

# Python Advanced



# Introduction

# Program



Python Recap  
Python Advanced Concepts



File Formats  
Pickle  
XML  
JSON



Database Access  
ORM  
SqlAlchemy



Web Frameworks  
Pyramid  
Django  
Flask

# Topics

- Pythonic / Idiomatic Python
- OO - associations
- virtual environments
- type hinting
- logging
- zipfile
- pylint
- docstrings
- pydoc
- unit testing
- File I/O
- Concurrency
- GUI
- configparser
- git & github
- functools
- itertools
- pathlib
- shutil
- dateutil, calendar, arrow
- database access
- web applications
- executables

# Python Basics

- Built-in numeric types: bool, int, float, complex
- Operators: arithmetic, comparative, boolean
- Strings: concatenation, string methods, formating, unicode
- Flow statements: while, for, if-elif-else
- Functions: arguments, default values, variadic arguments, keyword arguments, return value, lambda
- Built-in functions
- Reading from and writing to files, context manager
- Lists: indexing, slicing, modification, unpacking, comprehension
- Dicts: creation, modification, iterating
- Object Orientation: classes, objects, methods and attributes

# Advanced Python

# Pythonic / Idiomatic Python

- PEP 8 – Style Guide for Python Code <https://peps.python.org/pep-0008/>

# pip

Usage:

```
pip <command> [options]
```

Commands:

install	Install packages.
download	Download packages.
uninstall	Uninstall packages.
freeze	Output installed packages in requirements format.
inspect	Inspect the python environment.
list	List installed packages.
show	Show information about installed packages.
check	Verify installed packages have compatible dependencies.
config	Manage local and global configuration.
search	Search PyPI for packages.
cache	Inspect and manage pip's wheel cache.
index	Inspect information available from package indexes.
wheel	Build wheels from your requirements.
hash	Compute hashes of package archives.
completion	A helper command used for command completion.
debug	Show information useful for debugging.
help	Show help for commands.

```
$ pip list > requirements.txt  
$ pip install -r requirements.txt
```

# Project structure

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

```
"""Script to ..."""

import sys

def main():
    """Main entry point for the script."""
    pass

if __name__ == '__main__':
    sys.exit(main())
```

```
helloworld/
    -- bin/
    -- docs/
        -- hello.md
        -- world.md
    -- helloworld/
        -- __init__.py
        -- runner.py
        -- hello/
            -- __init__.py
            -- hello.py
            -- helpers.py
        -- world/
            -- __init__.py
            -- helpers.py
            -- world.py
    -- data/
        -- input.csv
    -- tests/
        . -- hello
            -- helpers_tests.py
            -- hello_tests.py
        -- world/
            -- helpers_tests.py
            -- world_tests.py
    -- .gitignore
    -- LICENSE
    -- README.md
    -- requirements.txt
    -- setup.py
    -- venv
```

# Import

- import in a script
- vs
- import in a package

```
try:  
    from .encryption import Encryption  
  
except ImportError:  
    from encryption import Encryption
```

# Virtual Environments

Python virtual environments aim to provide a lightweight, isolated Python environment

- venv

```
$ python -m venv venv

$ venv\Scripts\activate          # windows
$ source venv/bin/activate       # macos

$ deactivate
```

- virtualenv
  - The package is a superset of venv, which allows you to do everything that you can do using venv, and more

# Conda Environments

- Conda Package and Environment Manager
- If you're working in the data science space and with Python alongside other data science projects, then conda is an excellent choice that works across platforms and languages.
- conda

```
$ conda create -n new_environment  
$ conda activate new_environment  
$ conda deactivate
```

# Type Hinting

- Type hinting is a formal solution to statically indicate the type of a value within your Python code. It was specified in **PEP 484** and introduced in **Python 3.5**.
- Python's type hints provide you with optional static typing to leverage the best of both static and dynamic typing.

```
def headline(text: str, align: bool = True) -> str:
    if align:
        return f"{text.title()}\n{'-' * len(text)}"
    else:
        return f" {text.title()} ".center(50, "o")
```

# Variadic arguments

- \*args
- \*\*kwargs

# Decorators

- A decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.
- Functions are first-class citizens:
  - functions can be passed around and used as arguments.
  - functions can be defined inside other functions. Such functions are called inner functions.
  - functions can be returned as return values.

```
def my_decorator(f):
    def wrapper(*args, **kwargs):
        # do something
        return_value = f(*args, **kwargs)
        # do something
        return return_value
    return wrapper
```

# Functools

- partial

# functools

- higher order functions
- functional programming: map, filter, sorted, reduce
- closure
- partial
- decorators

# Closures

- A closure is a function defined within an other function
- The inner function retains the variables during definition

```
def say():
    greeting = 'Hello'

    def display():
        nonlocal greeting
        print(greeting)

    return display
```

# Classes

- Decorators
  - @classmethod
  - @staticmethod
  - @property
  - @<property>.setter
- Inheritance
  - super()
  - multiple inheritance, MRO

# Functional Programming

- Pure functions
- No side-effects
  - no printing
  - no globals
- Built-in functions
  - map
  - filter
  - sorted
  - reduce
- Lambda



# Generator expression

- Generator expression      (  $x^{**2}$  for  $x$  in range(100) )

```
# list comprehension
doubles = [2 * n for n in range(50)]

# generator expression
doubles = (2 * n for n in range(50))

# same as the list comprehension above
doubles = list(2 * n for n in range(50))
```

# Generator functions

- The keyword **yield** specifies a generator function
- When the **yield** keyword is hit the function returns a result
- The next time the function is called the function continues where it left off

```
import random

def random_order1(numbers):
    random.shuffle(numbers)
    for number in numbers:
        yield number

def random_order2(numbers):
    random.shuffle(numbers)
    yield from numbers
```

# Logging facility for Python

- import `logging`
- Setup with `basicConfig`
- Logging Levels: DEBUG, INFO, WARNING, ERROR, CRITICAL

```
import logging

logging.basicConfig(
    filename = None, # or to a file 'example.log',
    level = logging.ERROR,
    format = '%(asctime)s.%(msecs)03d - %(message)s',
    datefmt = '%Y-%m-%dT%H:%M:%S')

logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
logging.error('Watch out!')
logging.critical('ERROR!!!!!!')
```

# Pylint

- Pylint is a **static code analyser** for Python
- Pylint analyses your code without actually running it. It checks for errors, enforces a coding standard, looks for code smells, and can make suggestions about how the code could be refactored.
- It can also be integrated in most editors or IDEs.
- `pylint my_script.py`

# Docstrings

- Best practices
  - All modules, classes, methods, and functions, including the `__init__` constructor in packages should have docstrings.
  - Descriptions are capitalized and have end-of-sentence punctuation.
  - Always use """"Triple double quotes."""" around docstrings.
  - Docstrings are not followed by a blank line.
- Conventions for multi-line docstrings
  - [Sphinx style](#)
  - [Google Format](#)
  - [NumPy Format](#)

```
"""[Summary]

:param [ParamName]: [ParamDescription], defaults to [DefaultParamVal]
:type [ParamName]: [ParamType](, optional)
...
:raises [ErrorType]: [ErrorDescription]
...
:return: [ReturnDescription]
:rtype: [ReturnType]
"""
```

# pydoc

- Documentation generator and online help system
- The built-in function [help\(\)](#) invokes the online help system in the interactive interpreter, which uses [pydoc](#) to generate its documentation as text on the console.
- In order to find objects and their documentation, [pydoc](#) imports the module(s) to be documented. Therefore, any code on module level will be executed on that occasion. Use an `if __name__ == '__main__':` guard to only execute code when a file is invoked as a script and not just imported.

```
$ pydoc sys
```

# unittest - Unit testing framework

- There is also an **assert** statement

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

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

**assertEqual(a, b)**  
**assertNotEqual(a, b)**  
**assertTrue(x)**  
**assertFalse(x)**  
**assertIs(a, b)**  
**assertIsNot(a, b)**  
**assertIsNone(x)**  
**assertIsNotNone(x)**  
**assertIn(a, b)**  
**assertNotIn(a, b)**  
**assertIsInstance(a, b)**  
**assertNotIsInstance(a, b)**



# File I/O

# json - JavaScript Object Notation

- JSON encoder and decoder

```
json.dump(object, file)
json.dumps(object)
json.load(file)
json.loads(string)
```

```
import json

s = json.dumps([1,2,3,{'4': 5, '6': 7}])

with open('bestand.json', 'w') as f:
    json.dump([1,2,3,{'4': 5, '6': 7}], f)

json.loads('[1,2,3,{"4":5,"6":7}]')
```

# pickle - Python object serialization

- A **shelf** is a persistent, dictionary-like object that stores any arbitrary Python that can be pickled.

```
pickle.dump(object, file)
pickle.dumps(object)
pickle.load(file)
pickle.loads(string)
```

```
import pickle

class User:
    def saveToPickle(self):
        with open('user.pickle','wb') as f:
            pickle.dump(self, f)
    def loadFromPickle(self):
        with open('user.pickle','rb') as f:
            self.__dict__.update(pickle.load(f).__dict__)
    @classmethod
    def createFromPickle(cls):
        with open('user.pickle','rb') as f:
            return pickle.load(f)
```

# xml

- The **xml.etree.ElementTree** module implements a simple and efficient API for parsing and creating XML data.
- This module provides limited support for **XPath** expressions for locating elements in a tree. [https://www.w3schools.com/xml/xpath\\_intro.asp](https://www.w3schools.com/xml/xpath_intro.asp)
- ElementTree provides a simple way to build XML documents and write them to files.

```
import xml.etree.ElementTree as ET

tree = ET.parse('data.xml')
root = tree.getroot()

print(root.attrib)

for element in root.findall('//name'):
    print(element.text)
```

# GUI

- tkinter
- pyqt

# configparser

- This module provides the ConfigParser class which implements a basic configuration language which provides a structure similar to what's found in Microsoft Windows INI files.

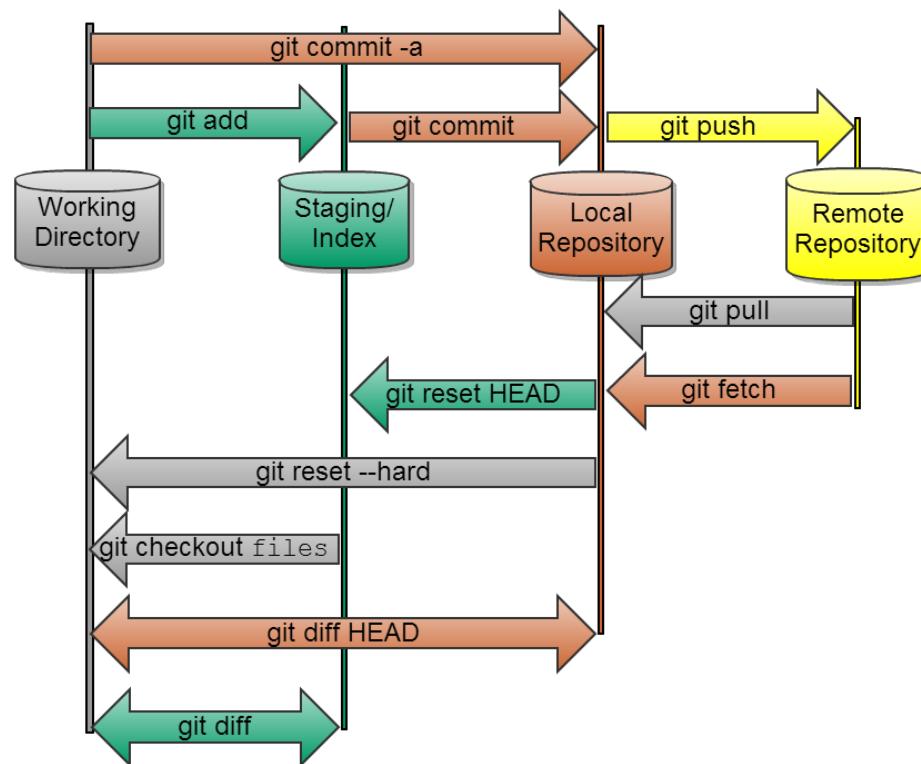
```
import configparser
config = configparser.ConfigParser()
config['DEFAULT'] = {'ServerAliveInterval': '45',
                     'Compression': 'yes',
                     'CompressionLevel': '9'}
config['bitbucket.org'] = {}
config['bitbucket.org']['User'] = 'hg'
config['topsecret.server.com'] = {}
topsecret = config['topsecret.server.com']
topsecret['Port'] = '50022' # mutates the parser
topsecret['ForwardX11'] = 'no' # same here
config['DEFAULT']['ForwardX11'] = 'yes'
with open('example.ini', 'w') as configfile:
    config.write(configfile)
```

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes
[bitbucket.org]
User = hg
[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

# git & github

- Git is a free and open source distributed version control system

## Git Workflow & Commands



```

$ git config --global user.name "My Name"
$ git config --global user.email "myname@gmail.com"

$ git init Demo
$ git clone URL

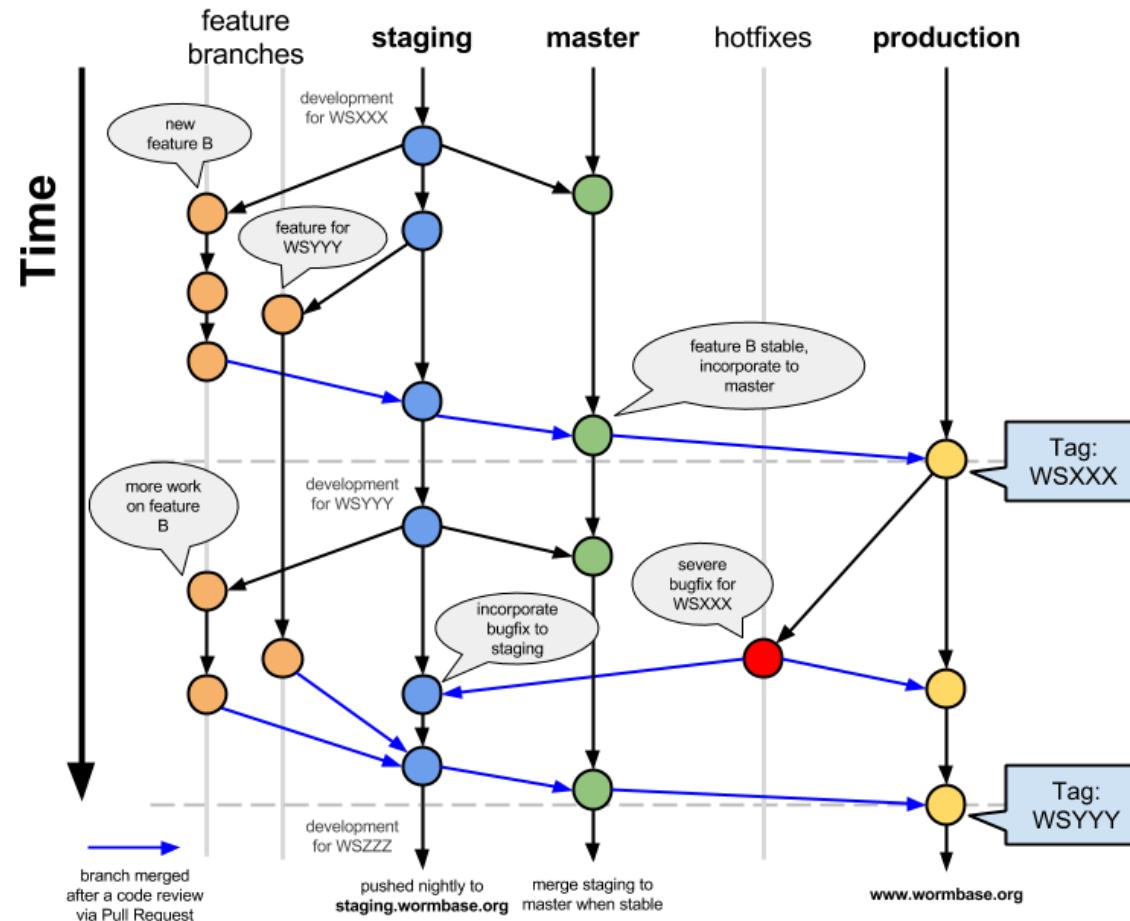
$ git add Filename
$ git commit -m "Commit Message"

$ git status

$ git push origin master
$ git push --all
$ git pull URL
  
```

# GIT Branches & Workflow

**WormBase Web Development Workflow with Git**





# itertools



# pathlib



# shutil

# datetime - Basic date and time types

- The datetime module supplies classes for manipulating dates and times.

```
class datetime.date  
class datetime.time  
class datetime.datetime  
class datetime.timedelta  
class datetime.tzinfo  
class datetime.timezone
```

**strftime**

**strptime**

```
from datetime import date  
  
d = date(2020, 2, 28)  
  
print(d.strftime('%Y-%m-%d'))
```

# dateutil, calendar, arrow

- The *dateutil* module provides powerful extensions to the standard *datetime* module, available in Python.

# database access

- PEP 249 - Database API Specification 2.0

# sqlite3

- DB-API 2.0 interface for SQLite databases

```
import sqlite3

conn = sqlite3.connect('example.db')
c = conn.cursor()

c.execute("""CREATE TABLE stocks
            (date text, trans text, symbol text, qty real, price real)""")

c.execute("""INSERT INTO stocks
            VALUES ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)""")

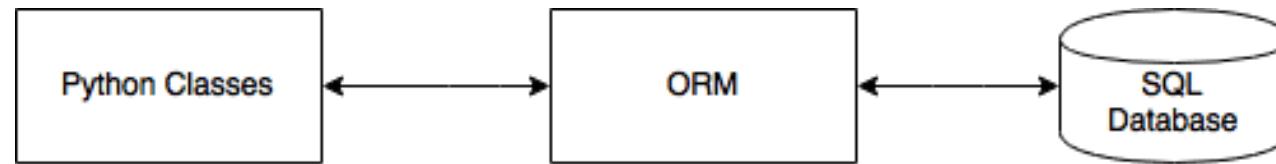
conn.commit()

for row in c.execute('SELECT * FROM stocks ORDER BY price'):
    print(row)

conn.close()
```

# SqlAlchemy

- The Python SQL Toolkit and Object Relational Mapper



**SQLAlchemy**

# web applications

- django
- flask
- pyramid
- fastapi
  
- dash
- streamlit

# django

MVC Framework

Models

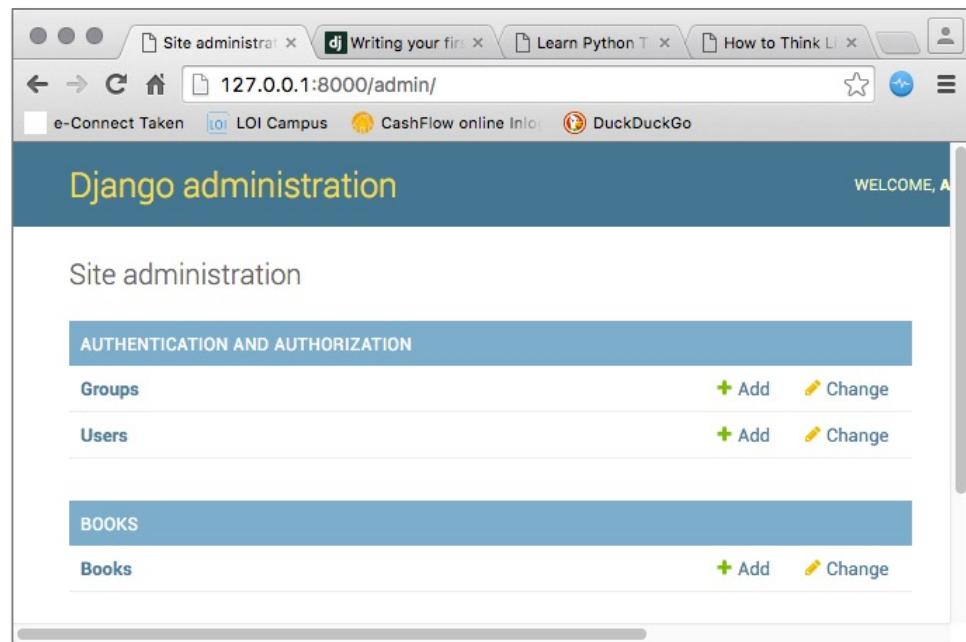
Object-Relational Mapping

URL Mapping

Views

HTML Templates

Command line bootstrap:



```
$ mkdir django-demo
$ cd django-demo
$ virtualenv -p python3.5 venv
$ . venv/bin/activate
$ pip install django
$ django-admin
$ django-admin startproject demo
$ python manage.py createsuperuser
$ python manage.py startapp books
$ python manage.py makemigrations
$ python manage.py migrate
$ python manage.py runserver
```

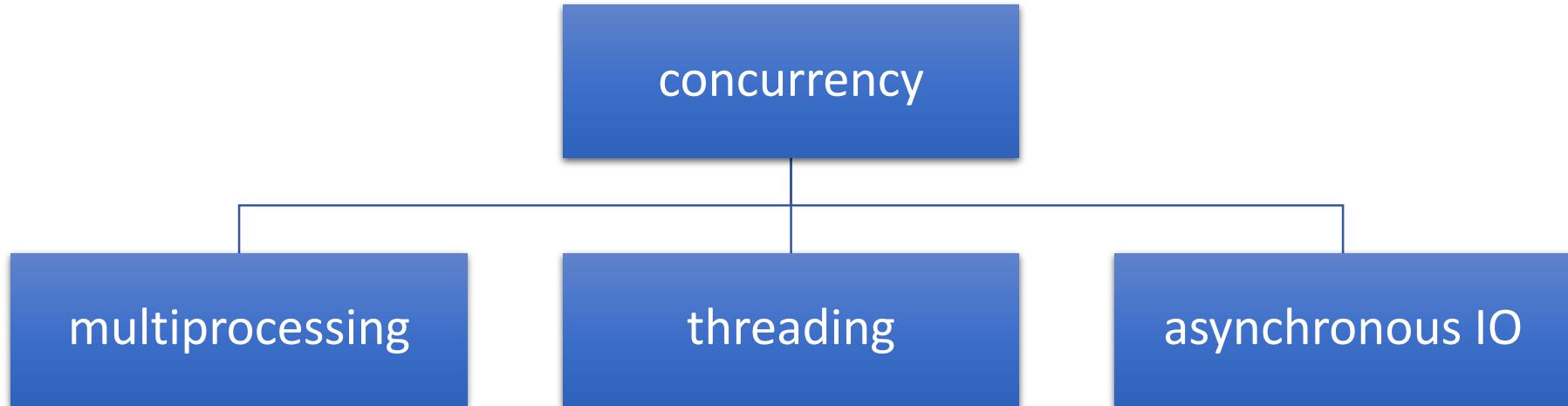
# executables

- shebang        `#!/usr/bin/env python3`
- py2exe        windows
- py2app        macos

# Concurrency

- Threading
- Asynchronous
- Multiprocessing

# Concurrency



Python Standard Library:

- **multiprocessing** package
- **threading** package
- **asyncio** package and **async/await** keywords (introduced in Python 3.4)

# Comparison

	Multiprocessing	Threading	Asynchronous IO
<b>Package</b>	<code>multiprocessing</code>	<code>threading</code>	<code>asyncio</code>
<b>Class</b>	<code>Process</code>	<code>Thread</code>	<code>Coroutine</code>
<b>Python</b>	<code>Process</code>	<code>Thread</code>	Keywords <code>async</code> , <code>await</code>
<b>Data sharing</b>	Message	Shared data	
<b>Usage</b>	CPU intensive	IO intensive	IO intensive

# Proces versus Thread

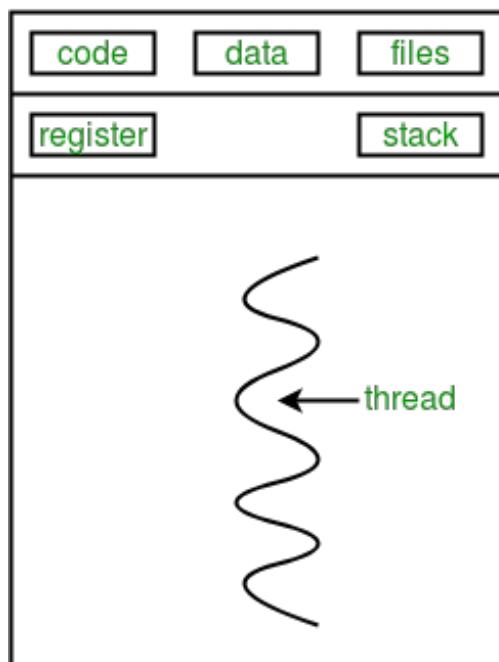
- True parallelism in Python is achieved by creating multiple processes, each having a Python interpreter with its own separate GIL.

<b>Process</b>	<b>Thread</b>
processes run in separate memory (process isolation)	threads share memory
uses more memory	uses less memory
children can become zombies	no zombies possible
more overhead	less overhead
slower to create and destroy	faster to create and destroy
easier to code and debug	can become harder to code and debug

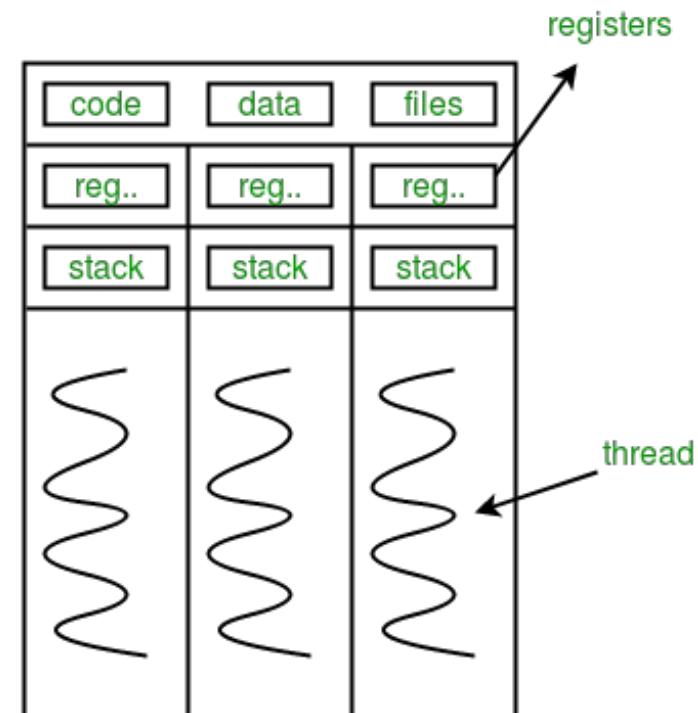
# Python GIL

- A **global interpreter lock (GIL)** is a mechanism used in Python interpreter to synchronize the execution of threads so that only one native thread can execute at a time, even if run on a multi-core processor.
- The C extensions, such as numpy, can manually release the GIL to speed up computations. Also, the GIL released before potentially blocking I/O operations.
- Note that both Jython and IronPython do not have the GIL.

# Threading



single-threaded process



multithreaded process

# threading - Thread-based parallelism

- Thread
  - start
  - run
  - join
  - name
- active\_count
- current\_thread
- main\_thread

```
import time
from threading import Thread

def myfunc(i):
    print "sleeping 5 sec from thread %d" % i
    time.sleep(5)
    print "finished sleeping from thread %d" % i

for i in range(10):
    t = Thread(target=myfunc, args=(i,))
    t.start()
```

# asyncio

- At the heart of async IO are **coroutines**. A coroutine is a specialized version of a Python generator function.

```
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"__file__ executed in {elapsed:.2f} seconds.")
```

# multiprocessing

- The multiprocessing library is based on spawning Processes.
- A process starts a fresh Python interpreter thereby side-stepping the Global Interpreter Lock
- The multiprocessing module allows the programmer to fully leverage multiple processors on a given machine.

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

# Implementations

- [CPython](#) reference implementation
- [IronPython](#) (Python running on .NET)
- [Jython](#) (Python running on the Java Virtual Machine)
- [PyPy](#) (A fast python implementation with a JIT compiler)
- [Stackless Python](#) (Branch of CPython supporting microthreads)
- [MicroPython](#) (Python running on micro controllers)







# Project

# Python DBI API 2.0

- connect
- cursor
- execute



# RDBMS

- |                        |                        |
|------------------------|------------------------|
| • SQLite               | sqlite3                |
| • Microsoft SQL Server | pyodbc                 |
| • Oracle               |                        |
| • PostgreSQL           | psycopg2               |
| • MySQL                | mysql-connector-python |

# NoSQL

NoSQL databases are more flexible than relational databases. In these types of databases, the data storage structure is designed and optimized for specific requirements. There are four main types for NoSQL libraries:

- Document-oriented
- Key-value pair
- Column-oriented
- Graph

Python NoSQL:

- MongoDB
- Redis
- Cassandra
- Neo4j

# SQL

- CREATE TABLE
- ALTER TABLE
- DROP TABLE
- INSERT
- SELECT
- UPDATE
- DELETE
- CRUD
  - Create
  - Read
  - Update
  - Delete
- LIMIT clause
- FROM clause
- JOIN clause
- WHERE clause
- ORDER BY clause
- GROUP BY clause



# SQLite



# PostgreSQL

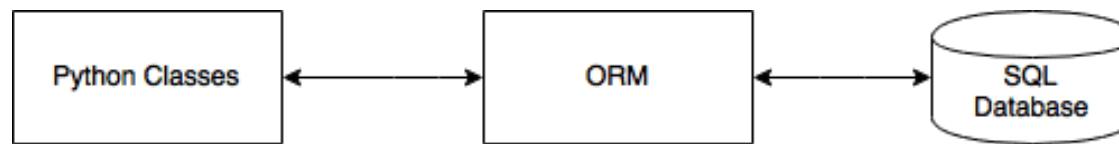
# Links

- <https://pythonspot.com/python-database/>
- <https://www.geeksforgeeks.org/python-database-tutorial/>
- <https://www.tutorialspoint.com/postgresql/index.htm>

# Links

- <https://www.geeksforgeeks.org/python-database-tutorial/>

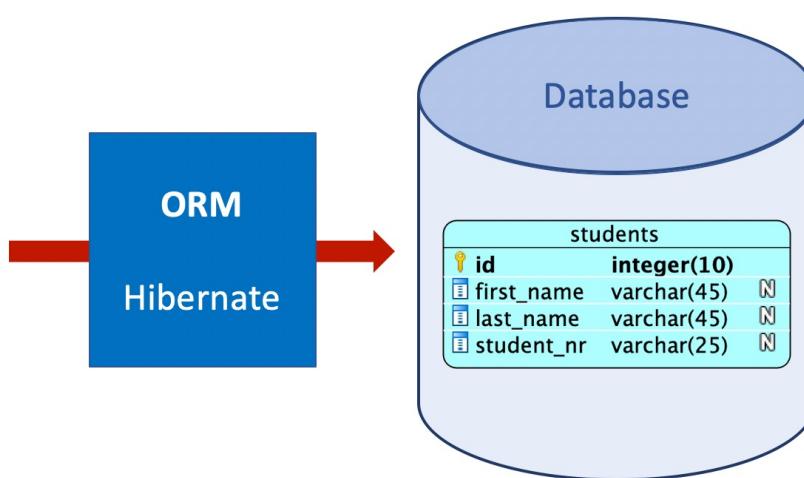
# SqlAlchemy



Student Class in Object Model

```
@Entity  
@Table(name = "students")  
public class Student {  
  
    @Id  
    @GeneratedValue  
    Long id;  
  
    @Column(name = "first_name")  
    String firstName;  
  
    @Column(name = "last_name")  
    String lastName;  
  
    @Column(name = "student_nr")  
    String studentNr;  
  
}
```

Student Table in Relational Model



# requests – HTTP for Humans

- **Requests** is an elegant and simple HTTP library for Python, built for human beings.

```
import requests

url = "http://api.openweathermap.org/data/2.5/weather"
url += "?appid=d1526a9039658a6f76950cff21823aff"
url += "&units=metric"
url += "&mode=json"
url += "&q>New York"

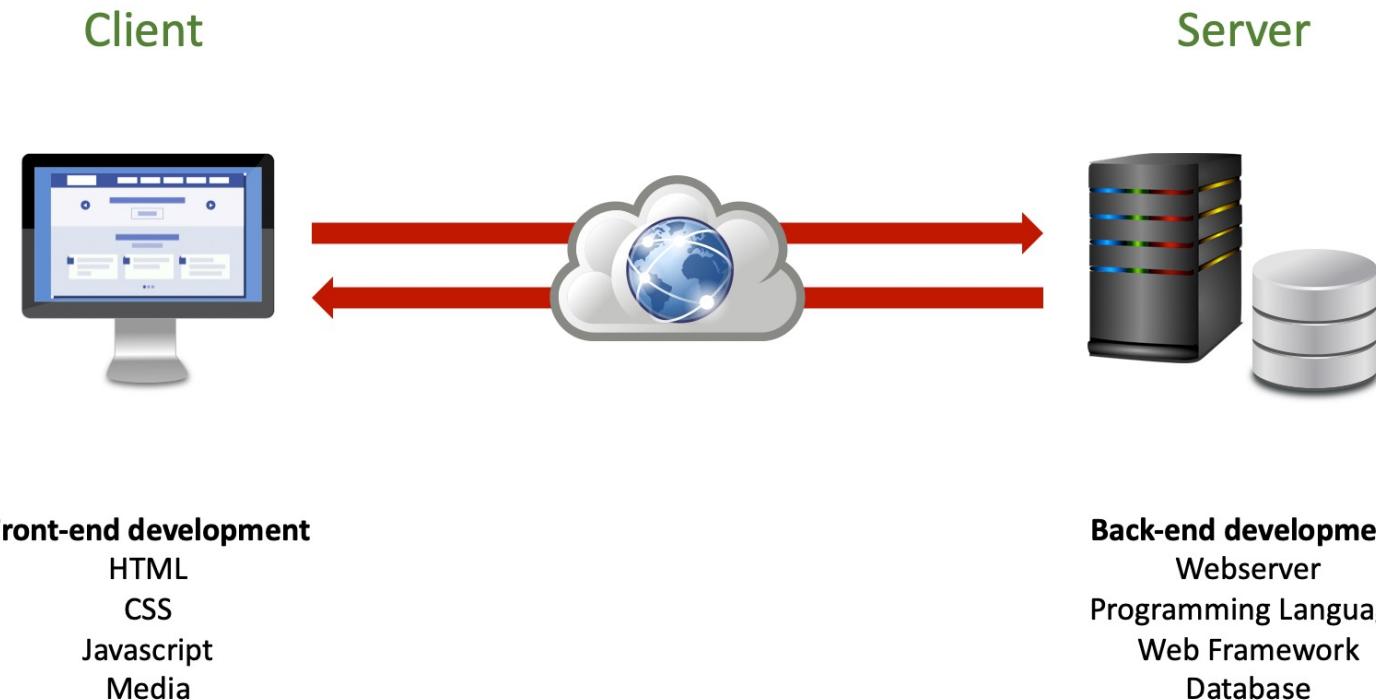
response = requests.get(url)
if (response.status_code == 200):
    body = response.text
    decoded = response.json()
    temperature = decoded['main']['temp']
else:
    print("Error for city %s" % (city))
```

# Web Frameworks

- Django
- Flask
- Pyramid
- Bottle
- Sierra
- FastAPI

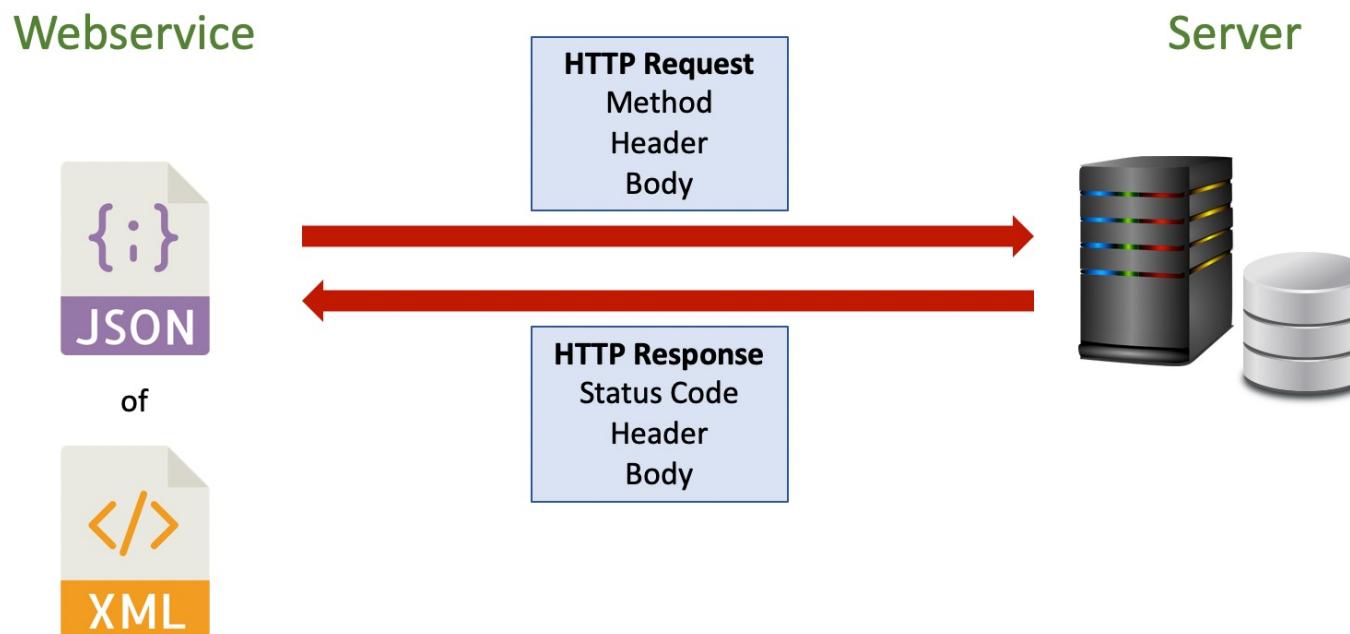
# Web Framework

- Routing
- Database ORM
- Templating



# RESTful

- Client – Server
- Http Verbs – Get, Post, Put, Delete => CRUD
- Payload – Data - JSON



# django

MVC Framework

Models

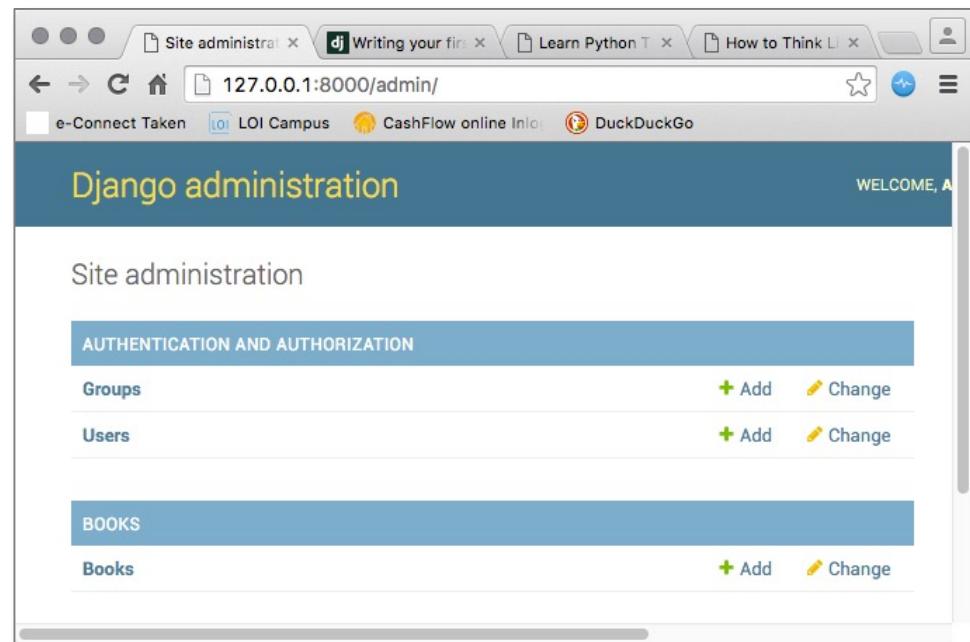
Object-Relational Mapping

URL Mapping

Views

HTML Templates

Command line bootstrap:



```
$ mkdir django-demo
$ cd django-demo
$ virtualenv -p python3.5 venv
$ . venv/bin/activate
$ pip install django
$ django-admin
$ django-admin startproject demo
$ python manage.py createsuperuser
$ python manage.py startapp books
$ python manage.py makemigrations
$ python manage.py migrate
$ python manage.py runserver
```

# Flask

- **Flask** is a micro [web framework](#) written in [Python](#). It is classified as a [microframework](#) because it does not require particular tools or libraries.<sup>[2]</sup> It has no [database](#) abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions. However, Flask supports extensions that can add application features as if they were implemented in Flask itself. Extensions exist for [object-relational mappers](#), form validation, upload handling, various open authentication technologies and several common framework related tools.





# FastAPI

# GUI Frameworks

- **TkInter** - The traditional Python user interface toolkit.
- **PyQt** - Bindings for the cross-platform Qt framework.
- **PySide** - PySide is a newer binding to the Qt toolkit
- **wxPython** - a cross-platform GUI toolkit that is built around wxWidgets
- **Win32Api** - native window dialogs
- **PyMsgBox**
  
- **Dash**
- **Jupyter Widgets**

# tkinter

- The tkinter package (“Tk interface”) is the standard Python interface to the Tk GUI toolkit.
- There are also Standard Dialogs

```
import tkinter as tk
import tkMessageBox

top = tk.Tk()

def hello():
    tkMessageBox.showinfo("Say Hello", "Hello World")

btn1 = tk.Button(top, text = "Say Hello", command = hello)
btn1.pack()

top.mainloop()
```



# tkinter

- widgets
- Layout Managers
  - pack
  - grid
  - place

Tkinter Frame  
Tkinter Button  
Tkinter Label  
Tkinter Entry  
Tkinter Radio Button  
Tkinter Check Button  
Tkinter Combobox  
Tkinter Menu Button  
Tkinter Toplevel  
Tkinter Scrollbar  
Tkinter Menu  
Tkinter List Box  
Tkinter Text Widget  
Tkinter Canvas  
Tkinter Scale  
Tkinter LabelFrame  
Tkinter SpinBox  
Tkinter Color Chooser

# Distributions

- py2exe
- py2app