Extra topics for Python Boot Camp

John Strickler

Version 1.0, April 2024

Table of Contents

Chapter 1: Database Access

Objectives

- Understand the Python DB API architecture
- Connect to a database
- Execute simple and parameterized queries
- Fetch single and multiple row results
- Get metadata about a query
- Execute non-query statements
- Start transactions and commit or rollback as needed

© 2024 JS Consulting *(Licensed to Trivera)*

The DB API

- Most popular DB interface
- Specification, not abstract class
- Many modules for different DBMSs
- Hides actual DBMS implementation

To make database programming simpler, Python has the DB API. This is an API to standardize working with databases. When a package is written to access a database, it is written to conform to the API, and thus programmers do not have to learn a new set of methods and functions.

DB API objects and methods

```
conn = package.connect(server, db)
cursor = conn.cursor()
num_lines = cursor.execute(query)
num_lines = cursor.execute(query-with-placeholders, param-iterable)
num_lines = cursor.executemany(query-with-placeholders, nested-param-iterable)
all_rows = cursor.fetchall()
some_rows = cursor.fetchmany(n)
one_row = cursor.fetchone()
conn.commit()
conn.rollback()
```


© 2024 JS Consulting *(Licensed to Trivera)*

Table 1. Available Interfaces (using Python DB API-2.0)

Database	Python package
Firebird (and Interbase)	KInterbasDB
IBM DB2	ibm-db
Informix	informixdb
Ingres	ingmod
Microsoft SQL Server	pymssql
MySQL	pymysql
ODBC	pyodbc
Oracle	cx_oracle
PostgreSQL	psycopg2
SAP DB (also known as "MaxDB")	sapdbapi
SQLite	sqlite3
Sybase	Sybase

This list is not comprehensive, and there may be additional interfaces to some of the listed DBMSs.

Connecting to a Server

- Import appropriate library
- Use connect() to get a database object
- Specify host, database, username, password

To connect to a database server, import the package for the specific database. Use the package's `connect()` method to get a database object, specifying the host, initial database, username, and password. If the username and password are not needed, use None.

Argument names for the `connect()` method may not be consistent across packages. Most `connect()` methods use individual arguments, such as ***host***, ***database***, etc., but some use a single string argument.

When finished with the connection, call the `close()` method on the connection object.

Many database modules support the context manager (`with` statement), and will automatically close the database when the `with` block is exited. Check the documentation to see how this is implemented for a specific database.

Example

```
import pymysql

conn = pymysql.connect (host = "dbserver",
                        user = "adeveloper",
                        passwd = "s3cr3t",
                        db = "samples")
# Interact with database here ...
conn.close()
```

```
import sqlite3

with sqlite3.connect('sample.db') as conn:
    # Interact with database here ...
```


© 2024 JS Consulting *(Licensed to Trivera)*

Table 2. connect() examples

Database	Python package	Connection

IBM DB2	ibm-db	`import ibm_db_dbi as db2`
		`conn = db2.connect(`
		
		`"DATABASE=testdb;HOSTNAME=localhost;PORT=50000;PROTOCOL=TCPIP;UID=db2inst1;PWD=scripts;",`
		`"",`
		`""`
		`)`

Oracle	cx_oracle	`ip = 'localhost'`
		`port = 1521`
		`SID = 'YOURSIDHERE'`
		`dsn_tns = cx_Oracle.makedsn(ip, port, SID)`
		`db = cx_Oracle.connect('adeveloper', '$3cr3t', dsn_tns)`

PostgreSQL	psychopg	`psycopg2.connect ('''`
		`host='localhost'`
		`user='adeveloper'`
		`password='$3cr3t'`
		`dbname='testdb'`
		`''')`
		
		`connect()` has one `str` parameter, not multiple parameters#

MS-SQL	pymssql	`pymssql.connect (`
		`host="localhost",`
		`user="adeveloper",`
		`passwd="$3cr3t",`
		`db="testdb",`
		`)`
		`pymssql.connect (`
		`dsn="DSN",`
		`)`

MySQL	pymysql	`pymysql.connect (`
		`host="localhost",`
		`user="adeveloper",`
		`passwd="$3cr3t",`
		`db="testdb",`
		`)`

Database	Python package	Connection

ODBC-compliant DB	pyodbc	`pyodbc.connect('''`
		`DRIVER={SQL Server};`
		`SERVER=localhost;`
		`DATABASE=testdb;`
		`UID=adeveloper;`
		`PWD=$3cr3t`
		`''')`
		
		`pyodbc.connect('DSN=testdsn;PWD=$3cr3t')`
		
		  `connect()` has one (string) parameter, not multiple parameters

SqlLite3	sqlite3	`sqlite3.connect('testdb')  # on-disk database(single file)` `sqlite3.connect(':memory:')  # in-memory database`

© 2024 JS Consulting *(Licensed to Trivera)*

Creating a Cursor

- Cursor can execute SQL statements
- Create with `cursor()` method
- Multiple cursors available
  - Standard cursor
    - Returns tuples
  - Other cursors
    - Returns dictionaries
    - Leaves data on server

Once you have a connection object, you can call `cursor()` to create a cursor object. A cursor is an object that can execute SQL code and fetch results. One connection may have one or more active cursors.

The default cursor for most packages returns each row as a tuple of values. There are different types of cursors that can return data in different formats, or that control whether data is stored on the client or the server.

See **db_*.py** for examples using DB2, Postgres, MySQL, and MS-SQL. Most of the **sqlite3** examples in this chapter are also implemented for MySQL, Postgres, and DB2, plus a few extras.

Example

```
import sqlite3
conn = sqlite3.connect("sample.db")
cursor = conn.cursor()
```

Querying data

- `_cursor_.execute(_query_)`
  - Gets all data from query
  - Returns # rows in result set
- Use **fetch...** methods
  - `.fetchall()`
  - `.fetchone()`
  - `.fetchmany()`
- Return rows as tuples of values

Once you have a cursor, you can use it to execute queries via the `execute()` method. The first argument to `execute()` is a string containing one SQL statement.

For queries, `_cursor_.execute()` returns the number of rows in the result set. (In Sqlite3, `_cursor_.execute()` returns the cursor object, so you can say `__cursor__.execute(__query__).fetchall()`.)

Cursors provide three methods for returning query results.

`fetchone()` returns the next available row from the query results.

`fetchall()` returns a tuple of all rows.

`fetchmany(n)` returns up to n rows. This is useful when the query returns a large number of rows.

For all three methods, each row is returned as a tuple of values.

© 2024 JS Consulting *(Licensed to Trivera)*

Example

db_sqlite_basics.py

```
import sqlite3

# conn = sqlite3.Connection(...)
with sqlite3.connect("../DATA/presidents.db") as conn:  # connect to the database

    cursor = conn.cursor()  # get a cursor object

    # select first name, last name from all presidents
    cursor.execute('''
        select *
        from presidents
    ''')  # execute a SQL statement

    print("Sqlite3 does not provide a row count\n")  # (included for consistency with
other DBMS modules)

    for row in cursor.fetchall():  # fetchall() returns all rows
        print(row)
#        print(' '.join(row))  # each row is a tuple
```

db_sqlite_basics.py

```
(37, 'Nixon', 'Richard Milhous', '1969-01-20', '1974-08-09', 'Yorba Linda', 'California',
'1913-01-09', '1994-04-22', 'Republican')
(38, 'Ford', 'Gerald Rudolph', '1974-08-09', '1977-01-20', 'Omaha', 'Nebraska', '1913-07-
14', '2006-12-26', 'Republican')
(39, 'Carter', "James Earl 'Jimmy'", '1977-01-20', '1981-01-20', 'Plains', 'Georgia',
'1924-10-01', None, 'Democratic')
(40, 'Reagan', 'Ronald Wilson', '1981-01-20', '1989-01-20', 'Tampico', 'Illinois', '1911-
02-06', '2004-06-05', 'Republican')
(41, 'Bush', 'George Herbert Walker', '1989-01-20', '1993-01-20', 'Milton',
'Massachusetts', '1924-06-12', None, 'Republican')
(42, 'Clinton', "William Jefferson 'Bill'", '1993-01-20', '2001-01-20', 'Hope',
'Arkansas', '1946-08-19', None, 'Democratic')
(43, 'Bush', 'George Walker', '2001-01-20', '2009-01-20', 'New Haven', 'Connecticut',
'1946-07-06', None, 'Republican')
(44, 'Obama', 'Barack Hussein', '2009-01-20', '2017-01-20', 'Honolulu', 'Hawaii', '1961-
08-04', None, 'Democratic')
(45, 'Trump', 'Donald J', '2017-01-20', '2021-01-20', 'Queens, NYC', 'New York', '1946-
06-14', None, 'Republican')
(46, 'Biden', 'Joseph Robinette', '2021-01-20', None, 'Scranton', 'Pennsylvania', '1942-
11-10', None, 'Democratic')
```


© 2024 JS Consulting *(Licensed to Trivera)*

Non-query statements

- Updates database
- Returns # rows in result set
- Must commit changes

The `execute()`method is also used to execute non-query statements.

As with queries, the first argument is a string containing one SQL statement. The optional second argument is an iterable of values to fill in placeholders in a parameterized statement.

For most DB packages, `execute()` returns the number of rows affected.

Example

db_sqlite_add_row.py

```
from datetime import date
import sqlite3


with sqlite3.connect("../DATA/presidents.db") as s3conn:  # connect to database

    sql_insert = """
    insert into presidents
    (termnum, lastname, firstname, birthdate, deathdate, birthplace, birthstate,
termstart, termend, party)
    values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    """


    new_row_data = [47, 'Ramirez', 'Mary', date(1968, 9, 22), None,
                    'Topeka', 'Kansas', date(2025, 1, 20), None, 'Independent']

    cursor = s3conn.cursor()

    try:
        cursor.execute(sql_insert, new_row_data)
    except (sqlite3.OperationalError, sqlite3.DatabaseError, sqlite3.DataError) as err:
        print(err)
        s3conn.rollback()
    else:
        s3conn.commit()

    cursor.close()
```


© 2024 JS Consulting *(Licensed to Trivera)*

Example

db_sqlite_delete_row.py

```
from datetime import date
import sqlite3


with sqlite3.connect("../DATA/presidents.db") as conn:  # connect to DB

    sql_delete = """
    delete from presidents
    where TERMNUM = 47
    """

    cursor = conn.cursor()  # get a cursor

    try:
        cursor.execute(sql_delete)
    except (sqlite3.DatabaseError, sqlite3.OperationalError, sqlite3.DataError) as err:
        print(err)
        conn.rollback()
    else:
        conn.commit()

    cursor.close()
```

SQL Injection

- "Hijacks" SQL code
- Result of string formatting
- Always use parameterized statements

One kind of vulnerability in SQL code is called *SQL injection*. This occurs when an attacker embeds SQL commands in input data. This can happen when naively using string formatting to build SQL statements.

Since the programmer is generating the SQL code as a string, there is no way to check for malicious SQL code. It is best practice to use parameterized statements.

Example

db_sql_injection.py

```
#
good_input = 'Google'
malicious_input = "'; drop table customers; -- "  # input would come from a web form, for
instance

naive_format = "select * from customers where company_name = '{}' and company_id != 0"

good_query = naive_format.format(good_input)  # string formatting naively adds the user
input to a field, expecting only a customer name
malicious_query = naive_format.format(malicious_input)  # string formatting naively adds
the user input to a field, expecting only a customer name

print("Good query:")
print(good_query)  # non-malicious input works fine
print()

print("Bad query:")
print(malicious_query)  # query now drops a table ('--' is SQL comment)
```


© 2024 JS Consulting *(Licensed to Trivera)*

db_sql_injection.py

```
Good query:
select * from customers where company_name = 'Google' and company_id != 0

Bad query:
select * from customers where company_name = ''; drop table customers; -- ' and
company_id != 0
```

see http://www.xkcd.com/327 for a well-known web comic on this subject.

Parameterized Statements

- Prevent SQL injection
- More efficient updates
- Use placeholders in query
  - Placeholders vary by DB
- Pass iterable of parameters
- Use cursor.execute() or cursor.executemany()

For efficiency, you can iterate over of sequence of input datasets when performing a non-query SQL statement. The `execute()` method takes a query, plus an iterable of values to fill in the placeholders. The database manager will only parse the query once, then reuse it for subsequent calls to execute().

All SQL statements may be parameterized, including queries.

Parameterized statements also protect against SQL injection attacks.

Different database modules use different placeholders. To see what kind of placeholder a module uses, check MODULE.paramstyle. Types include *pyformat*, meaning *%s*, and *qmark*, meaning *?*.

The `executemany()` method takes a query, plus an iterable of iterables. It will call `execute()` once for each nested iterable.

Table 3. Placeholders for SQL Parameters

Python package	Placeholder
`pymysql`	`%s`
`cx_oracle`	`:param_name`
`pyodbc`	`?` `
`pymssql`	`%d` for `int`, `%s` for `str`, etc.
`Psychopg`	`%s` or `%(param_name)s` `
`sqlite3`	`?` or `:param_name` `

with the exception of **pymssql** the same placeholder is used for all column types.

© 2024 JS Consulting *(Licensed to Trivera)*

Example

db_sqlite_parameterized.py

```
import sqlite3

with sqlite3.connect("../DATA/presidents.db") as s3conn:
    s3cursor = s3conn.cursor()

    party_query = '''
    select firstname, lastname
    from presidents
        where party = ?
    '''  # ? is SQLite3 placeholder for SQL statement parameter; different DBMSs use
different placeholders

    for party in 'Federalist', 'Whig':
        print(party)
        s3cursor.execute(party_query, (party,))  # second argument to execute() is
iterable of values to fill in placeholders from left to right
        print(s3cursor.fetchall())
        print()
```

db_sqlite_parameterized.py

```
Federalist
[('John', 'Adams')]

Whig
[('William Henry', 'Harrison'), ('John', 'Tyler'), ('Zachary', 'Taylor'), ('Millard',
'Fillmore')]
```

Example

db_sqlite_bulk_insert.py

```
import sqlite3
import os
import csv

DATA_FILE = '../DATA/fruit_data.csv'

DB_NAME = 'fruits.db'
DB_TABLE = 'fruits'

SQL_CREATE_TABLE = f"""
create table {DB_TABLE} (
    id integer primary key,
    name varchar(30),
    unit varchar(30),
    unitprice decimal(6, 2)
)
"""  # SQL statement to create table

SQL_INSERT_ROW = f'''
insert into {DB_TABLE} (name, unit, unitprice) values (?, ?, ?)
'''  # parameterized SQL statement to insert one record

SQL_SELECT_ALL = f"""
select name, unit, unitprice from {DB_TABLE}
"""

def main():
    """
    Program entry point.

    :return: None
    """
    conn, cursor = get_connection()
    create_database(cursor)
    populate_database(conn, cursor)
    read_database(cursor)

    cursor.close()
    conn.close()


def get_connection():
    """
```


© 2024 JS Consulting *(Licensed to Trivera)*

```
    Get a connection to the PRODUCE database

    :return: SQLite3 connection object.
    """
    if os.path.exists(DB_NAME):
        os.remove(DB_NAME)  # remove existing database if it exists

    conn = sqlite3.connect(DB_NAME)  # connect to (new) database
    cursor = conn.cursor()
    return conn, cursor


def create_database(cursor):
    """
    Create the fruit table

    :param conn: The database connection
    :return: None
    """
    cursor.execute(SQL_CREATE_TABLE)  # run SQL to create table


def populate_database(conn, cursor):
    """
    Add rows to the fruit table

    :param conn: The database connection
    :return: None
    """
    with open(DATA_FILE) as file_in:
        fruit_data = csv.reader(file_in, quoting=csv.QUOTE_NONNUMERIC)

        try:
            cursor.executemany(SQL_INSERT_ROW, fruit_data)  # iterate over list of pairs
                                                            # and add each pair to
database
        except sqlite3.DatabaseError as err:
            print(err)
            conn.rollback()
        else:
            conn.commit()  # commit the inserts; without this, no data would be saved

def read_database(cursor):
    cursor.execute(SQL_SELECT_ALL)
    for name, unit, unitprice in cursor.fetchall():
        print(f'{name:12s} {unitprice:5.2f}/{unit}')
```

© 2024 JS Consulting *(Licensed to Trivera)*

```
if __name__ == '__main__':
    main()
```

db_sqlite_bulk_insert.py

```
pomegranate   0.99/each
cherry        2.25/pound
apricot       3.49/pound
date          1.20/pound
apple         0.55/pound
lemon         0.69/each
kiwi          0.88/each
orange        0.49/each
lime          0.49/each
watermelon    4.50/each
guava         2.88/pound
papaya        1.79/pound
fig           2.29/pound
pear          1.10/pound
banana        0.65/pound
```


© 2024 JS Consulting *(Licensed to Trivera)*

Dictionary Cursors

- Indexed by column name
- Not standardized in the DB API

The standard cursor provided by the DB API returns a tuple for each row. Most DB packages provide other kinds of cursors, including user-defined versions.

A very common cursor is a dictionary cursor, which returns a dictionary for each row, where the keys are the column names. Each package that provides a dictionary cursor has its own way of providing the dictionary cursor, although they all work the same way.

Table 4. Dictionary Cursors

Python package	How to get a dictionary cursor

pymysql	`import pymysql.cursors + conn = pymysql.connect(..., + cursorclass = pymysql.cursors.DictCursor + ) + dcur = conn.cursor()`
	all cursors will be dict cursors
	
	`dcur = conn.cursor( pymysql.cursors.DictCursor)`
	only this cursor will be a dict cursor

cx_oracle	*Not available*
pyodbc	*Not available*
pgdb	*Not available*
pymssql	`conn = pymssql.connect (..., as_dict=True) + dcur = conn.cursor()`
psychopg	`import psycopg2.extras + dcur = conn.cursor(cursor_factory=psycopg.extras.DictCursor)`
sqlite3	`conn = sqlite3.connect (..., row_factory=sqlite3.Row) + dcur = conn.cursor()` `conn.row_factory = sqlite3.Row + dcur = conn.cursor()`

© 2024 JS Consulting *(Licensed to Trivera)*

Example

db_sqlite_dict_cursor.py

```
import sqlite3

s3conn = sqlite3.connect("../DATA/presidents.db")
# uncomment to make _all_ cursors dictionary cursors
# conn.row_factory = sqlite3.Row

NAME_QUERY = '''
    select firstname, lastname
    from presidents
    where termnum < 5
'''

cur = s3conn.cursor()

# select first name, last name from all presidents
cur.execute(NAME_QUERY)

for row in cur.fetchall():
    print(row)
print('-' * 50)

dict_cursor = s3conn.cursor()  # get a normal SQLite3 cursor

# make _this_ cursor a dictionary cursor
dict_cursor.row_factory = sqlite3.Row  # set the row factory to be a Row object

# Row objects are dict/list hybrids -- row[name] or row[pos]

# select first name, last name from all presidents
dict_cursor.execute(NAME_QUERY)

for row in dict_cursor.fetchall():
    print(row['firstname'], row['lastname'])  # index row by column name

print('-' * 50)
```

db_sqlite_dict_cursor.py

```
('George', 'Washington')
('John', 'Adams')
('Thomas', 'Jefferson')
('James', 'Madison')
------------------------------------------------
George Washington
John Adams
Thomas Jefferson
James Madison
------------------------------------------------
```


© 2024 JS Consulting *(Licensed to Trivera)*

Metadata

- cursor.description returns tuple of tuples
- Fields
  - name
  - type_code
  - display_size
  - internal_size
  - precision
  - scale
  - null_ok

Once a query has been executed, the cursor's description attribute is a tuple with metadata about the columns in the query. It contains one tuple for each column in the query, containing 7 values describing the column.

For instance, to get the names of the columns, you could say `names = [d[0] for d in cursor.description]`

For non-query statements, cursor.description returns None.

The names are based on the query (with possible aliases), and not necessarily on the names in the table.

Sqlite3 only provides column names.

Generic alternate cursors

- Create generator function
  - Get column names from *cursor*.description()
  - For each row
    - Make object from column names and values
      - Dictionary
      - Named tuple
      - Dataclass

Many database modules have a dictionary cursor built in. For those that don't the `iterrows_asdict()` function can be used with a cursor from any DB API-compliant package.

The example uses the metadata from the cursor to get the column names, and forms a dictionary by zipping the column names with the column values. `db_iterrows` also provides `iterrows_asnamedtuple()`, which returns each row as a named tuple.

The functions in `db_iterrows` return generator objects. When you loop over the generator object, each element is a dictionary or a named tuple, depending on which function you called.

© 2024 JS Consulting *(Licensed to Trivera)*

Example

db_iterrows.py

```
"""
Generic functions that can be used with any DB API compliant
package.

To use, pass in a cursor after execute()-ing a
SQL query. Then iterate over the generator that is
returned
"""
from collections import namedtuple
from dataclasses import make_dataclass

def get_column_names(cursor):
    return [desc[0] for desc in cursor.description]

def iterrows_asdict(cursor):
    '''Generate rows as dictionaries'''
    column_names = get_column_names(cursor)
    for row in cursor.fetchall():
        row_dict = dict(zip(column_names, row))
        yield row_dict

def iterrows_asnamedtuple(cursor):
    '''Generate rows as named tuples'''
    column_names = get_column_names(cursor)
    Row = namedtuple('Row', column_names)
    for row in cursor.fetchall():
        yield Row(*row)

def iterrows_asdataclass(cursor):
    '''Generate rows as dataclass instances'''
    column_names = get_column_names(cursor)
    Row = make_dataclass('row_tuple', column_names)

    for row in cursor.fetchall():
        yield Row(*row)
```

Example

db_sqlite_iterrows.py

```
"""
Generic functions that can be used with any DB API compliant
package.

To use, pass in a cursor after execute()-ing a
SQL query. Then iterate over the generator that is
returned
"""
import sqlite3
from db_iterrows import *

sql_select = """
SELECT firstname, lastname, party
FROM presidents
WHERE termnum > 39
"""

conn = sqlite3.connect("../DATA/presidents.db")

cursor = conn.cursor()

cursor.execute(sql_select)

for row in iterrows_asdict(cursor):
    print(row['firstname'], row['lastname'], row['party'])

print('-' * 60)


cursor.execute(sql_select)

for row in iterrows_asnamedtuple(cursor):
    print(row.firstname, row.lastname, row.party)

print('-' * 60)


cursor.execute(sql_select)

for row in iterrows_asdataclass(cursor):
    print(row.firstname, row.lastname, row.party)
```


© 2024 JS Consulting *(Licensed to Trivera)*

db_sqlite_iterrows.py

```
Ronald Wilson Reagan Republican
George Herbert Walker Bush Republican
William Jefferson 'Bill' Clinton Democratic
George Walker Bush Republican
Barack Hussein Obama Democratic
Donald J Trump Republican
Joseph Robinette Biden Democratic
------------------------------------------------------------
Ronald Wilson Reagan Republican
George Herbert Walker Bush Republican
William Jefferson 'Bill' Clinton Democratic
George Walker Bush Republican
Barack Hussein Obama Democratic
Donald J Trump Republican
Joseph Robinette Biden Democratic
------------------------------------------------------------
Ronald Wilson Reagan Republican
George Herbert Walker Bush Republican
William Jefferson 'Bill' Clinton Democratic
George Walker Bush Republican
Barack Hussein Obama Democratic
Donald J Trump Republican
Joseph Robinette Biden Democratic
```

Transactions

- Transactions allow safer control of updates
- commit() to save transactions
- rollback() to discard

Sometimes a database task involves more than one change to your database (i.e., more than one SQL statement). You don't want the first SQL statement to succeed and the second to fail; this would leave your database in a corrupt state.

To be certain of data integrity, use **transactions**. This lets you make multiple changes to your database and only commit the changes if all the SQL statements were successful.

For all packages using the Python DB API, a transaction is started when you connect. At any point, you can call `__CONNECTION__.commit()` to save the changes, or `__CONNECTION__.rollback()` to discard the changes. If you don't call `commit()` after modify a table, the data will not be saved.

You can also turn on *autocommit*, which calls `commit()` after every statement. See the table below for how autocommit is implemented in various DB packages.

Table 5. How to turn on autocommit

Package	Method/Attribute
cx_oracle	`conn.autocommit = True`
ibm_db_api	`conn.set_autocommit(True)`

pymysql	`pymysql.connect(…, autocommit=True)`
	or
	`conn.autocommit(True)`

psycopg2	`conn.autocommit = True`
sqlite3	`sqlite3.connect(dbname, isolation_level=None)`

pymysql only supports transaction processing when using the **InnoDB** engine

© 2024 JS Consulting *(Licensed to Trivera)*

Example

```
try:
    for info in list_of_tuples:
         cursor.execute(query,info)
except SQLError:
    dbconn.rollback()
else:
    dbconn.commit()
```

Object-relational Mappers

- No SQL required
- Maps a class to a table
- All DB work is done by manipulating objects
- Most popular Python ORMs
  - SQLAlchemy
  - Django (which is a complete web framework)

An Object-relational mapper is a module or framework that creates a level of abstraction above the actual database tables and SQL queries. As the name implies, a Python class (object) is mapped to the actual table.

The two most popular Python ORMs are SQLAlchemy which is a standalone ORM, and Django ORM. Django is a comprehensive Web development framework, which provides an ORM as a subpackage. SQLAlchemy is the most fully developed package, and is the ORM used by Flask and some other Web development frameworks.

Instead of querying the database, you call a search method on an object representing a table. To add a row to the table, you create a new instance of the table class, populate it, and call a method like save(). You can create a large, complex database system, complete with foreign keys, composite indices, and all the other attributes near and dear to a DBA, without writing the first line of SQL.

You can use Python ORMs in two ways.

One way is to design the database with the ORM. To do this, you create a class for each table in the database, specifying the columns with predefined classes from the ORM. Then you run an ORM command which executes the queries needed to build the database. If you need to make changes, you update the class definitions, and run an ORM command to synchronize the actual DBMS to your classes.

The second way is to map tables to an existing database. You create the classes to match the schemas that have already been defined in the database. Both SQLAlchemy and the Django ORM have tools to automate this process.

© 2024 JS Consulting *(Licensed to Trivera)*

NoSQL

- Non-relational database
- Document-oriented
- Can be hierarchical (nested)
- Examples
  - MongoDB
  - Cassandra
  - Redis

A current trend in data storage are called "NoSQL" or non-relational databases. These databases consist of *documents*, which are indexed, and may contain nested data.

NoSQL databases don’t contain tables, and do not have relations.

While relational databases are great for tabular data, they are not as good a fit for nested data. Geo-spatial, engineering diagrams, and molecular modeling can have very complex structures. It is possible to shoehorn such data into a relational database, but a NoSQL database might work much better. Another advantage of NoSQL is that it can adapt to changing data structures, without having to rebuild tables if columns are added, deleted, or modified.

Some of the most common NoSQL database systems are MongoDB, Cassandra and Redis.

© 2024 JS Consulting *(Licensed to Trivera)*

Example

mongodb_example.py

```
import re
from pymongo import MongoClient, errors

FIELD_NAMES = (
    'termnumber lastname firstname '
    'birthdate '
    'deathdate birthplace birthstate '
    'termstartdate '
    'termenddate '
    'party'
).split()  # define some field name

mc = MongoClient()  # get a Mongo client

try:
    mc.drop_database("presidents")  # delete 'presidents' database if it exists
except errors.PyMongoError as err:
    print(err)

db = mc["presidents"]  # create a new database named 'presidents'

coll = db.presidents  # get the collection from presidents db

with open('../DATA/presidents.txt') as presidents_in:  # open a data file
    for line in presidents_in:
        flds = line[:-1].split(':')
        kvpairs = zip(FIELD_NAMES, flds)
        record_dict = dict(kvpairs)
        coll.insert_one(record_dict)  # insert a record into collection

print(db.list_collection_names())  # get list of collections
print()

abe = coll.find_one({'termnumber': '16'})  # search collection for doc where termnumber
== 16
print(abe, '\n')

for field in FIELD_NAMES:
    print(f"{field.upper():15s} {abe[field]}")  # print all fields for one record

print('-' * 50)

for president in coll.find():  # loop through all records in collection
```


© 2024 JS Consulting *(Licensed to Trivera)*

```
    print(f"{president['firstname']:25s} {president['lastname']:30s}")
print('-' * 50)

rx_lastname = re.compile('^roo', re.IGNORECASE)
for president in coll.find({'lastname': rx_lastname}):  # find record using regular
expression
    print(f"{president['firstname']:25s} {president['lastname']:30s}")
print('-' * 50)

for president in coll.find({"birthstate": 'Virginia'}):  # find record searching multiple
fields
    print(f"{president['firstname']:25s} {president['lastname']:30s}")

print('-' * 50)
print("removing Millard Fillmore")
result = coll.delete_one({'lastname': 'Fillmore'})  # delete record
print(result)
result = coll.delete_one({'lastname': 'Roosevelt'})  # delete record
print(result)
print('-' * 50)


result = coll.delete_one({'lastname': 'Bush'})
print(dir(result))
print()

result = coll.count_documents({})  # get count of records
print(result)


for president in coll.find():  # loop through all records in collection
    print(f"{president['firstname']:25s} {president['lastname']:30s}")
print('-' * 50)

animals = db.animals

print(animals, '\n')

animals.insert_one({'name': 'wombat', 'country': 'Australia'})
animals.insert_one({'name': 'ocelot', 'country': 'Mexico'})
animals.insert_one({'name': 'honey badger', 'country': 'Iran'})

for doc in animals.find():
    print(doc['name'])
```

mongodb_example.py

```
William Howard            Taft
Woodrow                   Wilson
Warren Gamaliel           Harding
Calvin                    Coolidge
Herbert Clark             Hoover
Franklin Delano           Roosevelt
Harry S.                  Truman
Dwight David              Eisenhower
John Fitzgerald           Kennedy
Lyndon Baines             Johnson
Richard Milhous           Nixon
Gerald Rudolph            Ford
James Earl 'Jimmy'        Carter
Ronald Wilson             Reagan
William Jefferson 'Bill'  Clinton
George Walker             Bush
Barack Hussein            Obama
Donald John               Trump
Joseph Robinette          Biden
--------------------------------------------------
Collection(Database(MongoClient(host=['localhost:27017'], document_class=dict,
tz_aware=False, connect=True), 'presidents'), 'animals')

wombat
ocelot
honey badger
```


© 2024 JS Consulting *(Licensed to Trivera)*

Chapter 1 Exercises

Exercise 1-1 (president_sqlite.py)

For this exercise, you can use the SQLite3 database provided, or use your own DBMS. The mkpres.sql script is generic and should work with any DBMS to create and populate the presidents table. The SQLite3 database is named **presidents.db** and is located in the DATA folder of the student files.

The data has the following layout

Table 6. Layout of President Table

Field Name	Data Type	Null	Default
termnum	int(11)	YES	NULL
lastname	varchar(32)	YES	NULL
firstname	varchar(64)	YES	NULL
termstart	date	YES	NULL
termend	date	YES	NULL
birthplace	varchar(128)	YES	NULL
birthstate	varchar(32)	YES	NULL
birthdate	date	YES	NULL
deathdate	date	YES	NULL
party	varchar(32)	YES	NULL

Refactor the **president.py** module to get its data from this table, rather than from a file. Re-run your previous scripts that used president.py; now they should get their data from the database, rather than from the flat file.

If you created a president.py module as part of an earlier lab, use that. Otherwise, use the supplied president.py module in the top folder of the student files.

Exercise 1-2 (add_pres_sqlite.py)

Add the next president to the presidents database. Just make up the data — let's keep this non-political. Don't use any real-life people.

SQL syntax for adding a record is

```
INSERT INTO table ("COL1-NAME",..) VALUES ("VALUE1",...)
```

To do a parameterized insert (the right way!):

```
INSERT INTO table ("COL1-NAME",..) VALUES (%s,%s,...)  # MySQL
INSERT INTO table ("COL1-NAME",..) VALUES (?,?,...)    # SQLite
```

or whatever your database uses as placeholders

There are also MySQL versions of the answers.

© 2024 JS Consulting *(Licensed to Trivera)*

Chapter 2: Packaging

Objectives

- Create a pyproject.toml file
- Understand the types of wheels
- Generate an installable wheel
- Configure dependencies
- Configure executable scripts
- Distribute and deploy packages

© 2024 JS Consulting *(Licensed to Trivera)*

Packaging overview

- Bundling project for distribution
- Uses build tools
- Needs metadata
- Extremely flexible

Packaging a project for distribution does not have to be complex. However, the tools are very flexible, and the amount of configuration can be overwhelming at first.

It boils down to these steps:

Create a virtual environment

While not absolutely necessary, creating a virtual environment for your project makes life easier, especially when it comes to dependency management.

Create a project layout

The arrangement of files and subfolders in the project folder. This can be *src* (recommended) or *flat*.

Specify metadata

Using `pyproject.toml`, specify metadata for your project. This metadata tells the build tools how and where to build and package your project.

Build the project

Use the build tools to create a *wheel* file. This wheel file can be distributed to developers.

Install the wheel file

Anyone who wants to install your project can use **pip** to install the project from the wheel file you built.

Upload the project to PyPI (Optional)

To share a project with everyone, upload it to the **PyPI** online repository.

© 2024 JS Consulting *(Licensed to Trivera)*

Terminology

Here are some terms used in Python packaging. They will be explained in more detail in the following pages.

build backend

A module that does the actual creation of installable files (wheels). E.g., `setuptools` (>=61), `poetry-core`, `hatchling`, `pdm-backend`, `flit-core`.

build frontend

A user interface for a build backend. E.g., `pip`, `build`, `poetry`, `hatch`, `pdm`, `flit`

cookiecutter

A tool to generate the files and folders needed for a project.

dependency

A package needed by the current package

editable install

An installation that is really a link to the development folder, so changes to the code are reflected whenever the package or module is imported.

package

A collection of code (usually a folder) to be bundled into a reusable (installable) "artifact" AKA *wheel* file. [1]

PEP

Python Enhancement Proposal — a document that describes some aspect of Python. Similar to RFCs in the Internet world.

pip

The standard tool to install a Python package.

script

An executable Python script that is installed in the `scripts` (Windows) or `bin` (Mac/Linux) folder of your Python installation

toml

A file format similar to INI that is used for describing projects.

wheel

A file that contains everything needed to install a package

[1] Not to be confused with "package", which is a folder used to organize modules. (Well, maybe to be confused a little).

© 2024 JS Consulting *(Licensed to Trivera)*

Project layout

A typical project has several parts: source code, documentation, tests, and metadata. These can be laid out in different ways, but most people either do a *flat* layout or a *src* layout.

A *flat* layout has the code in the top level of the project, and a *src* layout has code in a separate folder named `src`.

In the long run, the *src* layout seems to be the most readable, and that makes it the best practice.

Metadata goes in the `pyproject.toml` file.

Unit tests go in a folder named `tests`. Documentation, using a tool such as **Sphinx**, goes in a folder named `docs`.

You can add any other files or folders necessary for your project.

Typical layout

```
temperature
├── README.md
├── docs
│   ├── Makefile
│   ├── make.bat
│   └── source
│       ├── _static
│       ├── _templates
│       ├── conf.py
│       └── index.rst
├── pyproject.toml
├── src
│   └── temperature.py
└── tests
    └── test_temperature.py
```

the name of the project folder can be anything, but is typically the name of of the module or package you are creating.

The layouts on the following pages are not the only possibilities. You can combine them in whatever way works for your project.

© 2024 JS Consulting *(Licensed to Trivera)*

Sample Project layouts

Module

```
MODULE_PROJECT
 ├──── README.md
 ├──── docs
 ├──── pyproject.toml
 ├──── src
 │     └──── mymodule.py
 └──── tests
```

Code in mymodule.py will run as follows:

```
# import module (in a script)
# __name__ set to "mymodule"
import mymodule
from mymodule import MyClass, myfunction

# execute module (from command line)
# __name__ set to "__main__"
python mymodule.py
python -m mymodule
```

Module with callable scripts

```
MODULE_SCRIPTS_PROJECT
 ├──── README.md
 ├──── docs
 ├──── pyproject.toml  # script names mapped to `mymodule:function`
 ├──── src
 │     └──── mymodule.py  # functions called by scripts
 └──── tests
```

Module can be run or imported normally as above, plus scripts defined in `pyproject.toml` can run directly from the command line

```
myscript
```

© 2024 JS Consulting *(Licensed to Trivera)*

Package

```
PACKAGE_PROJECT
├──── README.md
├──── docs
├──── pyproject.toml
├──── src
│    └──── mypackage
│        ├──── mymodule1.py
│        └──── mymodule2.py
└──── tests
```

```python
from mypackage import mymodule1
from mypackage.mymodule1 import MyClass, myfunction
```

Package with subpackages

```
PACKAGE_PROJECT
├──── README.md
├──── docs
├──── pyproject.toml
├──── src
│    └──── mypackage
│        └──── mysubpackage1
│            ├──── mymodule1.py
│            └──── mymodule2.py
│        └──── mysubpackage2
│            ├──── mymodule3.py
│            └──── mymodule4.py
└──── tests
```

```python
from mypackage.subpackage1 import mymodule1
from mypackage.subpackage2.mymodule3 import MyClass, myfunction
```


© 2024 JS Consulting *(Licensed to Trivera)*

Package with callable scripts

```
PACKAGE_SCRIPTS_PROJECT
├──── README.md
├──── docs
├──── pyproject.toml  # script names mapped to `mypackage:mymodule1:function` etc
├──── src
│    └──── mypackage
│        ├──── mymodule1.py  # functions called by scripts
│        └──── mymodule2.py  # functions called by scripts
└──── tests
```

```
myscriptname1
myscriptname2
```

Package callable with `python -m packagename`

```
PACKAGE_APP_PROJECT
├──── README.md
├──── docs
├──── pyproject.toml
├──── src
│    └──── mypackage
│        ├──── __main__.py  # module executed by  `python  -m packagename`
│        ├──── mymodule1.py    # code to support main module
│        └──── mymodule2.py    # code to support main module
└──── tests
```

```
python -m mypackage
```

© 2024 JS Consulting *(Licensed to Trivera)*

Packages with scripts

- Provide utility scripts
- Run from command line
- Installed in `.../scripts` or `.../bin`
- Add config to `pyproject.toml`

It is easy to add one or more command-line scripts to your project. These scripts are created in the `scripts` (Windows) or `bin` (other OS) folders of your Python installation. While they require Python to be installed, they are run like any other command.

The scripts are based on functions in the module. Since the scripts are run from the CLI, they are not called with normal parameters. Instead, the functions access `sys.argv` for arguments, like any standalone Python script.

```
def _c2f_cli():
    """
    CLI utility script
    Called from command line as 'c2f'
    """
    cel = float(sys.argv[1])
    return c2f(cel)


def _f2c_cli():
    """
    CLI utility script
    Called from command line as 'f2c'
    """
    fahr = float(sys.argv[1])
    return f2c(fahr)
```

To configure scripts, add a section like the following to `pyproject.toml`. The names on the left are the installed script names. The values on the right are the module name and the function name, separated by a colon.

```
[project.scripts]
c2f = 'temperature:_c2f_cli'
f2c = 'temperature:_f2c_cli'
```

See the project `temperature_scripts` in EXAMPLES for details.

© 2024 JS Consulting *(Licensed to Trivera)*

python -m *NAME*

- python -m module
  - executes entire module
- python -m package
  - executes module `{dunder}main{dunder}.py in package`
- includes `if {dunder}name{dunder} == "{dunder}main{dunder}"` section

The command `python -m __name__` is designed to execute a module or package without knowing its exact location. It uses the module search mechanism to find and load the module.

If `__name__` is a package, it executes the module named `{dunder}main{dunder}.py` in the top level of the package.

If `__name__` is a module, it executes the entire module.

In both cases, the code in the `if {dunder}name{dunder} == "{dunder}main{dunder}"` block (if any) is executed. If a module is imported, that code is not executed.

A list of standard modules that have a CLI via `python -m` is here: https://docs.python.org/3/library/cmdline.html

Cookiecutter

- Creates standard layout
- Developed for Django
- Very flexible

cookiecutter is a utility written by Audrey and Roy Greenfeld to make it easy to replicate a standard setup for Django. However, it can be used create a template for any type of project.

The `cookiecutter` command prompts you for information, then creates the project folder.

It uses a cookiecutter *template*, which is a folder, to create the new project. There are many templates on **github** to choose from, and you can easily create your own.

The script copies the template layout (all folders and files) to a new folder which is the "slug" (short name) of your project. It inserts your project name in the appropriate places.

There are two **cookiecutter** templates provided in the SETUP folder of the student files to generate the layouts on the previous pages:

- `cookiecutter-python-module`
- `cookiecutter-python-package`

The project layouts will be generated based on answers to the cookiecutter questions.

cookiecutter home page: https://github.com/audreyr/cookiecutter
cookiecutter docs: https://cookiecutter.readthedocs.io

Feel free to copy the cookiecutter templates and modify them for your own projects.

Another useful tool for generating Python projects is **PyScaffold**. Details at https://pyscaffold.org/en/stable/index.html.

© 2024 JS Consulting *(Licensed to Trivera)*

tree cookiecutter-python-module

```
/Users/jstrick/curr/courses/python/common/setup/cookiecutter-python-module
├──── cookiecutter.json
├──── hooks
│    ├──── post_gen_project.py
│    └──── pre_gen_project.py
└──── {{cookiecutter.module_slug}}
    ├──── README.{{cookiecutter.readme_format}}
    ├──── docs
    │    ├──── Makefile
    │    ├──── make.bat
    │    └──── source
    │        ├──── _static
    │        ├──── _templates
    │        ├──── conf.py
    │        └──── index.rst
    ├──── pyproject.toml
    ├──── src
    │    └──── {{cookiecutter.module_slug}}.py
    └──── tests
        ├──── __init__.py
        └──── test_{{cookiecutter.module_slug}}.py

8 directories, 12 files
```

cookiecutter-python-module/cookiecutter.json

```
{
    "module_name": "Module Name (can have spaces)",
    "module_slug": "{{ cookiecutter.module_name.lower().replace(' ', '').replace('-',
'_') }}",
    "module_description": "Short Description of the Module",
    "has_scripts": "n",
    "author_name": "Author Name",
    "author_email": "Author Email",
    "author_url": "Author URL",
    "copyright_year": "2023",
    "readme_format": ["md", "rst"]
}
```

Defining project metadata

- Create `pyproject.toml`
- Use `build` to build the package

The **modern** way to package a Python project is using the `pyproject.toml` config file. The specifications that support this are specified in **PEP 518** and **PEP 621**.

The **TOML** format is similar to `.ini` files, but adds some features.

The first part of the file is always required. It tells the `build` program what tools to use.

```
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"
```

Put all the project metadata that the build system will need to package and install your project after the `[build-system]` section.

```
[project]
name = "wombatfun"
version = "1.0.0"
authors = [
    { name="Author Name", email="jstrickler@gmail.com" },
]
description = "Short Description of the Package"
readme = "README.rst"
requires-python = ">=3.0"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

dependencies = [
    'requests[security] < 3',
]
```


© 2024 JS Consulting *(Licensed to Trivera)*

TOML value types arrays are similar to Python `list` and TOML tables (including inline) are similar to `dict`.

The rest of the file after the `[build-system]` section is not needed if you are also using `setup.cfg` and `setup.py`. However, best practice is to *not* use those legacy files, as all the data needed for building the package, installing it, and uploading it to **PyPI** can be contained in `pyproject.toml`, and can then be used by nearly any build *backend*.

Editable installs

- Use `pip install -e __package__`
- Puts a link in library folder
- Allows testing as though module is installed

When using a `src` (or other name) folder for your codebase and `tests` for your test scripts, the tests need to find your package. While you could put the path to the `src` folder in PYTHONPATH, the best practice is to do an *editable install*.

This is an install that uses the path to your development folder. It achieves this by using a virtual environment. Then you can run your tests after making changes to your code, without having to reinstall the package.

From the top level folder of the project, type the following (you do not have to build the distribution for this step).

```
pip install -e .
```

Now the project is installed and is available to import or run like any other installed module.

© 2024 JS Consulting *(Licensed to Trivera)*

Running unit tests

- Use editable install
- Just use `pytest` or `pytest -v`

To run the tests that you have created in the `tests` folder, just run `pytest` or `pytest -v` (verbose) in the top level folder of the project. Because the project was installed with an editable install, tests can import the module or package normally.

Example

```
$ pytest -v
============================================================ test session starts
============================================================
platform darwin -- Python 3.9.17, pytest-7.1.2, pluggy-1.0.0 -- /Users/jstrick/opt/miniconda3/bin/python
cachedir: .pytest_cache
PyQt5 5.15.7 -- Qt runtime 5.15.2 -- Qt compiled 5.15.2
hypothesis profile 'default' ->
database=DirectoryBasedExampleDatabase('/Users/jstrick/curr/courses/python/common/examples/temperature/.hypothesis/examples')
rootdir: /Users/jstrick/curr/courses/python/common/examples/temperature
plugins: anyio-3.6.1, qt-4.1.0, remotedata-0.3.3, assert-utils-0.3.1, lambda-2.1.0, astropy-header-0.2.1, fixture-order-
0.1.4, common-subject-1.0.6, mock-3.8.2, typeguard-2.13.3, astropy-0.10.0, filter-subpackage-0.1.1, hypothesis-6.54.3,
openfiles-0.5.0, django-4.5.2, doctestplus-0.12.0, cov-3.0.0, arraydiff-0.5.0
collected 8 items

tests/test_temperature.py::test_c2f[100-212] PASSED
[ 12%]
tests/test_temperature.py::test_c2f[0-32] PASSED
[ 25%]
tests/test_temperature.py::test_c2f[37-98.6] PASSED
[ 37%]
tests/test_temperature.py::test_c2f[-40--40] PASSED
[ 50%]
tests/test_temperature.py::test_f2c[212-100] PASSED
[ 62%]
tests/test_temperature.py::test_f2c[32-0] PASSED
[ 75%]
tests/test_temperature.py::test_f2c[98.6-37] PASSED
[ 87%]
tests/test_temperature.py::test_f2c[-40--40] PASSED
[100%]

============================================================ 8 passed in 0.20s
============================================================
```

Wheels

- 3 kinds of wheels
  - Universal wheels (pure Python; python 2 *and* 3 compatible
  - Pure Python wheels (pure Python; Python 2 *or* 3 compatible
  - Platform wheels (Platform-specific; binary)

A wheel is prebuilt distribution. Wheels can be installed with pip.

A *Universal wheel* is a pure Python package (no extensions) that can be installed on either Python 2 or Python 3. It has to have been carefully written that way.

A *Pure Python wheel* is a pure Python package that is specific to one version of Python (either 2 or 3). It can only be installed by a matching version of pip.

A *Platform wheel* is a package that has extensions, and thus is platform-specific.

Build systems automatically create the correct wheel type.

© 2024 JS Consulting *(Licensed to Trivera)*

Building distributions

- `python -m build`
- Creates `dist` folder
- Binary distribution
  - `package-version.whl`
- Source distribution
  - `package-version.tar.gz`

To build the project, use

```
python -m build
```

This will create the wheel file (binary distribution) and a gzipped tar file (source distribution) in a folder named `dist`.

© 2024 JS Consulting *(Licensed to Trivera)*

python -m build

```
* Creating virtualenv isolated environment...
* Installing packages in isolated environment... (setuptools>=61.0)
* Getting build dependencies for sdist...
running egg_info
writing src/temperature.egg-info/PKG-INFO
writing dependency_links to src/temperature.egg-info/dependency_links.txt
writing top-level names to src/temperature.egg-info/top_level.txt
reading manifest file 'src/temperature.egg-info/SOURCES.txt'
writing manifest file 'src/temperature.egg-info/SOURCES.txt'
* Building sdist...
```

about 70 lines of output

```
running install_scripts
creating build/bdist.macosx-10.9-universal2/wheel/temperature-1.0.0.dist-info/WHEEL
creating '/Users/jstrick/curr/courses/python/common/examples/temperature/dist/.tmp-
a03ni8q2/temperature-1.0.0-py3-none-any.whl' and adding 'build/bdist.macosx-10.9-
universal2/wheel' to it
adding 'temperature.py'
adding 'temperature-1.0.0.dist-info/METADATA'
adding 'temperature-1.0.0.dist-info/WHEEL'
adding 'temperature-1.0.0.dist-info/top_level.txt'
adding 'temperature-1.0.0.dist-info/RECORD'
removing build/bdist.macosx-10.9-universal2/wheel
Successfully built temperature-1.0.0.tar.gz and temperature-1.0.0-py3-none-any.whl
```


© 2024 JS Consulting *(Licensed to Trivera)*

Installing a package

- Use `pip`
  - many options
  - can install just for user

A wheel makes installing packages simple. You can just use

```
pip install __package__.whl
```

This will install the package in the standard location for the current version of Python.

If you do not have permission to install modules in the standard location, you can do a user install, which installs modules under your home folder.

```
pip install --user __package__.whl
```

For more information

Python Packaging User Guide

https://packaging.python.org/en/latest/

Distributing Python Modules

https://docs.python.org/3/distributing/index.html

setuptools Quickstart

https://setuptools.pypa.io/en/latest/userguide/quickstart.html

Thoughts on the Python packaging ecosystem

https://pradyunsg.me/blog/2023/01/21/thoughts-on-python-packaging/

THE BASICS OF PYTHON PACKAGING IN EARLY 2023

https://drivendata.co/blog/python-packaging-2023

Structuring Your Project (from The Hitchhiker's Guide to Python)

https://docs.python-guide.org/writing/structure/

© 2024 JS Consulting *(Licensed to Trivera)*

Chapter 2 Exercises

Exercise 2-1 (carddeck/*)

Step 1

Create a distributable module named `carddeck` from the `carddeck.py` and `card.py` modules in the root folder of the student files.

HINT: To do it the easy way, use the `cookiecutter-python-module` template. Add the two source files to the `src` folder. (remove any existing sample Python scripts in `src`).

+ To do it the "hard" way, create the project layout by hand, create the `pyproject.toml` file, etc.

Step 2

Build a distribution (wheel file).

Step 3

Install the wheel file with `pip`. (The cookiecutter template automatically does an editable install)

Step 4

Then import the new module and create an instance of the `CardDeck` class. Shuffle the cards, and deal out all 52 cards.

© 2024 JS Consulting *(Licensed to Trivera)*

Chapter 3: IPython and JupyterLab

Objectives

- Learn the basics of IPython
- Apply magics
- List and replay commands
- Run external commands
- Create profiles
- Use Jupyter notebooks in JupyterLab

© 2024 JS Consulting *(Licensed to Trivera)*

About IPython

- Enhanced python interpreter
- Great for "playing around" with Python
- Saves running entire script
- Not intended for application development
- Embedded in Jupyter notebooks

IPython is an enhanced interpreter for Python. It provides a large number of "creature comforts" for the user, such as name completion and improved help features.

It is very handy for quickly trying out Python features or for casual data analysis.

Command line interface

When started from a command line, starts a read-execute-print loop (REPL), also known as an interactive interpreter.

Ipython uses different colors for variables, functions, strings, comments, and so forth.

Jupyter notebook

IPython also provides a *kernel* for embedding IPython in Jupyter notebooks. A Jupyter notebook is a web-based interface consisting of *cells*, which contain code or documentation. Jupyter notebooks can run code from many different languages in addition to Python.

JupyterLab

This is the web-based interface to manage multiple Jupyter notebooks, as well as images, terminal prompts, and other resources.

© 2024 JS Consulting *(Licensed to Trivera)*

Starting IPython

- Type `ipython` at the command line
- Huge number of options

To get started with IPython

- Type `ipython` at the command line

OR

- Double-click the IPython icon from Windows explorer.

IPython works like the normal interactive Python interpreter, but with many more features.

There is a huge number of options. To see them all, invoke IPython with the --help-all option:

```
ipython --help-all
```

Use the `--colors=NoColor` option to turn off syntax highlighting and other colorized features.

© 2024 JS Consulting *(Licensed to Trivera)*

Getting Help

- ? basic help
- %quickref quick reference
- help standard Python help
- thing? help on thing

IPython provides help in several ways.

Typing `?` at the prompt will display an introduction to IPython and a feature overview.

For a quick reference card, type `%quickref`.

To start Python's normal help system, type `help`.

For help on any Python object, type `object?` or `?object`. This is similar to saying `help("_object_")` in the default interpreter, but is "smarter".

For more help, add a second question mark. This does not work for all objects, however, and sometimes it displays the source code of the module containing the object definition.

IPython features

- Name completion (variables, modules, methods, folders, files, etc.)
- Enhanced help system
- Autoindent
- Syntax highlighting
- *Magic* commands for controlling IPython itself
- Easy access to shell commands
- Dynamic introspection (dir() on steroids)
- Search namespaces with wildcards
- Commands are numbered (and persistent) for recall
- Aliasing system for interpreter commands
- Simplified (and lightweight) persistence
- Session logging (can be saved as scripts)
- Detailed tracebacks when errors occur
- Session restoring (playback log to specific state)
- Flexible configuration system
- Easy access to Python debugger
- Simple profiling
- Interactive parallel computing (if supported by hardware)
- Background execution in separate thread
- Auto-parentheses (*sin 3* becomes *sin(3)*
- Auto-quoting (*,foo a b* becomes *foo("a","b")*

© 2024 JS Consulting *(Licensed to Trivera)*

Tab Completion

- Press `Tab` to complete
  - keywords
  - variables
  - modules
  - methods and attributes
  - parameters to functions
  - file and directory names

Pressing `Tab` will invoke **tab completion**, AKA **autocomplete**. If there is only one possible completion, it will be expanded. If there is more than one completion that will match, IPython will display a list of possible completions.

Autocomplete works on keywords, functions, classes, methods, and object attributes, as well as paths from your file system.

Magic Commands

- Start with `%` (line magic) or `%%` (cell magic)
- Simplify common tasks
- Use `%lsmagic` to list all magic commands

One of the enhancements in IPython is the set of "magic" commands. These are meta-commands (macros) that help you manipulate the IPython environment.

Normal magics apply to a single line. Cell magics apply to a cell (a group of lines).

For instance, `%history` will list previous commands.

Type `lsmagic` for a list of all magics

If the magic command is not the same as a name in your Python code, you can leave off the leading `%` or `%%`.

© 2024 JS Consulting *(Licensed to Trivera)*

Loading and running Python scripts

- Run script in current session
- `%run` runs script
- `%load` loads script source code into IPython

IPython provides two magics to run scripts — one to run directly, and one to run indirectly. Both will run the script in the context of the current IPython session.

Running scripts directly

The `%run` magic just takes a script name, and runs it. This method does not allow IPython magics to be executed as part of a script.

```
In [1]: %run ../EXAMPLES/my_vars.py
```

```
In [2]:  user_name
Out[2]: 'Susan'
```

```
In [3]: snake
Out[3]: 'Eastern Racer'
```

Running scripts indirectly

The `%load` magic takes a script name, and loads the contents of the script so it can then be executed.

This method allows IPython magics to be executed as part of a script.

This also useful if you want to run a script, but edit the script before it is run.

```
In [4]: %load imports.py
```

```
In [5]: # %load imports.py
...: import numpy as np
...: import scipy as sp
...: import pandas as pd
...: import matplotlib.pyplot as plt
...: import matplotlib as mpl
...: %matplotlib inline
...: import seaborn as sns
...: sns.set()
...:
...:
```


© 2024 JS Consulting *(Licensed to Trivera)*

External commands

- Precede command with !
- Can assign output to variable

Any OS command can be run by starting it with a !.

The resulting output is returned as a list of strings (stripping the trailing \n characters). The result can be assigned to a variable.

Windows

```
In [3]: !dir DATA\*.csv
 Volume in drive Z is Shared Folders
 Volume Serial Number is 0000-0064

 Directory of Z:\Desktop\py2forsci\DATA

02/20/2014  01:53 PM             5,511 airport_boardings.csv
02/20/2014  01:53 PM             2,182 energy_use_quad.csv
02/20/2014  01:53 PM             4,993 parasite_data.csv
               3 File(s)         12,686 bytes
               0 Dir(s)  352,625,324,032 bytes free

In [4]:
```

Non-Windows (Linux, OS X, etc)

```
In [2]: !ls -l DATA/*.csv
-rwxr-xr-x  1 jstrick  staff  5511 Jan 27 19:44 DATA/airport_boardings.csv
-rwxr-xr-x  1 jstrick  staff  2182 Jan 27 19:44 DATA/energy_use_quad.csv
-rwxr-xr-x  1 jstrick  staff  4642 Jan 27 19:44 DATA/parasite_data.csv

In [3]:
```

© 2024 JS Consulting *(Licensed to Trivera)*

Using history

- use %history magic
- `history` *list commands*
- `history -n` *list commands with numbers*
- `hist` *shortcut for "history"*

The `%history` magic will list previous commands. Use `-n` to list commands with their numbers.

Selecting commands

You can select a single command or a range of commands separated by a dash.

```
history 5
history 6-10
```

Use `~N/`, where N is 1 or greater, to select commands from previous sessions.

```
history ~2/3  third command in second previous session
```

To select more than one range or individual command, separate them by spaces.

```
history 4-6 9 12-16
```

The same syntax can be used with `%edit`, `%rerun`, `%recall`, `%macro`, `%save` and `%pastebin`.

© 2024 JS Consulting *(Licensed to Trivera)*

Recalling commands

The `%recall` magic will recall a previous command by number. It will leave the cursor at the end of the command so you can edit it.

```
recall 12
recall 4-7
```

Rerunning commands

`%rerun` will re-run a previous command without waiting for you to press `Enter`.

Saving sessions

- Save commands to Python script
- Specify one or more commands
- Use `%save` magic

It is easy to save a command, a range of commands, or any combination of commands to a Python script using the `%save` magic.

The syntax is

```
%save filename selected commands
```

.py will be appended to the filename.

© 2024 JS Consulting *(Licensed to Trivera)*

Using Pastebin

- Online "clipboard"
- Use `%pastebin` command

Pastebin is a free online service that accepts pasted text and provides a link to access the text. It can be used to share code snippets with other programmers.

The `%pastebin` magic will paste selected commands to **Pastebin** and return a link that can be used to retrieve them. The link provided will expire in 7 days.

Use `-d` to specify a title for the pasted code.

```
link = %pastebin -d "my code" 10-15  write commands 10 through 15 to Pastebin and get
link
```

Add ".txt" to the link to retrieve the plain text that you pasted. This can be done with `requests`:

```
import requests
link = %pastebin -d "my code" 10-15
pasted_text = requests.get(link + '.txt').text
```

Benchmarking

- Use %timeit

IPython has a handy magic for benchmarking.

```
In [1]: color_values = { 'red':66, 'green':85, 'blue':77 }

In [2]: %timeit red_value = color_values['red']
10000000 loops, best of 3: 54.5 ns per loop

In [3]: %timeit red_value = color_values.get('red')
10000000 loops, best of 3: 115 ns per loop
```

%timeit will benchmark whatever code comes after it on the same line. %%timeit will benchmark contents of a notebook cell

© 2024 JS Consulting *(Licensed to Trivera)*

Profiles

- Stored in `.ipython` folder in home folder
- Contains profiles and other configuration
- Can have multiple profiles
- `ipython profile` subcommands
  - `list`
  - `create`
  - `locate`

IPython supports *profiles* for storing custom configurations and startup scripts. There is a default profile, and any number of custom profiles can be created.

Each profile is a separate subfolder under the `.ipython` folder in a users's home folder.

Creating profiles

Use `ipython profile create _name_` to create a new named profile. If `_name_` is omitted, this will create the default profile (if it does not already exist)

Listing profiles

Use `ipython profile list` to list all profiles

Finding profiles

`ipython profile locate _name_` will display the path to the specified profile. As with creating, omitting the name shows the path to the default profile.

© 2024 JS Consulting *(Licensed to Trivera)*

```
.ipython
├──── cython
│     └──── Users
│           └──── mikedev
├──── extensions
├──── nbextensions
├──── profile_default
│     ├──── db
│     ├──── ipython_config.py
│     ├──── ipython_kernel_config.py
│     ├──── log
│     ├──── pid
│     ├──── security
│     ├──── startup
│     │     ├──── 00_imports.py
│     │     └──── 10_macros.py
│     └──── static
│           └──── custom
└──── profile_science
      ├──── db
      ├──── ipython_config.py
      ├──── ipython_kernel_config.py
      ├──── log
      ├──── pid
      ├──── security
      └──── startup
            └──── 00_imports.py
```


© 2024 JS Consulting *(Licensed to Trivera)*

Configuration

IPython has many configuration settings. You can change these settings by creating or editing the script named `ipython_config.py` in a profile folder.

Within this script you can use the global config object, named `c`.

For instance, the line

```
c.InteractiveShellApp.pylab_import_all = False
```

Will change how the `%pylab` magic works. When true, it will populate the user namespace with the contents of `numpy` and `pylab` as though you had entered `from numpy import *` and `from pylab import *`

When false, it will just import `numpy` as `np` and `pylab` as `pylab`.

Link to all IPython options:

https://ipython.readthedocs.io/en/stable/config/options/index.html

There are two groups of settings. "Terminal Python Options" refers to using iPython interactively. "IPython kernal options" refers to using iPython in a Jupyter Notebook.

When you create a profile, this config script is created with some commented code to get you started.

Startup

Startup scripts allow you to execute frequently used code, especially imports, when starting IPython.

Startup scripts go in the `startup` folder of the profile folder. All Python scripts in this folder will be executed, in lexicographical (sorted) order.

The scripts will be executed in the context of the IPython session, so all imports, variables, functions, classes, and other definitions will be available in the session.

It is convenient to prefix the startup scripts with "00", "10", "20", and so forth, to set the order of execution.

© 2024 JS Consulting *(Licensed to Trivera)*

Jupyter notebooks

- Extension of IPython
- Puts the interpreter in a web browser
- Code is grouped into "cells"
- Cells can be edited, repeated, etc.

In 2015, the developers of IPython pulled the notebook feature out of IPython to make a separate product called Jupyter. It is still invoked via the `jupyter notebook` command, and now supports over 130 language kernels in addition to Python.

A Jupyter notebook is a journal-like python interpreter that lives in a browser window. Code is grouped into cells, which can contain multiple statements. Cells can be edited, repeated, rearranged, and otherwise manipulated.

A notebook (i.e, a set of cells, can be saved, and reopened). Notebooks can be shared among members of a team via the notebook server which is built into Jupyter.

JupyterLab

In 2018, the Jupyter developers released **JupyterLab**, which is a web-based application to manage Jupyter notebooks and other files in one place.

JupyterLab Demo

Start JupyterLab and follow along with a demo of JupyterLab and Jupyter notebooks as directed by the instructor

Open an Anaconda prompt (on Windows) or a terminal window (on Mac or Linux) and navigate to the top folder of the student files, then

```
cd NOTEBOOKS
jupyter-lab
```

© 2024 JS Consulting *(Licensed to Trivera)*

For more information

- https://ipython.org/
- https://jupyter.org/
- https://ipythonbook.com

© 2024 JS Consulting *(Licensed to Trivera)*

Appendix A: Field Guide to Python Expressions

Table 7. Python Expressions

Expression	Meaning
`a, b, c` `(a, b, c)`	tuple
`[a, b, c]`	list
`{a, b, c}`	set
`{a:r, b:s, c:t}`	dictionary
`[expr for var in iterable if condition]`	list comprehension
`(expr for var in iterable if condition)`	generator expression
`{expr for var in iterable if condition}`	set comprehension
`{key:value for var in iterable if condition}`	list comprehension
`a, b, c = iterable`	iterable unpacking
`a, *b, c = iterable`	extended iterable unpacking
`a if b else c`	conditional expression
`lambda VAR ···: VALUE`	lambda function

© 2024 JS Consulting *(Licensed to Trivera)*

Index

© 2024 JS Consulting *(Licensed to Trivera)*