}'".format(tmp_file.name))
with tmp_file.file as tmp:
    …
    tmp.write(…)
    …
```

File names such as `tmpD45x.txt`

Walking the tree

- Walking a directory tree: `os.walk(…)`, e.g., print name of Python files in (sub)directories

```
import os
…
for directory, _, file_names in os.walk(dir_name):
    for file_name in file_names:
        _, ext = os.path.splitext(file_name)
        if ext == target_ext:
            print(os.path.join(directory, file_name))
…
```

- For each directory, tuple:
  - directory name
  - list of subdirectories
  - list of files in directory

For simple cases, use `path.rglob(…)`

Code Pack 21

- See the Files

os_module.py

Coding Bootcamp Code in Python

MORE ICING ON APPLICATION: THREADING, MULTIPROCESS, SECURITY

threading

- The threading module makes working with threads much easier and allows the program to run multiple operations at once.
  - Can share memory

```
import threading

def doubler(number):
    """
    A function that can be used by a thread
    """
    print(threading.currentThread().getName() + '\n')
    print(number * 2)
    print()

if __name__ == '__main__':
    for i in range(5):
        my_thread = threading.Thread(target=doubler, args=(i,))
        my_thread.start()
```

multiprocessing

- The Process class is very similar to the threading module’s Thread class, but by process.

```
import os
from multiprocessing import Process

def doubler(number):
result = number * 2
    proc = os.getpid()
    print('{0} doubled to {1} by process id: {2}'.format(
        number, result, proc))

if __name__ == '__main__':
    numbers = [5, 10, 15, 20, 25]
    procs = []

    for index, number in enumerate(numbers):
        proc = Process(target=doubler, args=(number,))
        procs.append(proc)
        proc.start()

    for proc in procs:
        proc.join()
```

The cryptography Package

```
#pip install cryptography

from cryptography.fernet import Fernet
cipher_key = Fernet.generate_key()
print (cipher_key)
#b'APM1JDVgT8WDGOWBgQv6EIhvxl4vDYvUnVdg-Vjdt0o='

cipher = Fernet(cipher_key)
text = b'My super secret message'
encrypted_text = cipher.encrypt(text)
print (encrypted_text)
#(b'gAAAAABXOnV86aeUGADA6mTe9xEL92y_m0_TlC9vcqaF6NzHqRKkjEqh4d21P
InEP3C9HuiUkS9f'
# b'6bdHsSlRiCNWbSkPuRd_62zfEv3eaZjJvLAm3omnya8=')

decrypted_text = cipher.decrypt(encrypted_text)
print (decrypted_text)
#b'My super secret message'
```

Code Pack 22

- See the files

1.threading

2.multiprocessing

3.The_cryptography_Package

Coding Bootcamp Code in Python

RELATIONAL DATABASES: PYTHON DB API & SQLALCHEMY ORM

Accessing relational databases

- Relational databases:
  - great to store structured data, table-oriented
  - can be accessed easily via command line, programming language, GUI
  - can be queried using SQL
  - examples: MySQL, PostgreSQL, Oracle, DB2, SQLite3,…
- Using DB from Python via standard interface
  - Support for sqlite3 built-in, ok for simple applications
- For non-trivial stuff, use SQLAlchemy
  - Object-relational mapping (ORM)
  - Connectors to many RDBMS

SQL

- Create table to store data

```
CREATE TABLE IF NOT EXISTS weather (
     city_name       TEXT    NOT NULL,
     date            TEXT    NOT NULL,
     temperature     REAL    NOT NULL);
```

- Store data

```
INSERT INTO weather (city_name, date, emperature
     VALUES ('London', '2012-03-14', 13.2);
```

- Query data

```
SELECT city_name, AVG(temperature) FROM weather
     WHERE date BETWEEN '2012-01-01' AND '2012-01-31'
     GROUP BY city_name;
```

- Modify data

```
UPDATE weather SET city_name = 'St. Petersburg'
     WHERE city_name = 'Leningrad';
```

Databases - The use of Basic SQL Syntax

- Packages to use
- ~~adodbapi~~
  - Not maintained
- pyodbc
  - C++
- pypyodbc
  - pure python
- MySQLdb
- psycopg2

```
import psycopg2  #postgresql

conn =
psycopg2.connect(dbname='my_databa
se', user='username')
cursor = conn.cursor()

# execute a query
cursor.execute("SELECT * FROM
table_name")
row = cursor.fetchone()

# close your cursor and connection
cursor.close()
conn.close()
```

Python DB access: inserting data

- Connect to a database & create cursor

```
import sqlite3
conn = sqlite3.connect('weather-db')
cursor = conn.cursor()
```

- Insert data tuples

```
for data in generate_data(nr_cities, start, end):
    cursor.execute('''INSERT INTO weather
                          (city_name, date, temperature)
                          VALUES (?, ?, ?)''',
                   data)
conn.commit()
cursor.close()
```

tuple

Python DB access: querying

- Compute average temperature for period per city

```
conn = sqlite3.connect('weather-db')
conn.row_factory = sqlite3.Row
cursor = conn.cursor()
cursor.execute(
  '''SELECT city_name, AVG(temperature) AS 'temperature'
       FROM weather WHERE date BETWEEN ? AND ?
       GROUP BY city_namee''',
  (start, end))
for row in cursor:
  print('{city}\t{tmp}'.format(city=row['city_name'],
                               tmp=row['temperature']))
cursor.close()
```

SQLAlchemy: ORM

- Define classes/tables

```
from sqlalchemy import (Column, ForeignKey, UniqueConstraint,
                        Integer, String, DateTime, Float)
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class City(Base):
    __tablename__ = 'cities'
    city_id = Column(Integer, primary_key=True)
    name = Column(String(100), nullable=False, unique=True)
```

class ≡ table

object attributes

class attribute ≡ column definition

column properties

SQLAlchemy: relationships

- Define relationship

```
class Measurement(Base):
    __tablename__ = 'measurements'
    __table_args__ = (
        UniqueConstraint('time', 'city_id'),
    )
    measurement_id = Column(Integer, primary_key=True)
    time = Column(DateTime, nullable=False)
    temperature = Column(Float, nullable=False)
    city_id = Column(Integer, ForeignKey('cities.city_id'))
    city = relationship(City)
```

table constraint

column properties

relationship for ORM queries

SQLAlchemy: create tables

- To interact, create engine

```
from sqlalchemy import create_engine
…
engine = create_engine('sqlite:///{0}'.format(db_name))
```

- Creating tables ≡ setting metadata

```
Base.metadata.create_all(engine)
```

That's it!

SQLAlchemy: inserts

- Create engine, session

```
…
from sqlalchemy.orm import sessionmaker
…
engine = create_engine('sqlite:///{0}'.format(db_name))
Base.metadata.bind = engine
DBSession = sessionmaker(bind=engine)
db_session = DBSession()
…
```

- Create and add objects

```
…
for city_name in ['New York', 'Leningrad', 'Paris']:
    city = City(name=city_name)
    db_session.add(city)
db_session.commit()
```

SQLAlchemy: inserting relationships

- Use objects to express relationships

```
…
for city in city_list:
    temperature = measuree_temperature(city)
    date = determine_date()
    measurement = Measurement(time=date,
                              temperature=temperature,
                              city=city)
    db_session.add(measurement)
db_session.commit()
…
```

use actual object

SQLAlchemy: queries

- Queries as method calls

```
…
city_list = db_session.query(City).all()
```

class ≡ table

- Natural join query

join on relationship

```
…
measurements = db_session.query(Measurement) \
                         .join('city') \
                         .filter(City.name == city_name,
                                 s_date <= Measurement.time,
                                 Measurement.time <= e_date) \
                         .all()
```

Note: class attributes!!!

SELECT * FROM … WHERE …

SQLAlchemy: updates

- Modify object attribute(s) ≡ update

```
…
leningrad = db_session.query(City) \
                      .filter(City.name == 'Leningrad') \
                      .one()
leningrad.name = 'Saint Petersburg'
db_session.commit()
```

don't forget commit!

SQLAlchemy: just classes

- Classes representing tables can have methods

```
class Measurement(Base):
…
    def __str__(self):
        fmt_str = '{city:s}: {time:s}\n\tT = {temp:.1f} C'
        return fmt_str.format(city=str(self.city),
                              time=str(self.time),
                              temp=self.temperature)
…
```

Pitfalls

- ORM "hides" database interaction
  - Easy to be inefficient
  - Object creation takes time
  - Can consume a lot of memory
  - Still necessary to understand
    - Relational model
    - How RDBMS works

Further reading: relational databases

- Introduction to relational database design
  http://www.ntu.edu.sg/home/ehchua/programming/sql/relational_database_design.html

Code Pack 23

- See the files:

The use of Basic SQL Syntax

Object Relational Mappers

Coding Bootcamp Code in Python

OTHER PYTHON BUILT-INS

any

- Will return True is any element in said iterable is True

```
print (all([0,0,1,0]))
#False

print (all([1,1,1,1]))
#True

print (any([0,0,0,1]))
#True

print (any([0,0,0,0]))
#False
```

- all built-in as it has similar functionality except that it will only return True if every single item in the iterable is True

enumerate

- This returns the position of each item in the iterable as well as the value

```
my_string = 'abcdefg'
for pos, letter in enumerate(my_string):
    print (pos, letter)

#0 a
#1 b
#2 c
#3 d
#4 e
#5 f
#6 g
```

eval

- Accepts strings and basically runs them

```
var = 10
source = 'var * 2'
print (eval(source))
#20
```

Can created a major
security breach

filter

- It will take a function and an iterable and return an iterator for those elements within the iterable for which the passed in function returns True

```
def less_than_ten(x):
    return x < 10

my_list = [1, 2, 3, 10, 11, 12]
for item in filter(less_than_ten, my_list):
    print(item)

#1
#2
#3
```

map

- The **map** built-in also takes a function and an iterable and return an iterator that applies the function to each item in the iterable

```
def doubler(x):
    return x * 2

my_list = [1, 2, 3, 4, 5]
for item in map(doubler, my_list):
    print(item)

#2
#4
#6
#8
#10
```

zip

- Takes a series of iterables and aggregates the elements from each of them

```
keys = ['x', 'y', 'z']
values = [5, 6, 7]
print (zip(keys, values))
#<zip object at 0x7faaad4dd848>

print (list(zip(keys, values)))
#[('x', 5), ('y', 6), ('z', 7)]
```

Code Pack 24

- See the files:

1. any
2. enumerate
3. eval
4. filter
5. map
6. zip