



# **Software Architecture Using Python**

## **Participant Guide**



#### Copyright

This subject matter contained herein is covered by a  
copyright owned by: Copyright © 2023 Robert Gance, LLC

This document contains information that may be proprietary. The contents of this document may not be duplicated by any means without the written permission of TEKsystems.

TEKsystems, Inc. is an Allegis Group, Inc. company. Certain names, products, and services listed in this document are trademarks, registered trademarks, or service marks of their respective companies.

All rights reserved

7437 Race Road  
Hanover, MD 21076

COURSE CODE IN1811 / 11.14.2023  
Phase 4 Accessibility

©2023 Robert Gance, LLC

ALL RIGHTS RESERVED

This course covers Software Architecture Using Python

---

No part of this manual may be copied, photocopied, or reproduced in any form or by any means without permission in writing from the author—Robert Gance, LLC, all other trademarks, service marks, products or services are trademarks or registered trademarks of their respective holders.

This course and all materials supplied to the student are designed to familiarize the student with the operation of the software programs.

THERE ARE NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, MADE WITH RESPECT TO THESE MATERIALS OR ANY OTHER INFORMATION PROVIDED TO THE STUDENT. ANY SIMILARITIES BETWEEN FICTITIOUS COMPANIES, THEIR DOMAIN NAMES, OR PERSONS WITH REAL COMPANIES OR PERSONS IS PURELY COINCIDENTAL AND IS NOT INTENDED TO PROMOTE, ENDORSE, OR REFER TO SUCH EXISTING COMPANIES OR PERSONS.

This version updated: 11/14/2023

# *Notes*

---

# *Chapters at a Glance*

Chapter 1	Software Architecture Overview	14
Chapter 2	Domain Modeling	47
Chapter 3	Service Layer	73
Chapter 4	High-Level Architecture Patterns	104
Chapter 5	Event-Driven Architectures	120
Chapter 6	Package Management and Deployment	138
	Course Summary	145

# *Notes*

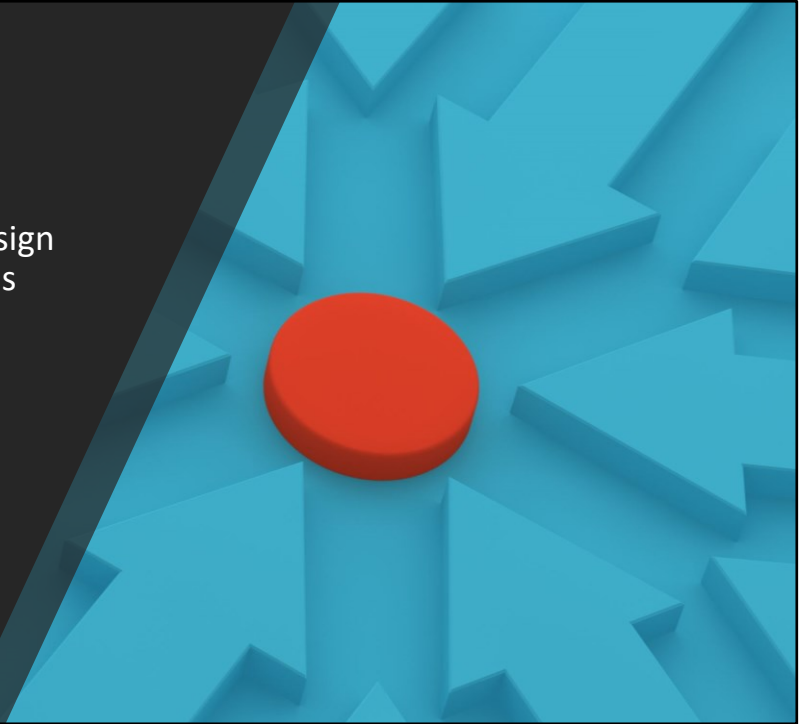
---



# Software Architecture Using Python

# Objectives

- Explore fundamental design and architecture patterns
- Utilize key patterns and principles to construct Python-based software solutions





# Day 1 Agenda

- Software Architecture Overview
  - Python Language Idioms
  - SOLID Principles
- Domain Modeling
- Repository

## Day 2 Agenda

- Business Services
- Event-Driven Architectures

## Introductions

Name (prefer to be called)



What you work on



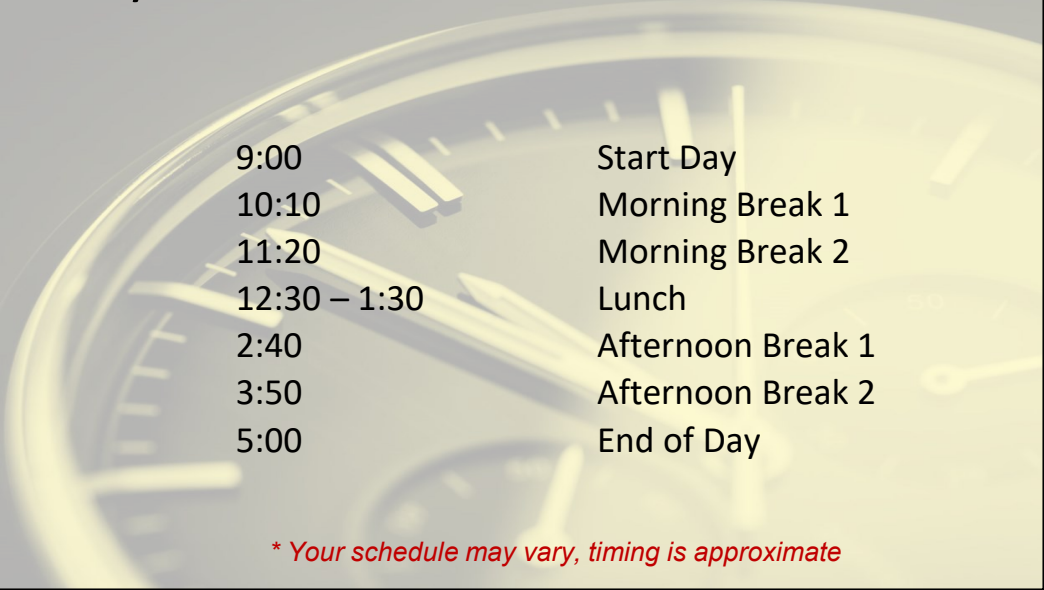
Background / experience with Python



Reason for attending



## Typical Daily Schedule



9:00	Start Day
10:10	Morning Break 1
11:20	Morning Break 2
12:30 – 1:30	Lunch
2:40	Afternoon Break 1
3:50	Afternoon Break 2
5:00	End of Day

*\* Your schedule may vary, timing is approximate*



Ask  
Questions



# Software Architecture Using Python Overview

# Overview

---

- Architecture vs. Design Patterns
- Python *Programming* Idioms
- Object-oriented Python
- SOLID Principles

# What is Software Architecture?

- **Software architecture** deals with the overall **structure** of the system
- Architecture patterns are used to
  - Break large, complex systems into manageable pieces
  - Create components (modeled after the company) that communicate with each other
  - Limit design choices that may promote a chaotic system
  - Simplify testing

Software architecture is common in enterprise application development due to the large, complex nature of systems created. Often, applications are built with many employees across distributed locations. This makes it important to use component architectures so that parts of the system can be worked on by different groups. Usually, enterprise systems have many subtasks to perform making it difficult to grasp the whole solution in its entirety. Component architectures help break systems into smaller manageable pieces.

Architecture deals with the system-level design structure. How do the components fit together and communicate with each other? What are the layers within the system? Security issues, performance factors, maintainability, scalability, reliability, and much more are all part of the architectural considerations.

At the design level, the components themselves are examined. Which lower-level patterns will comprise the subsystem? In other words, how will the components be constructed?



### Common Software Architectures

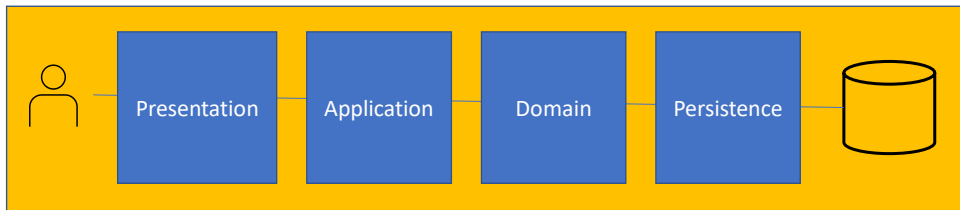
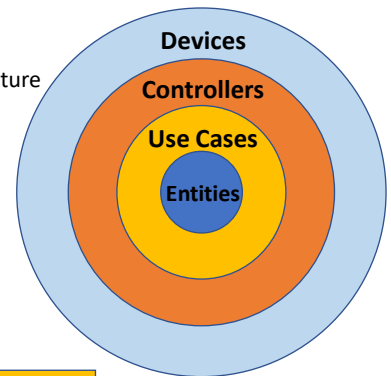
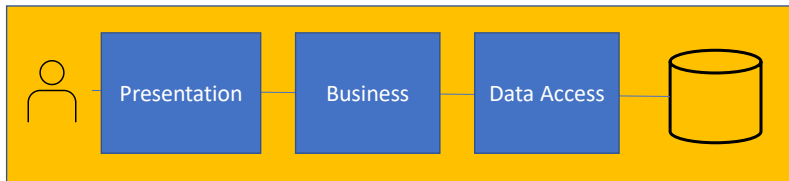
Numerous architectural patterns exist  
The patterns chosen are often dictated by the  
type of application being created

- Layered Architecture
  - N-tier (3/4 - tier)
  - Onion/Clean
  - Hexagonal
- Client-server
- MVC
- Event-Bus
- Microservices
- Master-slave

Other overall system architectures include broker, peer-to-peer, pipe-filter, interpreter, blackboard, and many others. The architectural pattern chosen is often dictated by the style of application being created.

## Layered Architecture (and Variations)

Clean Architecture



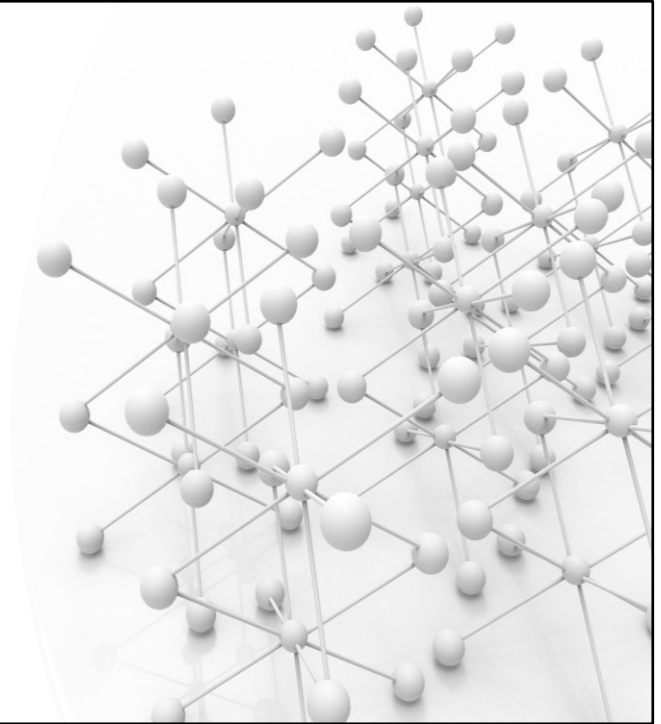
There are numerous variations of the layered architecture. Aside from some small differences, all of these variations are somewhat similar often using different terms for the same concepts. They each boil down to the domain layer being central or core to the entire system. The domain layer contains the domain model, which are the classes that model the business rules.

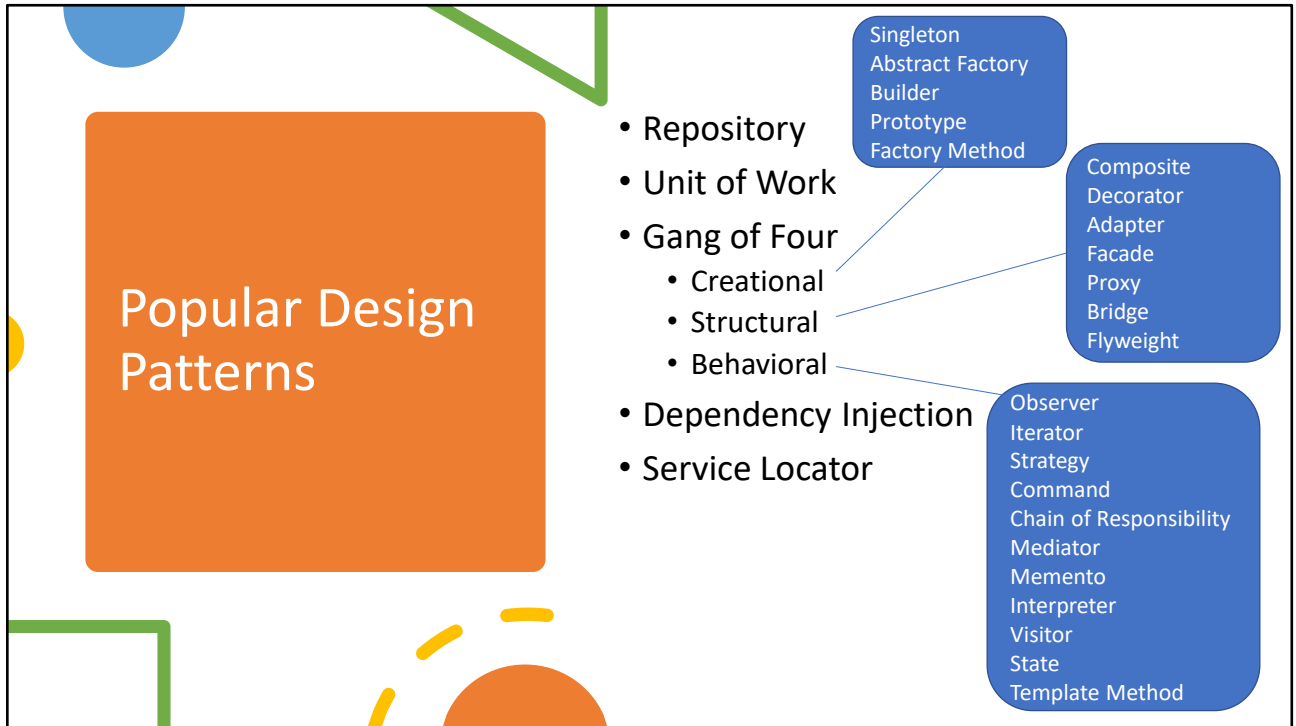
The business rules are the methods within these classes that communicate with the other domain objects. For example, the way in which an Order entity (aggregate) communicates with the Billing entity (aggregate) represents the business rules. It will be defined by the methods present in these objects and how they are invoked by the other domain objects.

# Importance of Design Patterns

---

- Design patterns provide structure *within a subsystem*
- *Provide proven solutions to recurring problems in software*
- They provide a means for communication among developers





There are dozens and dozens of well-known, proven design patterns. These patterns are usually found at the subsystem level. They typically help with solving a single problem within an overall architecture.

The *Gang of Four* patterns are twenty-three patterns that often form the basis for other patterns or are used directly within a subsystem. They are the result of the work from four different software designers.

## Your Turn! - Task 1-1

- Environment Setup Overview:

1. Launch PyCharm
2. Create a virtual environment
3. Install any needed packages
4. Configure the interpreter
5. Test it out!

Specific setup instructions  
can be found in the back of  
your student manual

Before going any further, we need to ensure our environments are properly configured.

## Python and Design Patterns

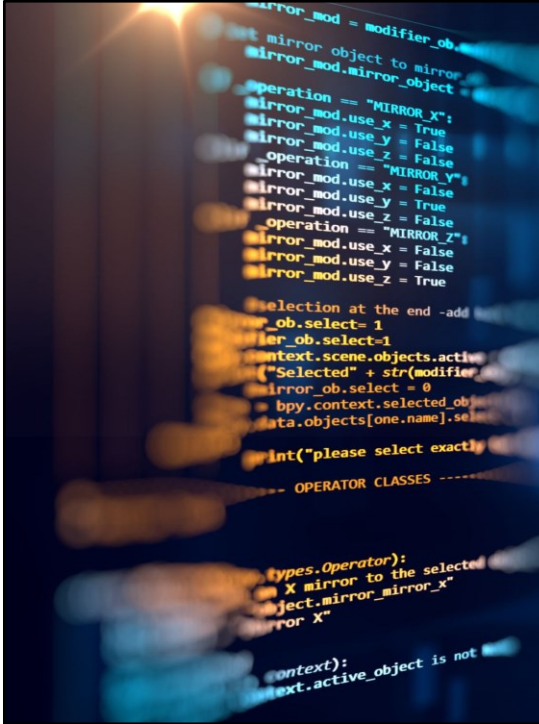
Some believe that design patterns in Python are an *anti-pattern*

Python's dynamic nature is such that adding extra abstractions might be over-engineering

Python has *duck typing*, *monkey patching*, and *dynamic creation capabilities* to avoid needing many of the patterns



Despite the belief by many that Python doesn't need design patterns, as systems get more complex, more structure and flexibility becomes needed.



## Programming Language Idioms

Python is different than other programming languages when it comes to coding style

It follows its own rules when coding and designing

- PEP-8 and language idioms
- Typing
- Boolean logic
- Iterating
- Object-oriented techniques

Often, Python is used as a secondary language. Many people who develop software solutions find Python useful, but it isn't their day-to-day tool. As a result, they often bring approaches and techniques, or idioms, from other programming languages to their Python solutions. It should be noted that Python has its own approaches and techniques to follow. Writing code from those languages may not follow preferred Python approaches. Often, you hear the word "Pythonic" which simply means idiomatic, or the preferred way within this programming language.

## PEP-8

Proper spacing for type hints

```
filepath: str = 'sample.dat'
```

Put different module imports on separate lines

```
from pathlib import Path
import sys
from typing import List, Tuple
```

Underscores between words in variables and function names

```
def load_data(filepath: str):
    pass
```

4-space indentation

```
def load_data(filepath: str,
               header: bool):
    pass
```

4-space rule doesn't apply on continuation lines

```
def load_data(filepath: str) -> List[Tuple]:
    results = []
    ...
    return results
```

Imports from within a module are okay on the same line

```
filepath = '../resources/celebrity_100.csv'
data = load_data(filepath)
print(f'{len(data)} names found.')
```

2 blank lines before/after function definitions

<https://peps.python.org/pep-0008/>

ch01\_overview/pep8.py

The most well-known PEP (Python Enhancement Proposal) is PEP 8. It defines ways in which Python code should be laid out from variable naming conventions to spacing to import formatting.



# Python Idioms

- Favor Readability

```
z = 10
if z is not None:
    print(z)

if not z is None:
    print(z)
```

← *Prefer this*

← *Over this*

```
fn = 'image.jpg'
if fn.endswith('jpg'):
    print(fn)

if fn[: -3] == 'jpg':
    print(fn)
```

← *Prefer this*

← *Over this*

ch01\_overview/idioms.py

There are multiple ways that code can be written in Python. The term "Pythonic" refers to how readable a solution is. Does it favor readability, and does it follow a common, preferred syntax used in Python? If so, the code is likely written idiomatically.

## Python Idioms for Booleans

- Use `==` when comparing values

```
x = 0x2a
y = 6**2 + 6
print(x == y)  ← Prefer this
print(x is y)  ← Over this
```

- Use *is* when comparing to `None`, `False`, or `True`

```
def get_value():
    data = None
    val = random.randint(1, 10)
    if val % 2:
        data = val
    return data

Prefer this → print(get_value() is None)
Over this → print(get_value() == None)
```

- Test Booleans directly

Prefer this →

```
x = True
if x:
    print(x)
```

```
if x == True:
    print(x)
```

← Over this

ch01\_overview/idioms.py

The general rule of thumb is to use `==` (double equals) for most comparisons except when comparing directly to `True`, `False`, or `None`. The reason for this is that you most likely are wanting to compare values, not memory addresses, so `==` is most appropriate. Also, `True`, `False`, and `None` are singletons (they only exist once in memory). Any two variables with similar values point to the same memory address.

## Python Idioms for Iterating (1 of 3)

```
with open('../resources/celebrity_100.csv', encoding='utf-8') as f:
    for count, line in enumerate(f):
        print(count, line)
```

← Prefer this

```
count = 0
with open('../resources/celebrity_100.csv', encoding='utf-8') as f:
    for line in f:
        print(count, line)
        count += 1
```

← Over this

```
max_lines_to_read = 5
with open('../resources/celebrity_100.csv', encoding='utf-8') as f:
    for line in itertools.islice(f, max_lines_to_read):
        print(line.strip())
```

← Prefer this

```
count = 0
with open('../resources/celebrity_100.csv', encoding='utf-8') as f:
    for line in f:
        print(line.strip())
        count += 1
        if count >= max_lines_to_read:
            break
```

← Over this

ch01\_overview/idioms.py

Whenever possible, avoid counting externally to the for-loop. In the first example, `enumerate()` can replace the use of a counting variable. In the second example, `islice()` can read an exact number of lines from the file without having to count them in a variable. It's not always possible to get rid of the counting variable, but for most situations there is a more "Pythonic" way to do it.

## Python Idioms for Iterating (2 of 3)

- Favor readability in expressions, use "don't care" values ( \_ )

```
with open('../resources/celebrity_100.csv', encoding='utf-8') as f:
    for name, pay, _, _ in itertools.islice(csv.reader(f), 3):
        print(name, pay)
```

← Prefer this

```
with open('../resources/celebrity_100.csv', encoding='utf-8') as f:
    for items in itertools.islice(csv.reader(f), 3):
        print(items[0], items[1])
```

← Over this

```
with open('../resources/celebrity_100.csv', encoding='utf-8') as f:
    for name, pay, year, category in itertools.islice(csv.reader(f), 3):
        print(name, pay)
```

← And over this

ch01\_overview/idioms.py

In the example above, the first approach does two things right over the subsequent two. First, it uses underscores ( \_ ) for values it doesn't care about. The year and category aren't used, so instead of variables, underscores denote their non-use. Second, unpacking the for-loop variable into 4 values is easier to read than leaving it as a single variable, named items.

## Python Idioms for Iterating (3 of 3)

- Consider generators and generator expressions over list building

```
def get_names(filename: str = '../resources/celebrity_100.csv'):
    results = []
    with open(filename, encoding='utf-8') as f:
        for name, _, _ in itertools.islice(csv.reader(f), 3):
            results.append(name)
    return results
```

← Over this

```
for celebrity in get_names():
    print(celebrity)
```

```
def get_names(filename: str = '../resources/celebrity_100.csv'):
    with open(filename, encoding='utf-8') as f:
        for name, _, _ in itertools.islice(csv.reader(f), 3):
            yield name
```

← Prefer this

```
for celebrity in get_names():
    print(celebrity)
```

ch01\_overview/idioms.py

While both output the same thing, the example above creates a list in memory first only to simply output it a moment later. The second example never creates the list first, instead it outputs values one at a time as they are being read.

## Typing and Python

- Python 3.5+ added the use of type declarations, called *type hints* or *annotations*
- Types in Python are optional, ignored during runtime
- The case for type hints increases as applications get more and more complex
- The *typing* module can assist with type declarations

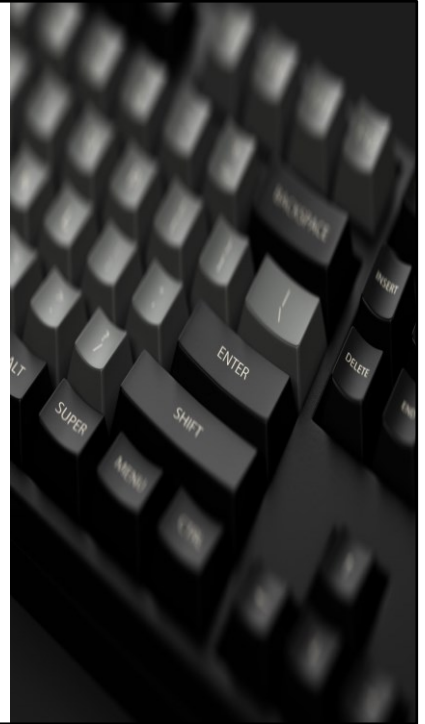
```
from typing import NamedTuple

Player = NamedTuple('Player',
                    [('name', str), ('height', int)])

def create_player(name: str, height: int) -> Player:
    return Player(name, height)

create_player('John', 'tall')
```

Violates the type hint



ch01\_overview/using\_type\_hints.py

Type hints are a result of growth in Python's use. Python applications have become more and more complex. This complexity doesn't always work well with the typical duck typing syntax that Python exhibits particularly when larger teams become involved in the project. More information must be conveyed to team members than just "this function accepts a variable called 'x'."

Type hints were added in Python 3.5. However, to avoid making breaking changes to the language, type hints don't have any affect at runtime. A "smart" IDE or editor that understands type hints can provide build-time info on specified types.

# Improvements to Type Hints (Python 3.6+)

- Python later extended type hints to individual variables

```
from typing import List, Tuple, Iterable
```

Using the typing module allows for declaring numerous specific type definitions

```
name: str
```

```
height: int
```

```
TeamList = List[Player]
```

Note: the equals (=) creates a custom type

```
def update_team(names: Iterable[Tuple[str, str]], heights: Iterable[int],
                team_list: TeamList) -> None:
    for name, height in zip(names, heights):
        team_list.append(Player(' '.join(name), height))
```

```
team = []
```

```
update_team([('John', 'Smith')], (177,), team)
```

```
update_team([('Sally', 'Jones')], ('163',), team)
```

Violates the type hint

You should use type hints when:

- Apps get complex or large
- Team development occurs
- Architecture is a concern

ch01\_overview/using\_type\_hints.py

Originally, only functions supported the use of type hints. Starting in Python 3.6, type hint support was extended allowing variables to be declared with type hints (as shown with name and height above) and custom types could be created (as shown with TeamList above).

Other interesting type declarations include:

```
from typing import Optional
```

```
PlayerTuple = tuple[str, int, Optional]
```

```
tup1: PlayerTuple = ('Ronaldo', 10, '')
```

```
tup2: PlayerTuple = ('Ronaldo', 10, None)
```

# Generators are iterators and thus return an Iterator type  
(where the parameter in brackets is the yield type):

```
def gen(max_count: int) -> Iterator[int]:
```

```
    i = 0
```

```
    while i < max_count:
```

```
        yield i
```

```
        i += 1
```

```
for val in gen(5):
```

```
    print(val)
```

## Continued Type Hint Improvements

Py 3.9

```
from typing import Union, Any
team: list[Player] = []
players: dict[str, tuple] = {'John': ('John', 177)}
```

Use of built-in types for type definitions

Py 3.8

```
def create_player(name: Union[str, None], height: Any) -> Player:
    if not name:
        name = 'Cristiano Ronaldo'
    return Player(name, height)

team.append(create_player('John', 'tall'))
```

Union[] to specify multiple possible types  
Any to specify any values

Py 3.10

```
from typing import Optional
def create_player(name: Optional[str], height: int | float) -> Player:
    if not name:
        name = 'Cristiano Ronaldo'
    return Player(name, height)
```

Optional[type] is the same as Union[type, None]  
X | Y is the same as Union[X, Y] (Python 3.10 only)

```
create_player(None, 185)           # allowed
create_player('John', 'tall')     # not allowed
```

ch01\_overview/using\_type\_hints.py

Additional type hint features continue to be added into subsequent Python versions. Python 3.9 allowed the native types (list, tuple, dict, set) to be used to declare type definitions instead of using the typing module versions (List, Tuple, Dict, Set). The Union type, introduced in Python 3.8 allowed us to declare types that may be one type **or** another (like an "or" Boolean operator)

Python 3.11 introduces the **Self** type (typing.Self).

For more on type hints, visit:

<https://docs.python.org/3/library/typing.html>



## Mini-task 1

- Refactor a working solution (that isn't following PEP 8 rules or idiomatic Python very well)
- Work from [mini\\_task1.py](#) within `ch01_overview`

The code already works, simply clean it up to be PEP 8 compliant and be more "Pythonic"

# Object-Oriented Programming

```
class Celebrity:
    def __init__(self, name: str, pay: int, year: str, category: str):
        self.name = name
        self.pay = pay
        self.year = year
        self.category = category

    def __str__(self):
        return f'{self.name} ({self.category})'

    __repr__ = __str__

filepath = Path('../resources/celebrity_100.csv')
with filepath.open(encoding='utf-8') as f:
    f.readline()
    for line in islice(f, 10):
        celeb = Celebrity(*line.strip().split(','))
        print(celeb)
```

Oprah Winfrey (Personalities)  
Tiger Woods (Athletes)  
Mel Gibson (Actors)  
George Lucas (Directors/Producers)  
Shaquille O'Neal (Athletes)  
Steven Spielberg (Directors/Producers)  
Johnny Depp (Actors)  
Madonna (Musicians)  
Elton John (Musicians)  
Tom Cruise (Actors)

ch01\_overview/classes.py

Python uses classes heavily throughout its standard library and within the countless third-party tools. Classes are essential in order to be effective within this language and they are highly important in defining architecture.

The example shown creates a Celebrity class. The code at the bottom reads from a file, line-by-line, up to 10 lines (after the header). The data read from each line from the file is placed into a Celebrity object. The object is then printed.

The `__init__()` magic method initializes attributes and is called automatically when a `Celebrity()` object is instantiated. The `__str__()` method is used to convert the object into a string representation. It is invoked at the bottom when the `print(celeb)` statement is called. The `__repr__()` method might come into play if we were getting info on a collection of `Celebrity` objects

## Modeling Objects and Relationships (1 of 2)

```
class House:
    def __init__(self, sq_ft: int = 0, location: str = '', value: int = 0):
        self.sq_ft = sq_ft
        self.location = location
        self.value = value

class PrivateJet:
    def __init__(self, typ: str, capacity: int = 0, cost: int = 0):
        self.typ = typ
        self.capacity = capacity
        self.cost = cost

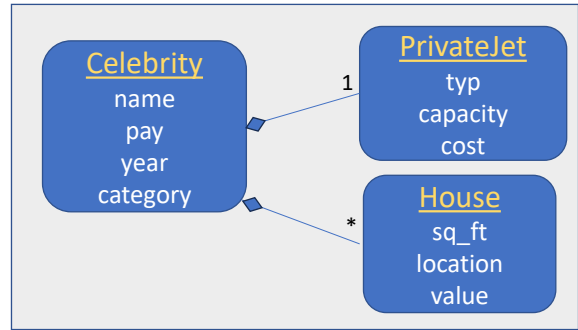
class Celebrity:
    def __init__(self, name: str, pay: int, year: str, category: str):
        self.plane = None
        self.houses = []
        self.name = name
        self.pay = pay
        self.year = year
        self.category = category
```

ch01\_overview/relationships.py

Classes, alone, are often enough to build an architecture. Usually there will be many objects made from many classes. These object (and classes) interact with each other. There are several ways to model the relationships between classes and objects. Shown here is a composition where a Celebrity class includes a private jet and list of houses. Notice how the modeled relationship is one of "ownership." The Celebrity "has a" plane and "has a" (bunch of) houses.

While each object has its own identity (existence in memory) and its own type, the Celebrity object would contain a reference to the plane and houses objects.

## Modeling Objects and Relationships (2 of 2)



```
celeb = Celebrity('Oprah Winfrey', 225, 2005, 'Personalities')

celeb.plane = PrivateJet('Gulfstream 550', 16, 40_000_000)

celeb.houses.append(House(23_000, 'Santa Barbara, CA', 50_000_000))
celeb.houses.append(House(7_300, 'Orcas Island, WA', 8_300_000))
celeb.houses.append(House(5_000, 'Santa Ynez, CA', 29_000_000))
celeb.houses.append(House(location='Montecito, CA', value=7_000_000))
celeb.houses.append(House(8_700, 'Telluride, CO', 14_000_000))
celeb.houses.append(House(location='Maui, HA', value=32_000_000))
```

ch01\_overview/relationships.py

Due to the rules of visibility within Python, the plane and houses attributes can be established while creating the Celebrity (in the `__init__()`) or even *after* creating the celebrity object. In addition, access to attributes within the PrivateJet or House is very easy. For example, to display the celebrity's private jet capacity:

```
print(celeb.plane.capacity)
```

Or, to display the location of Oprah's third house:

```
print(celeb.houses[2].location)
```

# Inheritance

```
class Person:
    def __init__(self, name: str, pay: int = 0):
        self.name = name
        self.pay = pay

    def __str__(self):
        return f'{self.name} ({self.pay})'

class Celebrity(Person):
    def __init__(self, name: str, pay: int = 0, year: int = 0, category: str = ''):
        super().__init__(name, pay)
        self.year = year
        self.category = category

celeb_data = Path('../resources/celebrity_100.csv') \
    .read_text(encoding='utf-8').split('\n', maxsplit=2)[1]

celeb = Celebrity(*celeb_data.split(','))
print(celeb)
```

ch01\_overview/inheritance.py

Another way to model the relationship between classes is via inheritance. In this example, a Celebrity is a Person and therefore invokes the `__init__()` of the parent by using the `super()` method call. This ensures the name and pay of the parent class are properly initialized for the child class object.

# Abstract Classes

- Abstract classes can be used to establish a "contract" for subclasses
  - The contract guarantees the child class will contain needed operations
- Python provides a module for working with abstract classes called **abc**
- Methods become abstract in Python by marking them with a decorator:

```
import abc
class Employee(abc.ABC):
    @abc.abstractmethod
    def calc_pay(self):
        pass

emp = Employee()
```

TypeError: Can't instantiate abstract class Employee with abstract method calc\_pay

Python relies on the **abc** module for abstract class implementation. Abstract classes should inherit from the ABC class defined within this module. In the parent, mark the methods using the *@abstractmethod* decorator that will be required by children classes.

Attempting to instantiate objects containing unimplemented abstract methods will cause the error shown above to occur.

## Abstract Classes (continued)

```
import abc
```

```
class Employee(abc.ABC):
    def __init__(self, name):
        self.name = name

    @abc.abstractmethod
    def calc_pay(self):
        pass
```

```
class Management(Employee):
    def __init__(self, name, salary):
        super().__init__(name)
        self.salary = salary

    def calc_pay(self):
        return self.salary / 26.0
```

```
class HourlyEmployee(Employee):
    def __init__(self, name, rate):
        super().__init__(name)
        self.hours_worked = 0
        self.rate = rate

    def calc_pay(self):
        return self.rate * self.hours_worked
```

```
def create_paycheck(emp: Employee):
    return emp.calc_pay()
```

```
emp1 = Management('Sally Jones', 85_000)
emp2 = HourlyEmployee('John Smith', 37.50)
emp2.hours_worked = 80
```

```
print(f'Manager: {create_paycheck(emp1):.2f}, \
      Hourly Employee: {create_paycheck(emp2):.2f}')
```

ch01\_overview/abstract.py

The example shows the use of abstract classes in Python. The `create_paycheck()` method accepts an `Employee` object. Because both `Management` (`emp1`) and `HourlyEmployee` (`emp2`) objects are children of `Employee`, they can be passed into this `create_paycheck()` without any problems.

# Abstract Classes and Duck Typing

```
class SlotMachine:
    def __init__(self, input_amount: float):
        self.input_amount = input_amount
        self.rate_usage = 0

    def calc_pay(self):
        return self.rate_usage * self.input_amount

def create_paycheck(emp: Employee):
    return emp.calc_pay()

sm = SlotMachine(0.25)
sm.rate_usage = 200
print(f'Works for a slot machine: {create_paycheck(sm)}')

emp3 = Employee('Oscar Bradford') # can't instantiate this
```

Python doesn't care about the parents to an object, only that it contains the required methods

Still works to pass sm into create\_paycheck() even though it is not an Employee

ch01\_overview/abstract.py

If we stick to our knowledge of how abstract classes work in other object-oriented languages, we may find ourselves in trouble when it comes to Python...

Abstract classes can be a bit controversial in Python. They may beg the question, "Do we even need them?"

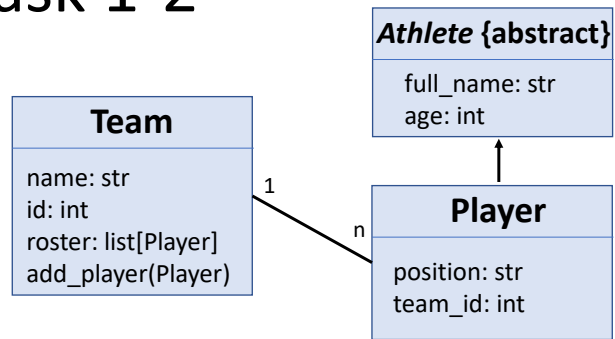
For example, consider a SlotMachine object which is not at all related to the Employee class. Since SlotMachine has a calc\_pay() method, it can be successfully passed into create\_paycheck() without any errors. This is because within create\_paycheck(), calc\_pay() is called. SlotMachine abides by this contract and has a calc\_pay(). Therefore, no error occurs. In other words, Python doesn't actually care what gets passed in as long as the object contains the requisite methods that will be invoked.

If that's the case, why do we even need abstract classes? Answer: for ourselves. Because create\_paycheck() claims to accept an Employee object, we know that we should only be passing in Employee objects. Not SlotMachines.



## Your Turn! - Task 1-2

- Object Composition and Abstract Classes
- Work from [task1\\_2\\_starter.py](#)
- Model the following classes shown to the right



## Python and SOLID Principles

- SOLID is a set a design principles aimed at creating software that is easy to maintain and understand

**SRP** - Single responsibility principle  
**OCP** - Open/closed principle  
**LSP** - Liskov substitution principle  
**ISP** - Interface segregation principle  
**DIP** - Dependency inversion principle

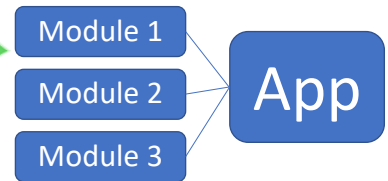
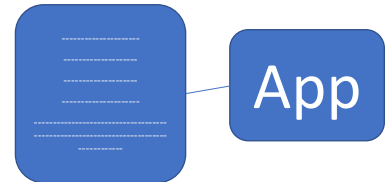
Does *SOLID*  
apply in Python?

SOLID represents a series of guiding principles in many languages (particularly statically typed languages) such as Java that have helped shaped solutions over the years. Does SOLID apply to Python?

In some cases, yes. In others no....

# Single Responsibility Principle (SRP)

- *A module should have only one reason to change*
- Modules (and classes) should specialize in doing one thing
  - It should only have one action, or
  - It should only focus on one topic
- Standard library modules follow this principle
  - `datetime`, `subprocess`, `thread`
- Create modules that have a single focus
- Violating this principle may cause a module to be highly coupled across the system



When a class has multiple responsibilities, it increases the likelihood that it will need to be changed more frequently over time.

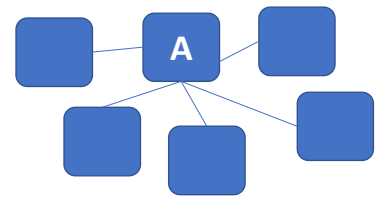
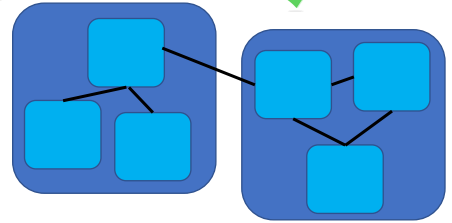
Single responsibility applies to Python. Having smaller modules allows apps to load only what is needed. At the class-level, this design principle applies as well. Classes in Python are the same as in other languages--keep them singular in concept/focus which ultimately improves reusability.

# Cohesion and Coupling

This is SRP!

- Favor **high cohesion**
  - How well elements within a module belong together
  - This builds better reusability, robustness, and understandability
- Favor **loose (low) coupling**
  - How much elements from different modules refer to each other
  - Changes in one module won't affect another module as much

High cohesion: Related items are contained together decreasing interdependencies ✓



High coupling: If module **A** changes, it may affect all other modules

High cohesion simply means that the things that need to talk to each other (functions, classes) are located with each other. In other words, put related items together in a module.

Coupling means that classes and functions that reference other items (classes, functions, and variables) create a dependency (a line between boxes, as in the lower picture). The desire is to have the least number of lines between boxes in the lower diagram. Fewer lines represents lower coupling.

# Open-Closed Principle (OCP)

```
class Persist:
    def get_connection(self):
        print('connect to db')

    def save(self):
        print('saving...')
```

```
class PersistOracle(Persist):
    def get_connection(self):
        super().get_connection()
        print('connect to oracle')
```

```
class PersistMySQL(Persist):
    def get_connection(self):
        super().get_connection()
        print('connect to MySQL')
```

- Open for extension, but closed for modification
- Classes should change their behavior via children classes, not internally

```
def new_get_connection(self):
    print('new get connection')
```

This violates OCP!

```
Persist.get_connection = new_get_connection
p = PersistMySQL()
p.get_connection()
```

new get connection  
connect to MySQL

ch01\_overview/ocp.py

This principle states that classes (since modules can't be extended in Python only classes will pertain to this rule) should only be extended from, they should not be modified directly (sometimes referred to as monkey patching).

As a good rule of thumb and best practice, extending classes should be the primary approach considered. However, because of the ease with which methods and attributes can be substituted, monkey patching does occur in Python. A degree of monkey patching is acceptable, such as in testing; however, overuse of this technique can lead to fractured code and can lead to unexpected behaviors in children classes.

# Liskov Substitution Principle (LSP)

- Subclasses should be substitutable with their base classes

```
class Pond:
    def interact(self,
                  quacker: Quacker):
        quacker.quack()
        quacker.swim()

pond = Pond()
pond.interact(Duck())
pond.interact(Frog())
pond.interact(Human())
pond.interact(Fish())
```

These 3 work!

AttributeError!

- This principle is *somewhat* valid in Python
  - Subclasses should be usable where a parent class is used
  - In Python, any object with the proper methods will work
    - Duck typing

```
class Quacker:
    def quack(self):
        print('quack!')

    def swim(self):
        print('splash!')

class Duck(Quacker):
    pass

class Frog(Quacker):
    pass

class Human:
    def quack(self):
        print('quack!')

    def swim(self):
        print('splash!')

class Fish:
    def swim(self):
        print('splash!')
```

ch01\_overview/lsp.py

If a method has a dependency of a given type, you should be able to provide an instance of that type or any of its subclasses without introducing unexpected results and without the method knowing the actual type provided (i.e., a Frog object can be passed into the interact() and it should work because Frog can do everything a Quacker can do).

And yes, some frogs can quack.

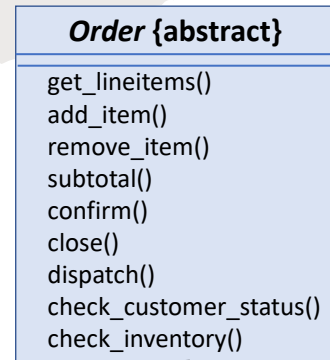
# Interface Segregation Principle (ISP)

- **Favor smaller interfaces**

- Smaller interfaces keep classes from violating the Single Responsibility Principle
- Python doesn't have interfaces, but can enforce behavior via abstract classes

```
class CustomOrder (Order) :  
    def check_inventory(self) :  
        pass
```

Does a custom order need a check\_inventory()?  
What do we do with it now?



CustomOrder

NormalOrder

When concrete classes are dependent upon a specific set of interfaces, then the concrete class must implement all of the methods of those interfaces. If interfaces have many methods, then the concrete classes also must have many (typically unneeded) methods. This leads to using adapters to implement stubs for the unneeded methods.

These concepts are rarely encountered in Python; therefore, this principle gets less consideration in Python.

# Dependency Inversion Principle (DIP)

- Depend upon abstractions, not concretions
- Possibly one of the most important of the SOLID principles to architecture
- You should adhere to this principle
- Python supports this principle (though it can be bypassed)

```
class Pond:
    def interact(self,
                  quacker: Quacker):
        quacker.quack()
        quacker.swim()

pond = Pond()
pond.interact(Duck())
pond.interact(Frog())
pond.interact(Human())
```

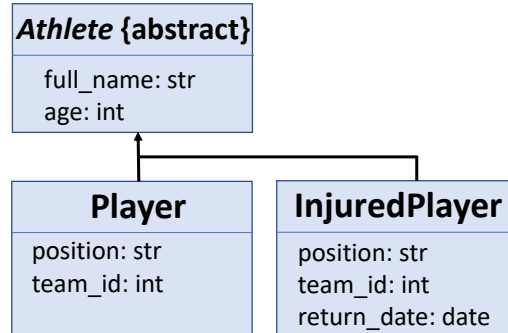
We did not specifically identify which type of object to pass into `interact()`, but rather, we indicate passing in a more generalized type. This way, we can easily pass different object types and we can easily mock it for testing purposes.

Though Python supports passing in "non-Quacker" types (as long as they swim and quack), at the design level we can ensure that our object hierarchy is valid and supported properly.



## Your Turn! Task 1-3

- Task1-2 now includes an InjuredPlayer (as shown)
- Redesign Task 1-2 to utilize DIP
- Change the Team's add\_player() method to add\_athlete()



*Work from task1\_3\_starter.py*

## CH 1 SUMMARY

- ▶ Architecture patterns deal with the structure of the overall system
- ▶ Python has its own rules and guidelines for how best to write code
- ▶ The SOLID principles deal with best practices for designing classes



# Domain Modeling

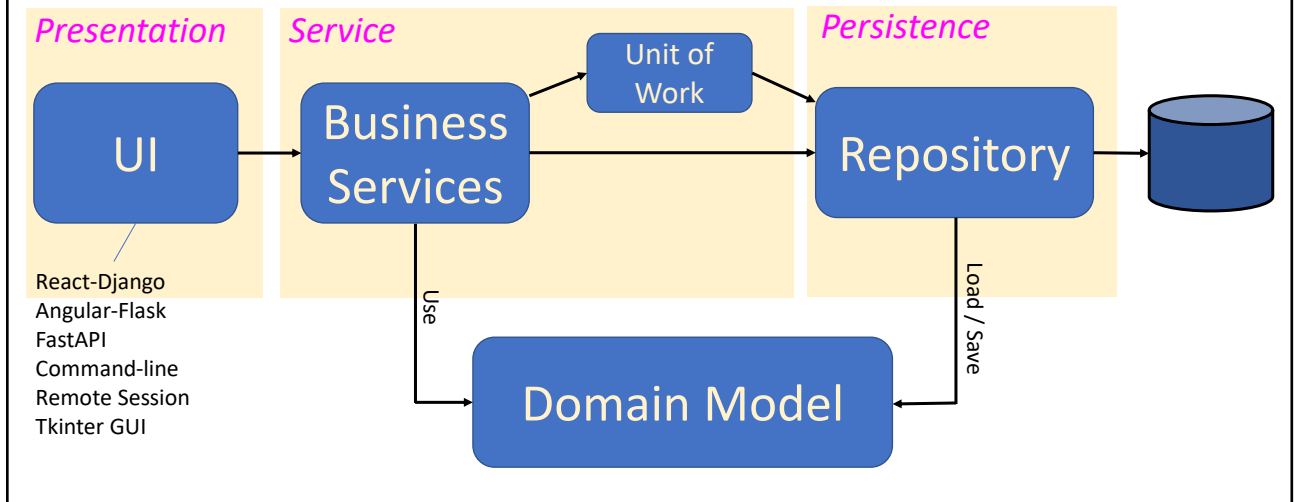
# Domain Modeling Overview

---

- Domain Models
- Repositories
- Python Object-Relational Mapping

## Overall Architecture

- A first look at our architecture

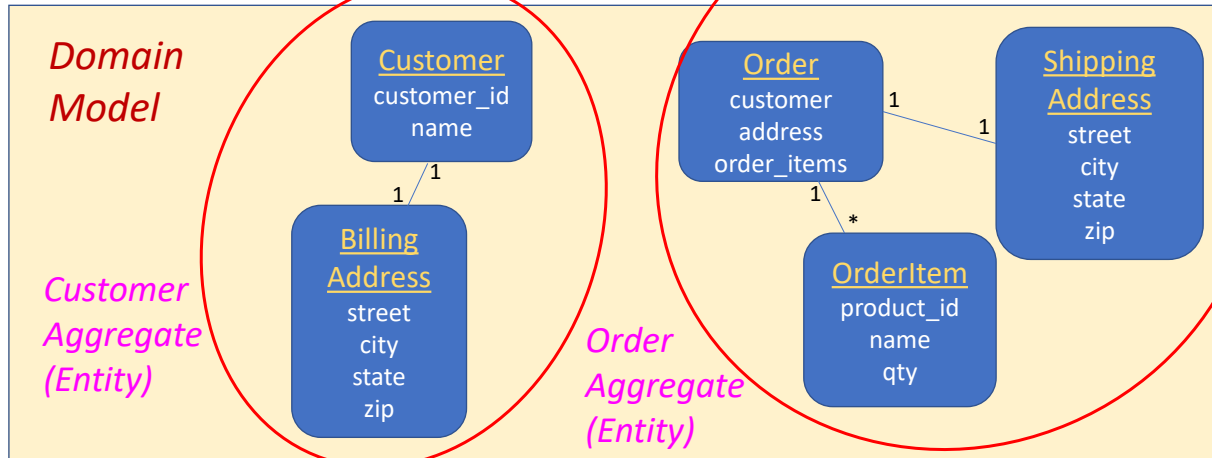


A layered architecture helps break a system into smaller manageable pieces. Layered architectures are classic architectures that have been around for many decades. The layered architecture can sometimes be found with modern variations (as in the Clean, Onion, or Hexagonal Architectures). Layered architectures even exist within other architectures, such as microservices. A microservice architecture will often describe the way a system is distributed while a layered architecture describes the layout of the system's internal components (more on microservices later).

Layered architectures create separations between different subsystems within the application. For example, in a layered architecture, the UI should not be aware of the persistence layer operations. The persistence layer should not "know about" business method operations. By separating concerns between layers, we can hope to minimize efforts needed to incorporate change in the application. Reducing the coupling between layers, reduces the affects of change on other subsystems (layers).

## What is a Domain Model?

- A domain model identifies objects that specify your business rules



The domain model is your object model. It knows about the business rules and relationships. It knows about the terms of your business domain. It is usually represented with a series of class diagrams. It *does not* know about presentation or other frameworks frameworks. A domain model describing an ordering system would contain classes and relationships related to accounts, lineitems, orders, transactions, etc. How these objects interact with each other is part of that model.

An *aggregate* is often composed of one or more domain objects. This combination of objects makes up an *entity*. Entities (aggregates) can be used throughout the system. They can be passed to the repository layer or provided to the presentation layer.

## Python Dataclasses (Python 3.7+)

- Python dataclasses can be used to build domain models
  - Domain models consist of classes that define "business rules"
  - Business rules are the methods that define how the models communicate with each other (how they message each other)

```
@dataclass
class Order:
    status: str
    customer: Customer
    address: Address
    order_items: Set[OrderItem] = None

    def add_item(self, item: OrderItem):
        for order_item in self.order_items:
            if item.id == order_item.id:
                order_item.qty += item.qty
            else:
                self.order_items.add(item)

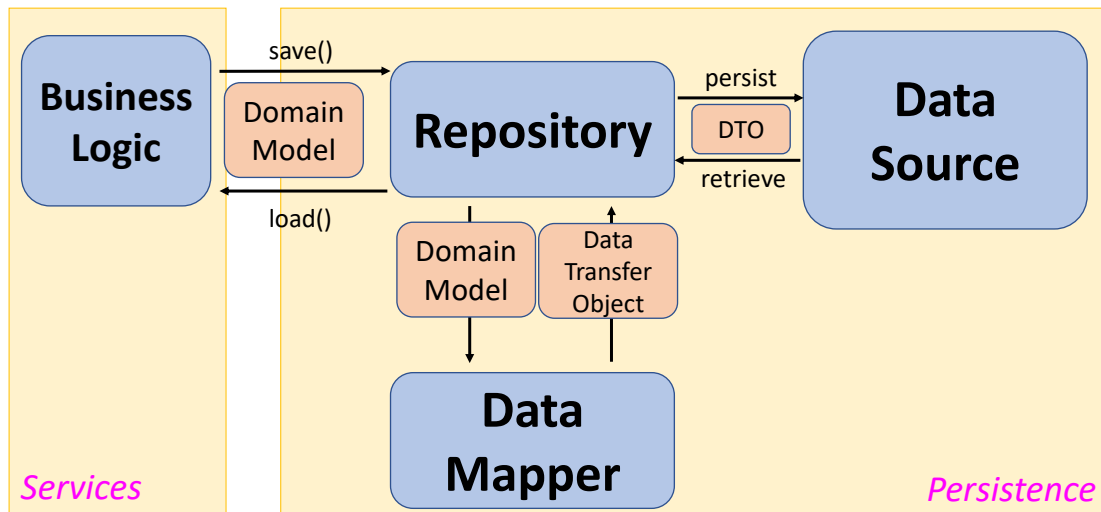
    def __str__(self):
        return f'{self.status}: {self.customer.name}, {self.order_items}'
```

ch02\_domain/data\_classes.py

In domain driven design, Order would be considered an aggregate and we would treat it as one entire business entity. It can have attributes that are composed of other objects and methods that define how it communicates with other entities.

Domain models do not need to be created from Python Dataclasses but they are a nice addition because they are easy to create and manage.

## Classic Repository Pattern



The classic repository pattern is responsible for making persistence operations transparent to the business services layer. Business services should only know about the repository and its methods. If a method in the repository is called `save()`, for example, the business service layer should not know anything about where `save()` actually saves its data. Nothing says the repository object serves a database. It might work with a JSON API or the file system.

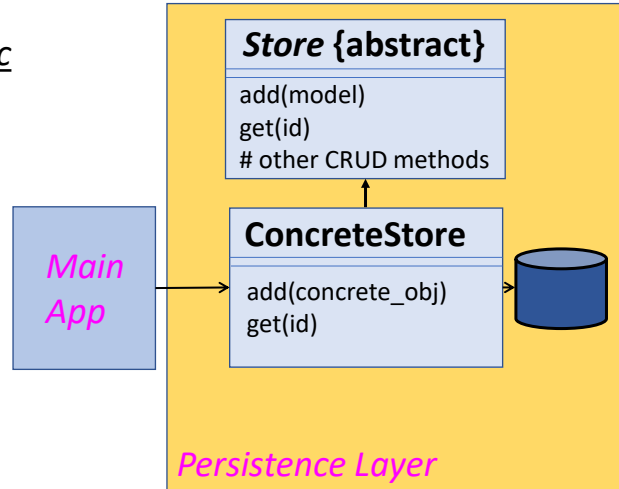
It is proper to send domain model objects from the services layer to the repository or to retrieve them from the repository. Data Transfer Objects (DTOs) are used within the persistence layer only to persist data. DTOs are intimate with the persistence framework and therefore should not be returned to the business services layer. DTOs will likely contain code that is framework specific (for example, they may contain knowledge of a database). It is not correct to send Data Transfer Objects (DTOs), objects geared for working with persistence operations or the persistence layer, back to the business logic.

This means objects related to persistence should be converted to domain model objects. Converting from domain model to DTO format is the role of a data mapper and the data mapper pattern.



## Creating the Repository Pattern

- Repositories *encapsulate the logic* for working with data sources
- They decouple the service or business layer from persistence infrastructure
- For each domain aggregate there should be one repository class
  - An aggregate is an overall business object, such as our Order aggregate previously shown



The repository pattern begins by creating an abstract base class that defines methods needed to work with the model against the data source. The concrete store will contain the actual logic for working with our data source. In this case, it will be an SQLite database. Other methods, such as `delete()`, `find_by_XXX()`, `update()`, etc., might be defined as needed, but keep the interface small (ISP) .

# Interacting with a Database Using Python

```

from pathlib import Path
import sqlite3
import sys

conn = None
try:
    db_url = Path(__file__).parents[1] / 'resources/course_data.db'
    conn = sqlite3.connect(db_url)
except sqlite3.Error as err:
    print(f'Error connecting to database. {err}')
    sys.exit(42)

print('Fetching multiple records:')
try:
    cursor = conn.cursor()
    cursor.execute('SELECT common_name, country, wins, draws, losses FROM teams')
    for record in cursor:
        print(record)
except sqlite3.Error as err:
    print(f'Database interaction error: {err}')

```

All RDBMS modules will have a connect() method

Obtain a cursor from the connection object

Iterate over the cursor directly

ch02\_domain/working\_with\_the\_db.py

Python supports a standard API for working with any RDBMS (NoSQL databases are not supported with this API). Each database vendor should provide a *connect()* method which returns a cursor. The cursor contains an *execute()* method to execute the SQL against the database. The cursor will also hold the results which must then be iterated over or retrieved.

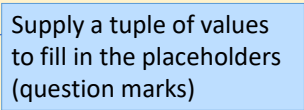
## Supplying SQL Parameters

```
team_id = 1
statement = 'SELECT common_name, country, wins, draws, losses FROM teams WHERE id = ?'
try:

    cursor = conn.cursor()
    cursor.execute(statement, (team_id,))
    print(cursor.fetchone())

except sqlite3.Error as err:

    print(f'Database interaction error: {err}')
```



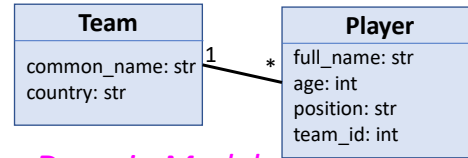
Supply a tuple of values to fill in the placeholders (question marks)

To tailor the SQL, variables to the SQL statements should be supplied as arguments to the `execute()` method as shown above. In this case, the `team_id` variable needed to be inserted into the SQL statement so a tuple containing one element was provided to the `cursor.execute()` method.

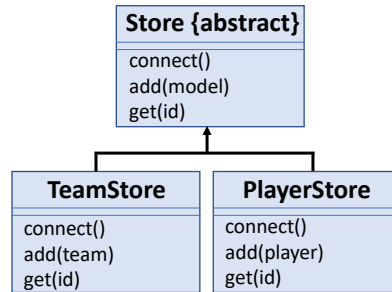
The fetch methods return database values. `fetchone()` and `fetchmany()` both return a limited number of records. Records are not actually retrieved from the database until they are fetched.

## Your Turn! - Task 2-1

- Use the *repository pattern* to build a persistence layer for the following relationship
- Work from the `ch02_domain/task2_1_starter` folder



*Domain Model*



*Repository*

Work from the files  
in this order:  
`models.py`  
`base_store.py`  
`team_store.py`  
`player_store.py`  
`driver.py`

Optionally, work from the workbook in the back of the student manual.

## Advanced: Designing with Generics

```
from typing import Generic, TypeVar
```

```
T = TypeVar('T')
```

```
class GenericStore(Generic[T]):
    def add(self, entity: T) -> None:
        # do "add" related stuff for any type entity here
```

```
    def get(self, id: int) -> T:
        # retrieve entity for any type here
```

```
class TeamStore(GenericStore[Team]):
    def add(self, entity: Team) -> None:
        # override parent implementation if desired

    def get(self, id: int) -> Team:
        # override parent implementation if desired
```

This works best when methods in children classes would be very similar

Smart IDE says, "Expected type 'Team' got 'int' instead"

```
ts = GenericStore[Team]()
ts.add(Team('Lakers', 'USA'))
ts.add(10)

ts2 = TeamStore()
ts2.add(Team('Marlins', 'USA'))
```

ch02\_domain/generics.py

While not commonly used in Python, it's possible to apply Generics to avoid having to rewrite lots of similar classes over and over that usually vary only in the type differences. Here, `T`, represents any possible type, like `Account`, `Customer`, `Team`, `Player`, etc.

We can then instantiate a `GenericStore` object based on the bracketed type, `[ ]`, provided. So, a `GenericStore[Team]` would be expected to take a `Team` object in its `add()` method or return a `Team` from its `get` method.

Smart IDEs (such as `PyCharm`) and static type checkers such as `mypy` or `Pyre` can be run to detect these "incompatibilities."

We won't use Generics for any further examples in this course.

## Python ORM with SQLAlchemy

- Object-relational mapping (ORM) is the task of converting Python objects to database relations and back
  - Python's most popular ORM tool is called **SQLAlchemy**
- Steps toward working with SQLAlchemy
  - 1) Define how to *map classes* to database tables
  - 2) *Initialize* SQLAlchemy providing connection info
  - 3) Perform database operations within a *session*

ORM tools are popular for mapping between Python objects and databases. **SQLAlchemy** can do this with little effort typically without the use of any manually created SQL.

SQLAlchemy provides support for various relational databases. See this reference for supported databases: <https://docs.sqlalchemy.org/en/20/dialects/index.html>

# SQLAlchemy Model Definitions

- Step 1. Create DB Model Definitions: *table-to-class* mappings

```
from sqlalchemy import create_engine, Column
from sqlalchemy.orm import sessionmaker, declarative_base
from sqlalchemy.types import String
from sqlalchemy.exc import SQLAlchemyError
```

```
Base = declarative_base()
```

Dynamically creates the base class  
for your classes to inherit from

```
class NetflixTitle(Base):
```

Name of table in database

id = attribute name in class  
show\_id = column name in table

```
    __tablename__ = 'netflix_titles'
    id = Column('show_id', String(10), primary_key=True)
    type = Column(String(20))
    title = Column(String(60))
    cast = Column(String(250))
    date_added = Column(String(40))
    rating = Column(String(10))
```

This class defines which class  
attributes and database table  
columns map to each other

ch02\_domain/using\_sqlalchemy.py

SQLAlchemy allows for sophisticated mappings between your classes and the database relations. You can map one-to-one, many-to-many, one-to-many, and many-to-one relationships within your classes.

Column() definitions defined within the class are used to map attributes to columns in the database table. String() types correspond to varchar types. Other column types may be specified, such as Integer, Boolean, and Numeric. Relationships between objects are defined in the object model which should correspond to the relationships defined in the database tables. A one-to-many relationship, for example, could be modeled as follows:

```
class School(Base):
    __tablename__ = 'schools'
    school_id = Column(String(30), primary_key=True)
    students = relationship('Student')

class Student(Base):
    __tablename__ = 'students'
    student_id = Column(String, ForeignKey('schools.school_id'))
```

# SQLAlchemy Initialization

- Step 2. Initialize SQLAlchemy using `create_engine(dburl)`

```
from sqlalchemy import create_engine, Column
from sqlalchemy.orm import sessionmaker, declarative_base
from sqlalchemy.types import String
from sqlalchemy.exc import SQLAlchemyError

db_url = Path(__file__).parents[1] / 'resources/course_data.db'

class NetflixTitle(Base):
    ... (unchanged from previous slide)

try:
    db = create_engine('sqlite:/// ' + str(db_url), echo=True)
    Session = sessionmaker(bind=db)
except SQLAlchemyError as err:
    print(f'Error connecting to database. Error: {err}', file=sys.stderr)
    sys.exit()
```

Prepares to connect to the db. No actual connection occurs at this point

Other databases will have slightly different connection strings

```
create_engine('postgresql://user:pswd@localhost:1542/schools')
```

ch02\_domain/using\_sqlalchemy.py

The `create_engine()` call doesn't connect to the database immediately. It does, however, prepare a pool of connections and establish the dialect to be used. Different databases have different connection URLs. An example showing how to connect to PostgreSQL is shown at the bottom of the slide.

The call to `sessionmaker()` creates a *Session* class that is tailored to work with our specific database. The *Session* class instantiates a *session* object used performs operations such as `add()`, `update()`, `query()`, `commit()`, `rollback()`, `delete()`, etc.



# SQLAlchemy Usage

The *session* object is used to track and manage persistent objects and commit changes to the database

## Session Methods Include:

*add(obj)* - INSERT objects into the database  
*query(Model)* - SELECT objects from the database  
*query(Model).filter(params)* - SELECT objects using a WHERE clause  
*query(Model).get(id)* - SELECTS objects by primary key  
*delete(obj)* - DELETES obj from the database  
*commit()* - can UPDATE modified objects

```
def title_query(title: str):
    query_results = []
    try:
        with Session() as session:
            query_results = session.query(NetflixTitle)
                                .filter(NetflixTitle.title.like('%'+title+'%')).all()
    except SQLAlchemyError as err:
        print(f'Error working with db. Error: {err}', file=sys.stderr)
    return query_results

results = title_query('Charlie')
format_str = '{0:<50} ({1}, {2})'
print(format_str.format('Title', 'Type', 'Rating'))
for record in results:
    print(format_str.format(record.title, record.type, record.rating))
```

Title	(Type, Rating)
Charlie's Angels	(Movie, PG-13)
Charlie and the Chocolate Factory	(Movie, PG)
Charlie's Angels: Full Throttle	(Movie, PG-13)

ch02\_domain/using\_sqlalchemy.py

The session object is the primary object used for database interactions. It is closely tied to the transaction. The session tracks and manages objects performing inserts, updates, queries, and deletes as needed. Usually the session behaves lazily, only performing tasks when absolutely required. This way it can minimize extra or unneeded calls to the database.

## SQLAlchemy Updating an Object

```
try:
    with Session() as session:
        session.begin()
        show = session.query(NetflixTitle).get('s3393')

        show.title = '10 Days in Moon City'

        session.commit()
except SQLAlchemyError as err:
    print(f'Rolling back. Error: {err}', file=sys.stderr)
    session.rollback()
```

Retrieves the object by its id

Modifies its title

Performs an UPDATE and commits

ch02\_domain/using\_sqlalchemy.py

SQL operations might not occur until the session's `commit()` method is called. In this example, an object queries and retrieves a Netflix movie by its primary key. The object is then modified (the title is changed). Upon committing, the SQL UPDATE operation occurs.

## SQLAlchemy Complete Picture

```
Base = declarative_base()

class NetflixTitle(Base):
    __tablename__ = 'netflix_titles'
    id = Column('show_id', String(10), primary_key=True)
    type = Column(String(20))
    title = Column(String(60))
    cast = Column(String(250))
    date_added = Column(String(40))
    rating = Column(String(10))

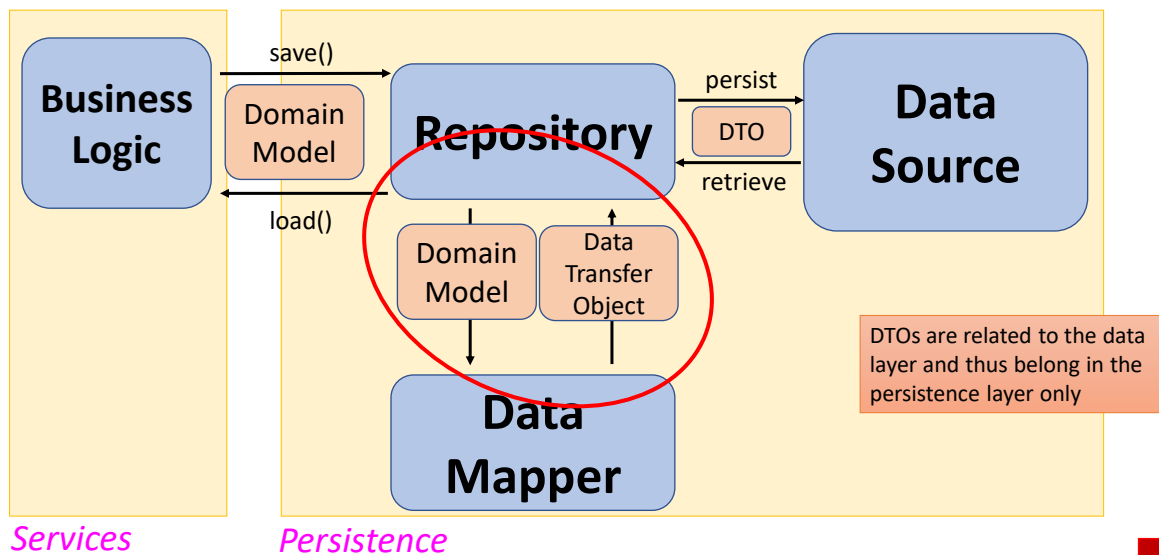
db = create_engine('sqlite:/// ' + str(db_url), echo=True)
Session = sessionmaker(bind=db)

try:
    with Session() as session:
        session.begin()
        show = session.query(NetflixTitle).get('s3393')
        print(show.title)
        show.title = '10 Days in Moon City'
        session.commit()
except SQLAlchemyError as err:
    print(f'Rolling back. Error: {err}', file=sys.stderr)
    session.rollback()
```

ch02\_domain/using\_sqlalchemy.py

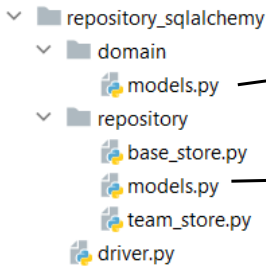
A complete picture of our SQLAlchemy NetflixTitle table interaction is shown. For brevity, some exception handling has been removed.

## Data Transfer Objects and SQLAlchemy



DTOs are merely objects that are used to persist data. DTOs can be specific to a persistence framework. Your business model objects should not have any specific persistence framework-related code (such as SQLAlchemy). That's the whole point of a repository. DTOs, on the other hand, are used by the persistence framework only; therefore, it is okay to place persistence-related information in these classes.

# Domain Model Objects vs Data Transfer Objects



```
@dataclass
class Team:
    common_name: str
    country: str

    # business methods allowed
```

*Domain Model*

```
class TeamDTO(Base):

    __tablename__ = 'teams'

    id = Column('id', Integer, primary_key=True)
    common_name = Column(String(40))
    country = Column(String(40))
```

*Persistence Model*

One of the primary goals of a layered architecture is to keep the layers from mixing implementation-related code. While the domain model can be used in other layers, the opposite is not true. To maintain an architecture that scales, adhering to structure becomes important. We try not to let persistence-related framework code to proliferate into the other layers. This includes the objects used to hold the data before and after it is persisted. These objects are called data transfer objects. In order to ensure this happens, data is transferred or mapped from the DTOs to the domain model (see next slide).

# Repository Pattern Using SQLAlchemy

```
class TeamStore(Store):
    db_url = Path(__file__).parents[3] / 'resources/course_data.db'
    db = create_engine('sqlite:/// ' + str(db_url), echo=True)
    def connect(self):
        try:
            return sessionmaker(bind=TeamStore.db)()
        except SQLAlchemyError as err:
            raise StoreException('Error connecting to DB.')

    def add(self, team: Team):
        try:
            with self.connect() as session:
                teamDTO = TeamDTO(team.common_name, team.country)
                session.add(teamDTO)
                session.flush()
                session.refresh(teamDTO)
                id = teamDTO.id
                session.commit()
            except SQLAlchemyError as err:
                raise StoreException('Error adding object.') from err
            return id
```

Domain object

Transfer data to DTO, persist it

```
class Store(abc.ABC):
    @abc.abstractmethod
    def connect(self):
        pass

    @abc.abstractmethod
    def add(self, model):
        pass

    @abc.abstractmethod
    def get(self, id):
        pass
```

ch02\_domain/repository\_sqlalchemy/repository/team\_store.py

This version of our repository uses SQLAlchemy for the persistence operations. This means data model objects don't work directly with the solution. Instead, a DTO is used.

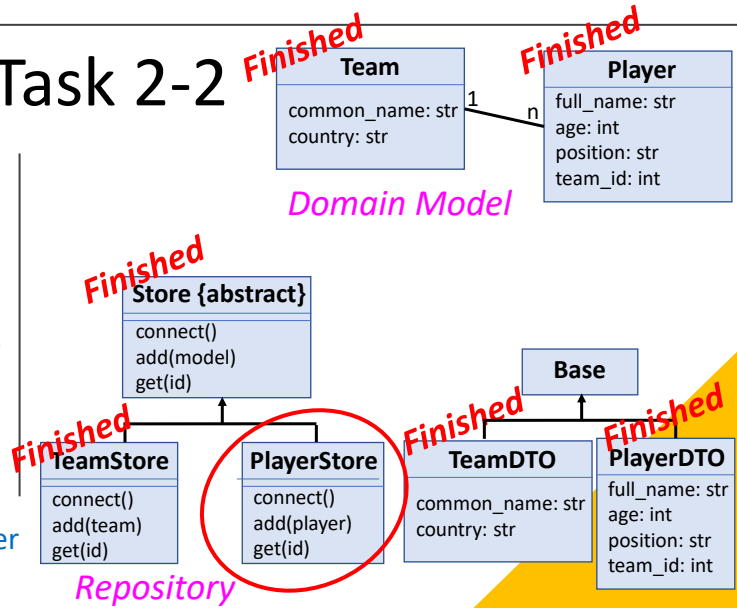
This line:

```
teamDTO = TeamDTO(team.common_name, team.country)
```

maps the Team domain object data into a TeamDTO object. A TeamDTO is specific to SQLAlchemy.

## Your Turn! - Task 2-2

- Refactor the *PlayerStore* using SQLAlchemy
- Team, Player, TeamStore, and the DTOs have been completed already
- Work from [ch02\\_domain/task2\\_2\\_starter](#)



Optionally, work from the exercise workbook in the back of the student manual.

## Data Mapper Pattern

- Our repository manually transfers data, but a *data mapper pattern* can be used
- SQLAlchemy provides a built-in (imperative) mapper

```
from sqlalchemy import Table, Column, Integer,
                        String, ForeignKey
from sqlalchemy.orm import registry

mapper_registry = registry()

team_table = Table('teams',
                    mapper_registry.metadata,
                    Column('id', Integer, primary_key=True),
                    Column('full_name', String(50)),
                    Column('age', Integer),
                    Column('position', String(12)),
                    Column('team_id', Integer))

class Team:
    pass

mapper_registry.map_imperatively(Team, team_table)
```

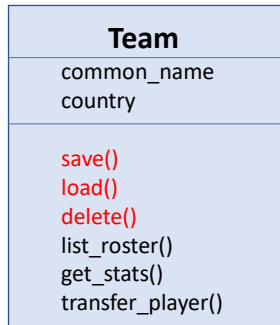
This can be a domain class with business logic and no persistence metadata is added by us

For Task 2-2, we specified how to transfer data from the DTO to the domain object and back again. Worth noting that SQLAlchemy provides a way to map values automatically via a *data mapper*. The mapper allows for taking a regular domain object and then defining how to map table columns to the object.

In the example, our Team class represents a plain domain object (it is essentially empty). *map\_imperatively()* takes the table definition and applies it to the Team class for us. The Team class could be a business model object that has application methods within it. It can now be used in persistence operations because the table definitions have been applied to it, but it hasn't been physically mixed with the code directly.



## Other Persistence Patterns



- **Active Record Pattern**

- Mixes domain logic with persistence metadata
- Active record object contains both data and application behavior

Other patterns exist for managing persistence information. One, called Active Record, allows for the persistence methods to exist within the business objects. A common example of this approach can be found within Ruby on Rails model objects. Django, a Python web application framework, is another example where the active record pattern is applied. This is called Django ORM.

## CH 2 SUMMARY

- ▶ In a layered architecture, the domain model is core to the application
- ▶ A repository pattern will help ensure that persistence-related code stays local to that layer
- ▶ SQLAlchemy is Python's most popular ORM tool. It provides a repository pattern via the session



# Service Layer

# Services Overview

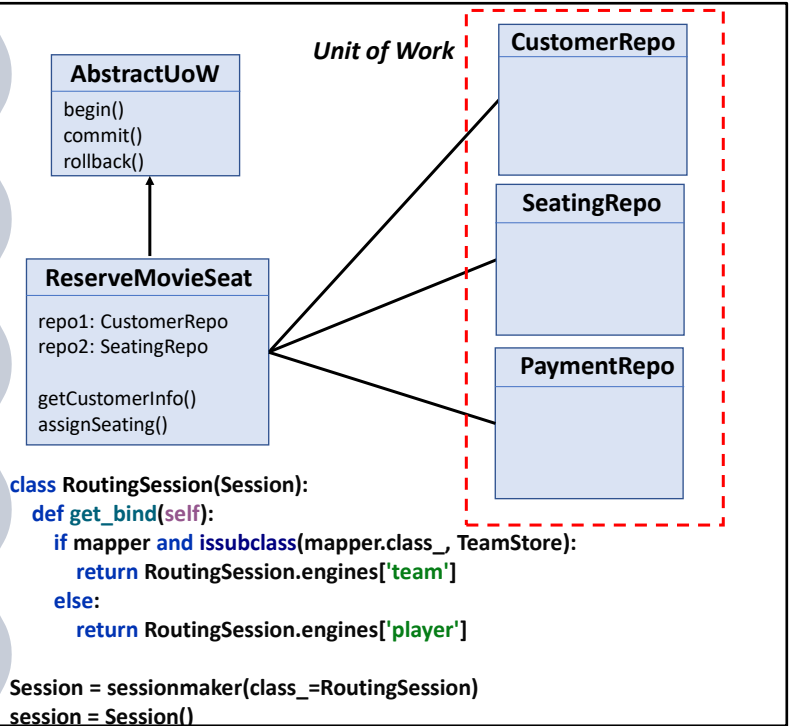
---

- Unit of Work
- Services Layer
- Other Patterns found in Applications
  - Chain of Responsibility
  - Sentinel Value Pattern
  - Singleton
  - Global Object
  - Factory Method
  - Façade
  - Proxy
  - Adapter

# Unit of Work

- The **Unit of Work** (UoW) pattern defines multiple coordinated steps that may be performed against one or more repositories
- A strict UoW tracks changes made throughout the interaction and then updates at the end
- A UoW doesn't have to be strictly tied to a single transaction

SQLAlchemy can manage these objects as a single session and commit or rollback all changes made



ch03\_service/unit\_of\_work/unit\_of\_work.py

The *unit of work* pattern is often tied to a single transaction that may involve multiple repositories. Often the unit of work will either succeed entirely or fail entirely if one part of the interaction fails.

Two ways to implement a unit of work would be to define the UoW in the repository layer, or to define it at the service layer and use repositories within the UoW. A common way to implement the UoW is to track changes made during the interaction and then perform all of the updates at the end. SQLAlchemy's session object does this for us internally.

An example of a UoW would be an online movie reservation. During this interaction, you may be asked to provide your customer information. You may later define your seat selections. Finally, you may be asked to provide payment details. Along the way, if any step fails or is canceled, we want to ensure all steps are rolled back (such as the releasing of the selected seats). SQLAlchemy's session can track the Customer repository object changes as well as the TheaterSeating repository object in the same session. It would rollback any changes made to any of these objects during the unit of work.

While a UoW is commonly applied to a single transaction, that doesn't have to be the case. It could merely involve multiple steps that are outside of any transaction. Also, repository and UoW are separate patterns. The use of one pattern doesn't require the use of the other.

## Creating a Services Layer

- The *Services Layer* is responsible for orchestrating multiple tasks to achieve higher-level logic
- For example, perhaps we have a business requirement to notify customers of our latest team roster
- This task might involve building the roster, preparing it, and sending it in an email

`def publish():`

```
# acquire team roster data  
# prepare it  
# generate PDF of it  
# attach into an email  
# send it
```

We'll use a *Unit of Work* for this step

These can be provided by other services

A task for our Team-Players application is to publish the roster. Doing this will involve several steps: getting the player info, preparing it for rendering, rendering it into an output format (PDF in this case)

## Your Turn! - Task 3-1

- Implement the **Unit of Work** Pattern and complete the *publish()* service for generating and sending the team roster
- Work from **ch03\_service/task3\_1\_starter**

```
def publish():  
    acquire_roster()  
    # later:  
    # generate PDF from it  
    # attach it into an email  
    # send it
```

Only *acquire\_roster()* within the *services/acquire\_roster* module needs to be modified

Unit of Work

```
graph LR  
    A[acquire_roster()] --> B[Identify the Team ID]  
    B --> C[Get the Players for that team]
```

Identify the Team ID

Get the Players for that team

## Chain of Responsibility Pattern

- Chain of Responsibility is a GoF behavioral pattern that applies various tasks one after the other
- CoR avoids coupling the sender of a request (task) to its destination
  - It allows flexibility in adding/removing handlers
- Can assist with pipelines or workflows
  - Example: data cleansing where values need to be imputed, encoded, and then scaled

Can a *Chain of Responsibility* Help?

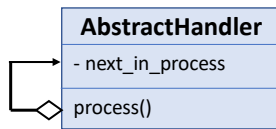


Chain of responsibility allows a "sender" of the task or request to perform a series of work items (a pipeline or workflow) on it and then onto the final destination. The final destination generally does not know about the intermediate steps that the task went through.

Why not just implement a list of handlers instead? While this is certainly possible, a list would not be very dynamic. CoR would have the ability to conditionally call the next element based on internal checks. This would be trickier with a list of handlers since the list will just always continue processing the next item in the collection. The handlers are managed externally (by the list) and thus logic determining whether the list would continue to the next handler would likely be found outside of the handler (somewhere within the application).

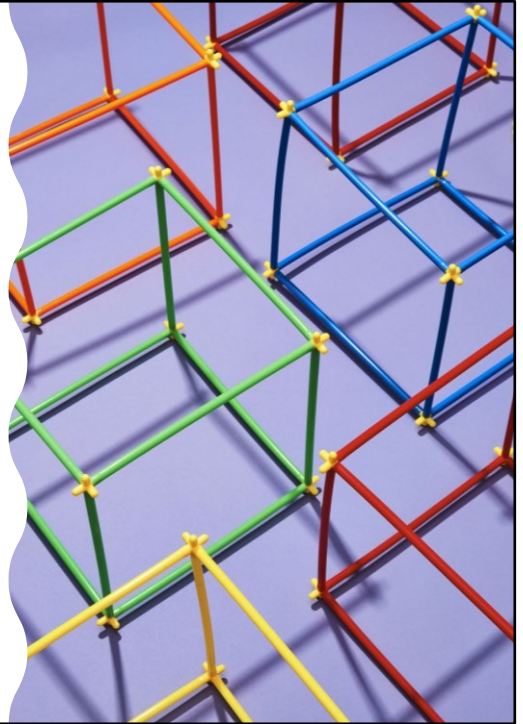


## Chain of Responsibility



Each ConcreteHandler contains a reference to the the next one in the chain

ConcreteHandlers should be able to be used in any order desired



The key to this pattern is that each ConcreteHandler should call the `process()` method of the next handler at the end of the `process()` method. To ensure this happens, the parent could actually do this for us. In order for that to happen, each `process()` method of the concrete classes needs to call `super()` at the end each time (see slide 2 of 3 coming up).

## Data Cleansing Using Chain of Responsibility (1 of 3)

```
import abc

class AbstractHandler(abc.ABC):
    def __init__(self):
        self._next_in_process = None

    @property
    def next_in_process(self):
        return self._next_in_process

    @next_in_process.setter
    def next_in_process(self, next_handler):
        self._next_in_process = next_handler

    @abc.abstractmethod
    def process(self, data):
        if self._next_in_process:
            return self._next_in_process.process(data)
```

The use of a Python *property* makes assignment to the next link in the chain easy

ch03\_service/chain\_of\_responsibility/abstract\_handler.py

Here, our AbstractHandler sets the `_next_in_process` to `None`. This way there will be a next item in the chain, and the last one will always point to `None`. But this means we must ensure that the handlers in the chain before the last one properly set their `next_in_process` or else the chain will be broken.

## Data Cleansing Using Chain of Responsibility (2 of 3)

```
from abstract_handler import AbstractHandler

class Imputer(AbstractHandler):
    def process(self, data):
        data.append('went through Imputer')
        super().process(data)

class DataEncoder(AbstractHandler):
    def process(self, data):
        data.append('went through DataEncoder')
        super().process(data)

class DataScaler(AbstractHandler):
    def process(self, data):
        data.append('went through DataScaler')
        super().process(data)
```

These are the ConcreteHandlers. They do their work and then rely on the AbstractHandler to call the next one in the chain. This requires each handler to call the `super().process()` method each time at the end.

ch03\_service/chain\_of\_responsibility/imputer.py, data\_encoder.py, scaler.py

Each concrete handler overrides the parent `process()` method but must call the parent process at the end or the chain will be broken.

## Data Cleansing Using Chain of Responsibility (3 of 3)

```
def clean_data(data, handler: AbstractHandler):
    handler.process(data)

def create_chain(handlers: List[AbstractHandler]):
    # not shown, merely connects each handler

start = create_chain([Imputer(),
                      Imputer(),
                      DataScaler(),
                      DataEncoder(),
                      DataScaler()])

sample_data = ['This is some starting data']
clean_data(sample_data, start)
print(sample_data)
```

These can be placed in any order, in any quantity

['This is some starting data',  
'went through Imputer',  
'went through Imputer',  
'went through DataScaler',  
'went through DataEncoder',  
'went through DataScaler']

ch03\_service/chain\_of\_responsibility/driver.py

Our chain is created (not within the abstract or concrete classes, but rather, within the driver file.

## Sentinel Value Pattern

- Python uses the sentinel value pattern frequently within its code base
- A sentinel value is a stub that guarantees that a variable will be valid

In Python, *None* is a commonly used sentinel value

```
s = 'A sample string'
print(s.find('sample'))
print(s.find('not there'))
```

2

-1

Sentinel value ensures it won't crash, but rather returns something meaningful to be tested later

```
sentinel = -1
age = int(input('Enter an age (-1 to stop): '))
while age != sentinel:
    print(age)
    age = int(input('Enter an age (-1 to stop): '))
```

The chain of responsibility pattern (previously shown) relied on the *None* sentinel value occurring on the last *ConcreteHandler* in the chain

ch03\_service/sentinel.py

A sentinel value is a common pattern in Python. The Python `find()` method implements a value of `-1` when a substring is not found within the test string. This is a sentinel value and can be used to test against. Numerous instances of sentinel values can occur such as in the second example where the loop continues until the sentinel value is input.

# Sentinels Module

- Sentinels can be any object type
- The third-party *sentinels* module can assist

```
populations = {
    'Cintra': 100_000,
    'Kings Landing': 250_000,
    'Asgard': None,
    'Wonderland': 400_000,
    'Emerald City': None
}
```

```
def find_population(name: str, default=None):
    return populations.get(name, default)
```

```
print(find_population('Emerald City'))
```

None

This returns an actual value of None

```
print(find_population('Emerald Citie'))
```

None

This is not found in the dictionary

```
from sentinels import NOTHING
```

```
def find_population(name: str, default=NOTHING):
    return populations.get(name, default)
```

```
print(find_population('Emerald City'))
```

None

This properly returns the *None* value

```
print(find_population('Emerald Citie'))
```

&lt;NOTHING&gt;

This returns NOTHING to represent the fact the key is not found in the dict

ch03\_service/sentinel.py

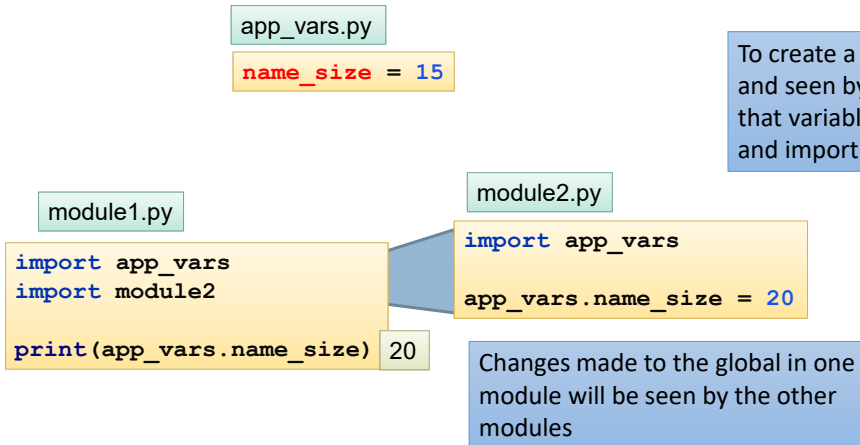
The sentinels third-party module helps with distinguishing between values that are missing and values that are None. We imported the NOTHING object which is used to distinguish between None and a missing value.

NOTHING is not equal to "NOTHING" nor is it equal to None.

# Global Object Pattern

The concept of a "global" variable is a little different in Python

To create a global that can be used and seen by all other modules, put that variable in its own module and import it into the others

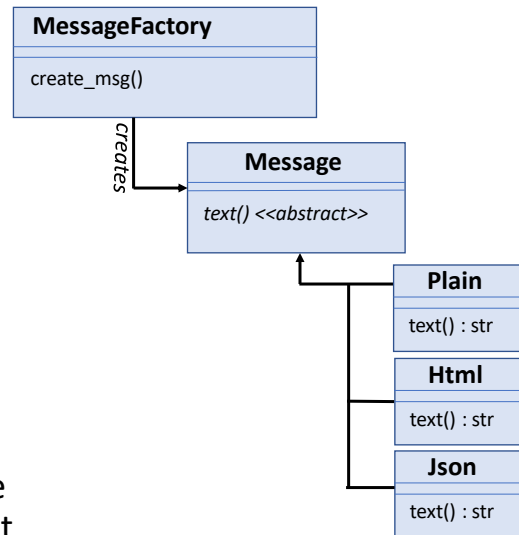


ch03\_service/app\_vars/app\_vars.py, module1.py, module2.py

Generally, to have a variable that all modules can "see," place it in a separate module and then import it into all modules that need it. If one module changes the global variable, all modules "see" this change.

## Factory Method

- The *factory method* pattern is a GoF *creational pattern*
- It attempts to abstract away the creation of classes so that we don't directly hardcode them
- This might be helpful when a large number of class types might need to be created or perhaps later, we might want to create new class types



The factory method is a smaller scale version of the abstract factory (covered next) for creating objects. Its purpose is to instantiate objects without hardcoding them into the source code.



## Factory Method Implemented (1 of 2)

```
import abc
class Message(abc.ABC):
    def __init__(self, msg=''):
        self.msg = msg
    @abc.abstractmethod
    def text(self):
        pass
```

Base class with abstract  
`text()` method

Subclasses define their  
own `text()` methods

```
class Text(Message):
    def __init__(self, msg):
        super().__init__(msg)

    def text(self):
        return self.msg

class Html(Message):
    def __init__(self, msg):
        super().__init__(msg)

    def text(self):
        return f'<span>{self.msg}</span>'

class Json(Message):
    def __init__(self, msg):
        super().__init__(msg)

    def text(self):
        return f'{{ "msg": "{self.msg}" }}'
```

chapter03\_service/factory\_method.py

This pattern defines the *Gang of Four* abstract factory pattern in Python. The Liskov Substitution Pattern says that Text (or Html or Json) Messages should be substitutable by Message and still work. It does in this case, but because Python doesn't care about types, only the presence or absence of attributes (methods in this case) are considered. Therefore, a class called *Duck*, for example, that also contains a `text()` method will work in this example as well.

## Factory Method Implemented (2 of 2)

```
class MessageFactory:
    message_formats = {'text': Text, 'html': Html, 'json': Json}

    @classmethod
    def create_msg(cls, msg_type, msg):
        return cls.message_formats.get(msg_type, Text)(msg)

msg = MessageFactory.create_msg('html', 'Sample HTML msg.')
print(type(msg), msg.get_output())
<class Html> <span>Sample HTML msg.</span>

msg = MessageFactory.create_msg('json', 'Sample JSON msg.')
print(type(msg), msg.get_output())
<class Json> {"msg": "Sample JSON msg."}

msg = MessageFactory.create_msg('text', 'A simple text msg.')
print(type(msg), msg.get_output())
<class Text> A simple text msg.
```

A dictionary provides the possible types that can be instantiated

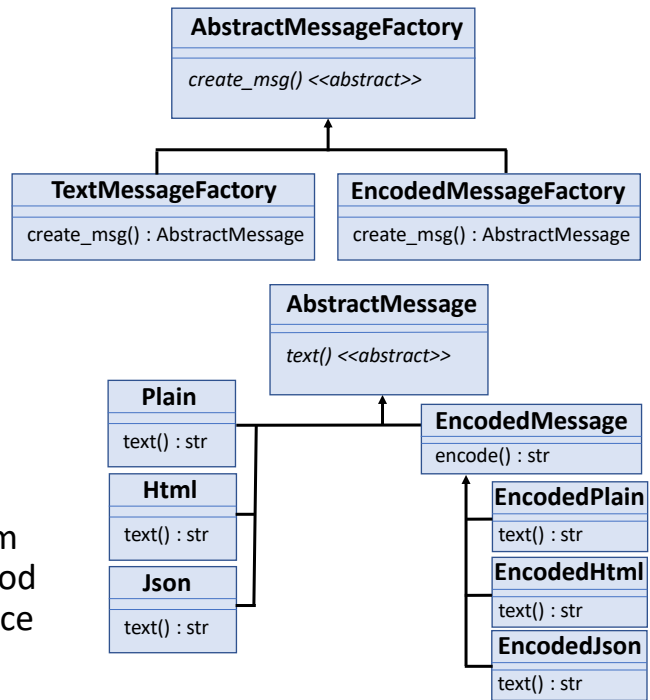
Instantiation

chapter03\_service/factory\_method.py

We made use of our pattern by creating the MessageFactory class. As a test, at the bottom of the slide we instantiate several messages of different types (using the msg\_formats dictionary). create\_msg() is called repeatedly and passed a string: 'html', 'json', or 'text'. This string is used to perform a lookup into the message\_formats dictionary, which in turn, returns the appropriate class type to instantiate.

# Abstract Factory

- An *abstract factory* pattern is a slightly more complicated GoF creational pattern
- It can be used to create objects without hardcoding the concrete class being created
- Due to Python's loose typing system an abstract factory or factory method pattern doesn't always provide a nice architectural fit



ch03\_service/abstract\_factory

The abstract factory pattern is slightly more complicated than the factory method. This pattern allows for swapping out the factory used to create the family of objects. Here, we have two different factories (TextMessageFactory and EncodedMessageFactory) used to create two different sets (families) of messages.

## Abstract Factory Implementation (1 of 3)

```
class AbstractMessageFactory(abc.ABC):  
    @abc.abstractmethod  
    def create_msg(self, msg_type, msg):  
        pass
```

```
class TextMessageFactory(AbstractMessageFactory):  
    message_formats = {'text': Plain, 'html': Html, 'json': Json}  
  
    def create_msg(self, msg_type, msg):  
        return self.message_formats.get(msg_type, Plain)(msg)
```

```
class EncodedMessageFactory(AbstractMessageFactory):  
    message_formats = {'text': EncodedPlain, 'html': EncodedHtml, 'json': EncodedJson}  
  
    def create_msg(self, msg_type, msg):  
        return self.message_formats.get(msg_type, EncodedPlain)(msg)
```

ch03\_service/abstract\_factory/text\_message\_factory.py and encoded\_message\_factory.py

This time there are multiple possible factories that can be used to generate classes. Here, each factory creates different types of classes.

## Abstract Factory Implementation (2 of 3)

```
class AbstractMessage(abc.ABC):
    def __init__(self, msg=''):
        self.msg = msg

    @abc.abstractmethod
    def text(self):
        pass
```

```
class Plain(AbstractMessage):
    def __init__(self, msg):
        super().__init__(msg)

    def text(self):
        return self.msg + '>'
```

```
class Html(AbstractMessage):
    def __init__(self, msg):
        super().__init__(msg)

    def text(self):
        return f'<span>{self.msg}</span>'
```

```
class EncodedMessage(AbstractMessage):
    def __init__(self, msg=''):
        super().__init__(msg)

    def encode(self):
        encoded = base64.b64encode(
            bytes(self.msg, 'utf-8'))
        print(encoded)
        return str(encoded, 'utf-8')
```

```
class EncodedHtml(EncodedMessage):
    def __init__(self, msg):
        super().__init__(msg)

    def text(self):
        return f'<span>{super().encode()}</span>'
```

ch03\_service/abstract\_factory/text\_message\_factory.py and encoded\_message\_factory.py

Some of the classes created by the two factories are shown. On the left is the top class, the AbstractMessage class. Below it are two of the classes created by the TextMessageFactory. On the right are the classes related to the EncodedMessageFactory. The types of classes created are all dependent upon the type of factory used to create each family of message objects.

## Abstract Factory Implementation (3 of 3)

Our factory is used to create messages

```
factory = Factory()

msg_obj = factory.create_msg('html', 'Sample HTML msg.')
print(type(msg_obj), msg_obj.text())

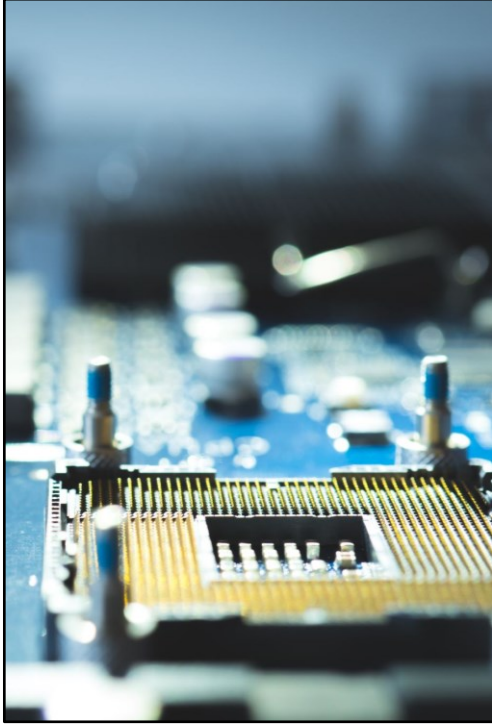
msg_obj = factory.create_msg('json', 'Sample JSON msg.')
print(type(msg_obj), msg_obj.text())

msg_obj = factory.create_msg('text', 'A simple text msg.')
print(type(msg_obj), msg_obj.text())
```

Which factory is this?  
How does it get defined?

ch03\_service/abstract\_factory/driver.py

In the example, `Factory()` could be a `TextMessageFactory` or an `EncodedMessageFactory`. Either one will generate their types of messages. But which factory is that? And how does it get instantiated?



## Python Modules

- When modules in Python are loaded, they are found in a location specified on the **PYTHONPATH** variable
- The file representing the module (e.g., `glob.py` or `os.py`) is opened and read entirely
- With a typical import (e.g., `import glob` or `import os`), the module remains in memory with a variable pointing to it (e.g., `glob` or `os`)
- *A module is only loaded one time no matter how many import statements ask for it!*

## Using *argparse* for Command-line Arguments

- *argparse* can be used to retrieve command-line arguments

```
$ python driver.py
```

```
$ python driver.py TextMessageFactory
```

```
$ python driver.py EncodedMessageFactory
```

We want to specify which factory to use on the command-line

```
def get_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('factory', nargs='?', default='TextMessageFactory')
    return parser.parse_args()
```

```
factory = get_args().factory
```

This will be a string of the name of the factory to load (e.g., "TextMessageFactory")

ch03\_service/abstract\_factory/driver.py

The chosen factory to use has been specified on the command-line. Argparse is used to retrieve the parameters from the command-line as shown. The factory variable will be a string representation of the factory to load for the abstract factory pattern.



## Dynamically Loading the Factory

- It is now necessary to take the command-line provided factory name and load it as a class

```
factory = get_args().factory
mod_names = re.findall(r'[A-Z] (?:[a-z]+|[A-Z]*(?=[A-Z]|$))', factory)
mod_name_str = '_'.join(mod_names).lower()
mod = importlib.import_module(mod_name_str)
Factory = getattr(mod, factory)
```

"TextMessageFactory"

['Text', 'Message', 'Factory']

text\_message\_factory

text\_message\_factory (module)

TextMessageFactory (class)

Now we can load  
this module

Now we can load  
the class

ch03\_service/abstract\_factory/driver.py

This last interesting code snippet found in the abstract factory's driver.py file illustrates how to take an item from the command-line and turn it into actual code that can be executed. In this case that code will become the name of the factory to load. In order to do this, we first bring in the string name of the factory from the command-line using argparse. Then we changed the name into a format to look for the module name (our module will follow the pattern `text_message_factory.py`). Using `importlib.import_module()` the module is dynamically loaded and finally `getattr()` is used to get the class from the module.

# Singleton

- The best pattern for a singleton in Python is to use the module
- If a class must be used (perhaps for inheritance reasons), the following can work

```
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = object.__new__(cls)
            cls._instance.args = args
            cls._instance.kwargs = kwargs
            return cls._instance

    def __init__(self, value: int = 0):
        self.value = value

s1 = Singleton(10)
print(id(s1))

s2 = Singleton(20)
print(id(s2))

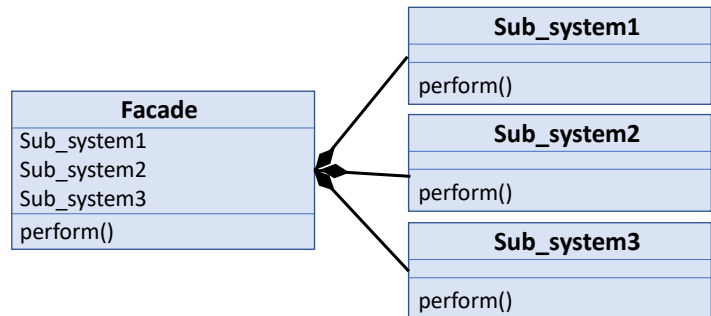
print(s1.value)
```

ch03\_service/singleton.py

Python singletons exist naturally via the module. Modules in Python are only loaded once and then not again no matter how many times an import statement is encountered during the running of that script. One limitation of the module as a singleton, however, is that modules don't support inheritance. So, if inheritance is desired in the singleton, a class should be used, as shown above.

## Facade

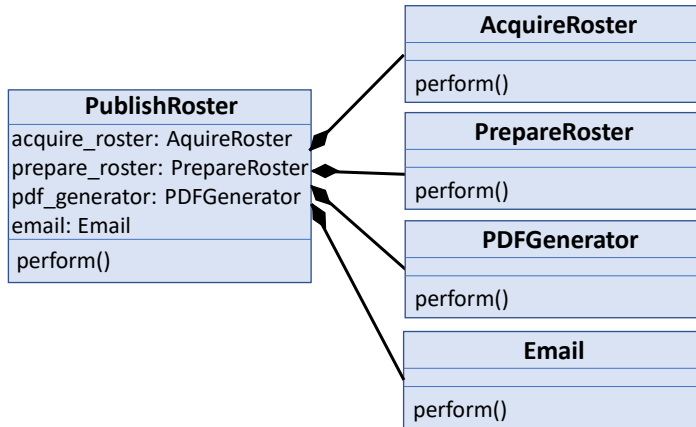
- *Façade* is a GoF *structural* pattern provides a simplified interface to a complex system that may be overly coupled



The role of the façade is to provide a simplified interface to the user. The problem without the Façade is that the user (client) code becomes more complex, and the user must become aware of many subparts of the system. If the user is remote, this can become a performance bottleneck.

It takes a complex system or set of tasks and presents them to the user in a simplified manner. The façade does not need to inherit from an abstract class, nor implement any special methods. The contained classes do not either.

## Façade Implemented (1 of 2)



ch03\_service/façade

Our façade implementation provides a `publish()` method that attempts to connect several services together. This pattern doesn't require the services to implement a common interface (though it could). But the point of it is to simplify the details the user sees, hiding the ugliness of the concrete classes behind the façade. This pattern doesn't attempt to decrease coupling.

## Façade Implemented (2 of 2)

```
class PublishRoster:
    def publish(self, team_name: str,
               filename: str = 'roster.pdf') -> None:
        players = []
        try:
            players = AcquireRoster(team_name).perform()
        except StoreException as err:
            print(f'StoreException--> {err}', file=sys.stderr)

        players_prepped = PrepareRoster(players).perform()
        print('Roster before PDF Gen:')
        print(players_prepped)

        PDFGenerator(filename).perform(players_prepped)

        email = Email('joe@yahoo.com', 'sally@gmail.com',
                     'Team Roster', filename)
        email.perform()
```

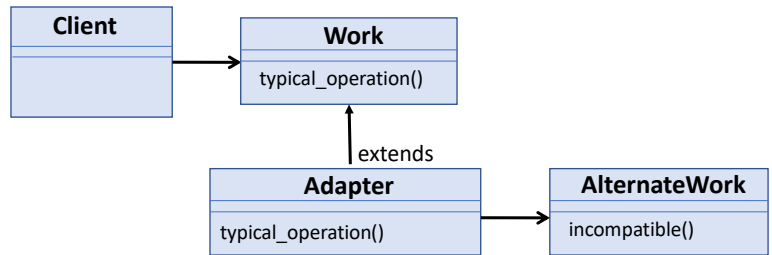
ch03\_service/façade

Here, the following is implemented:

- Acquire the data from the UoW and repository
- Prepare the roster (converting it to a list of lists from a list of players) for ReportLab
- Generate the PDF
- Email it

# Adapter

- **Adapter** is a GoF *structural* pattern that creates a bridge between to incompatible interfaces
  - Analogy: a US vs European plug adapter

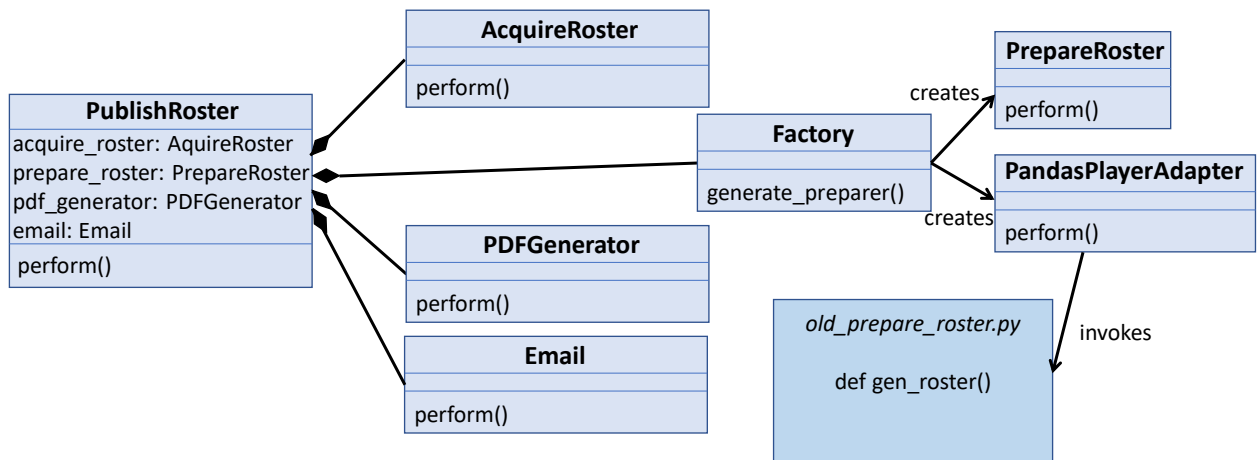


The role of the adapter is to change the implementation interface to the client rather than change the client. Imagine a tool that requires JSON as an input, but you have data generated in XML format. This will require either rewriting the client or creating an adapter that can convert the XML to a JSON format.

In this pattern, the adapter does not have to inherit from Work. Within the adapter's `typical_operation()` method, the `incompatible()` method would be invoked, results would then be translated to a format suitable for the client.

As an aside: a proxy pattern (a GoF structural pattern) is very similar to the adapter. The difference is the adapter may not be related to the object it is adapting (the adaptee) while the proxy typically inherits from the same abstract class as the object it is standing in for. Both cases typically wrap the object they are replacing and call that object's methods. The proxy will typically contain a method with the same name as the one it is standing in for due to the inheritance requirement.

## Adapter Implemented (1 of 2)



ch03\_service/adapter

This class diagram shows the publish roster interaction once again. This time however, a factory chooses the roster implementation to use. There are two options: **PrepareRoster** or **PandasPlayerAdapter**.

It can use the **PrepareRoster** class or a completely different way which uses a function called `gen_roster()`. But `gen_roster()` doesn't work within the client, so an adapter was created, called **PandasPlayerAdapter**. **PandasPlayerAdapter** invokes the `gen_roster()` function but supports the client calling it. This makes the **PandasPlayerAdapter** cause the `gen_roster()` function to work with the client now.

## Adapter Implemented (2 of 2)

```
preparer = PrepareFactory().generate_preparer('pandas', team_name)
players_prepped = preparer.perform()
```

Replace this line to create a different Preparer:

```
preparer = PrepareFactory()
.generate_preparer('current',
players)
```

```
class PandasPlayerAdapter:
    def __init__(self, team_name):
        self.team_name = team_name

    def perform(self) -> List[List]:
        df = adaptee.gen_roster(self.team_name)

        header = list(df.columns)
        results = df.values.tolist()
        results.insert(0, header)

        return results
```

```
def gen_roster(team_name: str)
    ...
    return pd.read_sql(sql_query,
        conn, params=(team_id, ))
```

ch03\_service/adapter

This code illustrates the adapter example showing the PandasPlayerAdapter invoking the incompatible gen\_roster() function.

Separate from the adapter, an abstract factory is used (a factory method could easily have been used instead) to decide which implementation to use: either the PrepareFactory or the PandasPlayerAdapter class.





## CH 3 SUMMARY

Unit of Work can be considered a repository pattern but also a service layer pattern depending on its granularity

A chain of responsibility can implement a form of pipeline to help achieve a flexible workflow

Use global variables and sentinel values in Python according to the way this language prefers



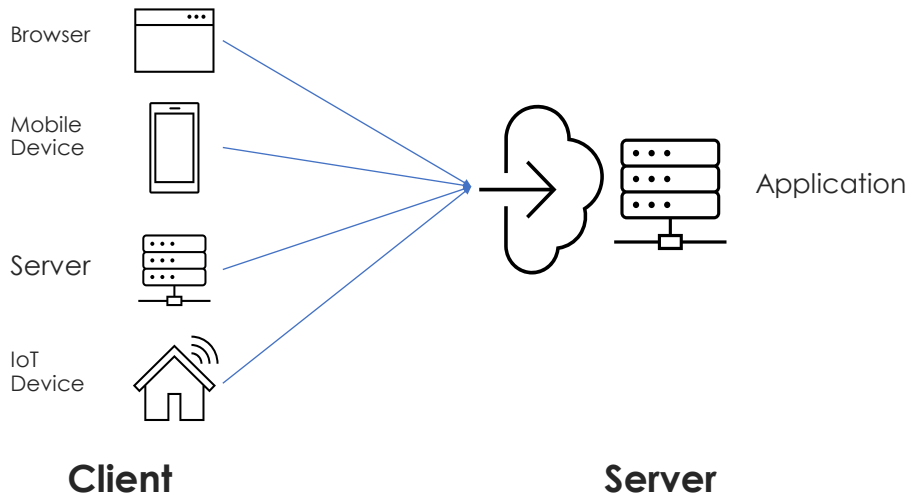
# High-Level Architecture Patterns

# High-Level Architectures Overview

---

- Restful Architectures
- Layered Architectures Disadvantages
- Clean Architecture
- Microservices

## RESTful API Architectures



AN API stands for Application Programming Interface. It allows various devices to make requests to the server and receive a response in return (usually in the form of JSON, though this is not a requirement of REST). This request-response cycle is performed between a client and a server. The client can often be any application including browsers, apps on a mobile device, other application servers, or firmware running on an IoT device.

## Key Components of a RESTful Service

- URIs map to *resources*

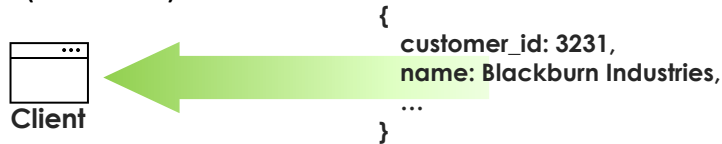


- Manipulation of resources via *HTTP methods*

GET  
POST  
PUT  
PATCH  
DELETE

`http://myserver.com/api/customers/3231`

- Transfer of state (*stateless*)



RESTful APIs have a few important rules that must be followed. Chief among these is that they are modeled around resources. Resources are mapped to by URLs (these URLs are URIs in that they are unique identifiers as well, so may hear the term URI used).

A customer, for example, could represent a resource and there would be a unique URL that could be used to identify that resource. HTTP methods are used to manipulate that resource. This is done by the server receiving the RESTful HTTP request which will indicate an HTTP method such as GET, POST, PUT, and DELETE. The server typically responds with JSON formatted data to the client.

Resource vs. endpoint:

Resource - refers to the noun object(s) being served (e.g., Customer, Celebrity, etc.)

Endpoint - refers to the location (the path) to reach that resource

## RESTful Design Principles

- Design the API around resources (nouns)

By placing the resource in the *path* part of the URL, this is a proper way to map a resource

`http://myserver.com/api/customers/3231`



`http://myserver.com/api/getCustomer/3231`



URL mappings should not be "actions" (verbs)

`http://myserver.com/api/customer?id=3231`



Resource IDs should not be in the query string of the URL

Resources should map to a unique URI (like `http://myserver.com/api/customers`). Resources are usually domain or business objects from your application.

To make the API sound (read) consistent, generally prefer the use of plural nouns.

## RESTful Design Principles *(continued)*

- Use HTTP methods to define operations

	<b>GET</b> /api/customers/3231	Returns a specific customer
	<b>PUT</b> /api/customers/3231	Updates a specific customer
	<b>PATCH</b> /api/customers/3231	Partially Updates a specific customer
	<b>DELETE</b> /api/customers/3231	Deletes a specific customer
<hr/>		
No specific identifier included	<b>POST</b> /api/customers	Creates a new customer
	<b>GET</b> /api/customers	Retrieves all customers (potentially)

Think of a RESTful request as being similar to a sentence. The HTTP method is the "verb" of the sentence while the resource is the noun.

(from above) "GET (verb, HTTP method) the invoice 2952 (noun, resource)."

As previously mentioned, we tend to utilize the plural form of our resource in the URL.

The PATCH method is used when you only wish to update one or two (or so) fields of an object. In doing so, you may PATCH by only sending the required fields that need updating. Generally, a PUT will "resend" the entire object, performing an overwrite of the entire object, which may overwrite fields you didn't intend to update (and may accidentally change values that you didn't intend to change). More is discussed with PATCH a little later.



# Flask

web development,  
one drop at a time

## Introducing Flask

- Flask is a top Python tool for building RESTful applications

<https://flask.palletsprojects.com>



- Since it is third-party, it must be installed

`pip install Flask`

```
from flask import Flask
```

```
app = Flask(__name__)
```

Defines the main Flask application object

```
app.run(host='localhost', port=8051)
```

Starts the server

For more on Flask, visit <https://flask.palletsprojects.com/en/2.0.x/>.

Flask is a commonly employed tool within Python for building APIs. Django and FastAPI are also tools used within Python. Choice of which one to use is a longer topic and will be saved from this discussion.

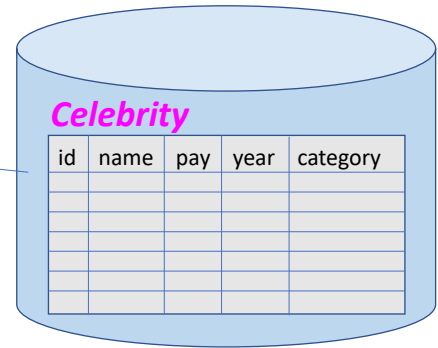


## Flask as a Services Layer (1 of 3)

This is not Flask-related, purely a SQLAlchemy definition

```
class CelebrityModel(db.Model):
    __tablename__ = 'celebrity'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100))
    pay = db.Column(db.Float)
    year = db.Column(db.String(15))
    category = db.Column(db.String(50))

    def __init__(self, name, pay, year, category):
        self.name = name
        self.pay = pay
        self.year = year
        self.category = category
```



ch04\_architecture/flask\_example/flask\_server.py

We'll integrate our Flask solution with SQLAlchemy. In doing this, we'll create a model, much like was done in chapter 2. The difference here is that we are using a plugin called Flask-SQLAlchemy that allows the two frameworks to interact with each other easily.

Our model class is similar to what we saw in chapter 2 except for the base class (`db.Model`), which is a result of incorporating the plugin.

This class defines how to map to the database.

## Flask as a Services Layer (2 of 3)

```
from flask import Flask, jsonify, Response, abort, request
from flask_sqlalchemy import SQLAlchemy
```

```
app = Flask(__name__)
```

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' + str(db_file)
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)
```

This configures Flask to integrate with SQLAlchemy

```
@app.route('/api/celebrities/id/<id>', methods=['GET'])
def get_one_celebrity(id):
    celeb = CelebrityModel.query.get(id)

    if not celeb:
        handle_no_celebrity_exists_error()

    return jsonify(results=celeb.to_dict(), id=id)
```

This **route** connects the incoming request URL to the `get_one_celebrity()` function

**jsonify()** builds a JSON-based response to send

ch04\_architecture/flask\_example/flask\_server.py

In the first code box, a Flask-SQLAlchemy plugin is imported, the location of the database is specified, and the plugin is handed the Flask app object.

With Flask, routes are created by using a special syntax called a decorator. The Flask `app.route()` decorator is placed above your service function which then links the request URL to that function (it creates a mapping).

## Flask as a Services Layer (3 of 3)

```
celeb_id = 1

r = requests.get(f'http://localhost:8051/api/celebrities/id/356')
results = r.json()
print(results)
```



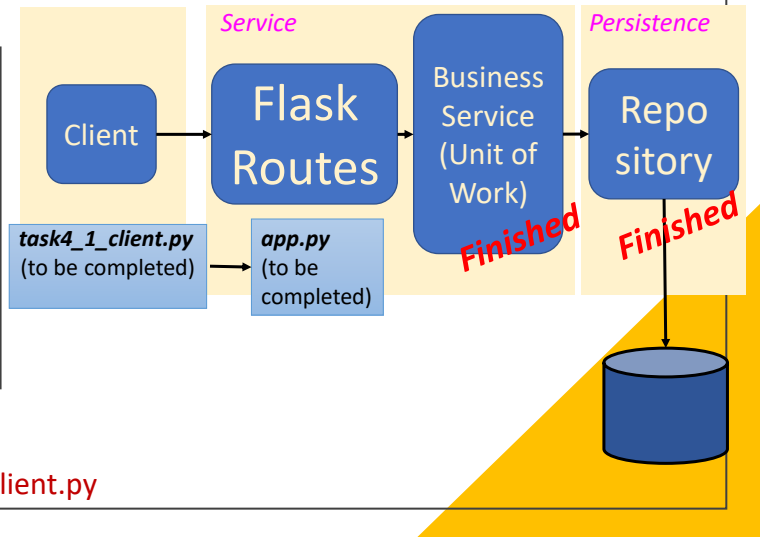
```
{'id': '356', 'results': {'category': 'Athletes', 'id': 356,
'name': 'Kevin Garnett', 'pay': 29.0, 'year': '2008'}}
```

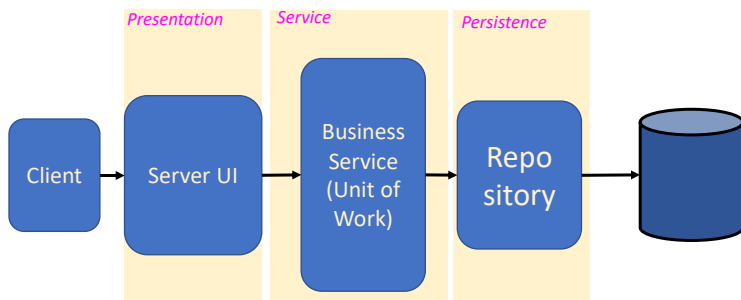
ch04\_architecture/flask\_example/flask\_example/flask\_client.py

Shown above is a client to our Flask app. It uses the requests module to make an HTTP GET request to the server. The server needs to be running to test this.

## Task 4-1

- Implement the Flask services layer to complete the interaction for obtaining the roster
- Work from the provided `ch04_architecture/task4_1_starter` folder
- Work only on:
  - `task4_1_starter/app.py`
  - `task4_1_starter/task4_1_client.py`





### Weaknesses of Classic Layered Architectures

Difficult to follow as it gets larger

Turns into a "monolith"

Changes in one part of the app, require changes in the other layers (lack of scalability)

Layered architectures can promote ease of testability and separation of dependencies from other layers.

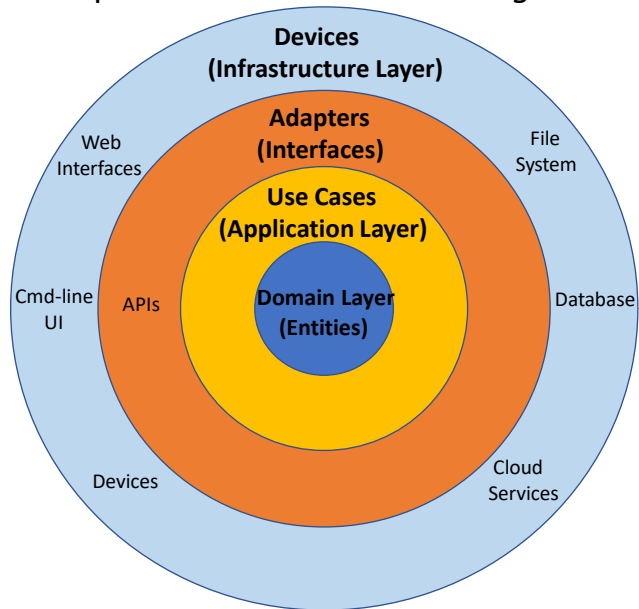
While the layered architecture has been a staple for software system designs for decades, it does present some problems:

1. As systems get more complex, this architecture can become overly large and difficult to follow (even with proper structure applied).
2. It can lead to what is sometimes called a "Monolithic" architecture. In other words, it becomes too big to manage effectively.
3. Changes in one part of the application can lead to the need to change other parts of the application still, and this can take a while when the app is quite large. It is not very scalable.

Several remedies exist that attempt to fix these shortcomings...

### Clean Architecture

- A *Clean Architecture* attempts to make layers less susceptible to the effects of *change*



A clean architecture (defined in 2012 by Robert Martin) is a modern (yet controversial) take on the classic layered architecture. The clean architecture attempts to remedy one of the problems of the classic three-tier layered architecture. Despite attempts to separate concerns, particularly between the data (persistence) layer and business layer, major changes in the data layer can still affect the business layer. The clean architecture tries to abstract these layers even further by imposing some rules. Instead of layers, think in terms of rings. The innermost ring, represented here by our business entities, followed by our use cases, are the most insulated from changes. The outermost rings, which represent your implementation choices, are the most susceptible to changes.

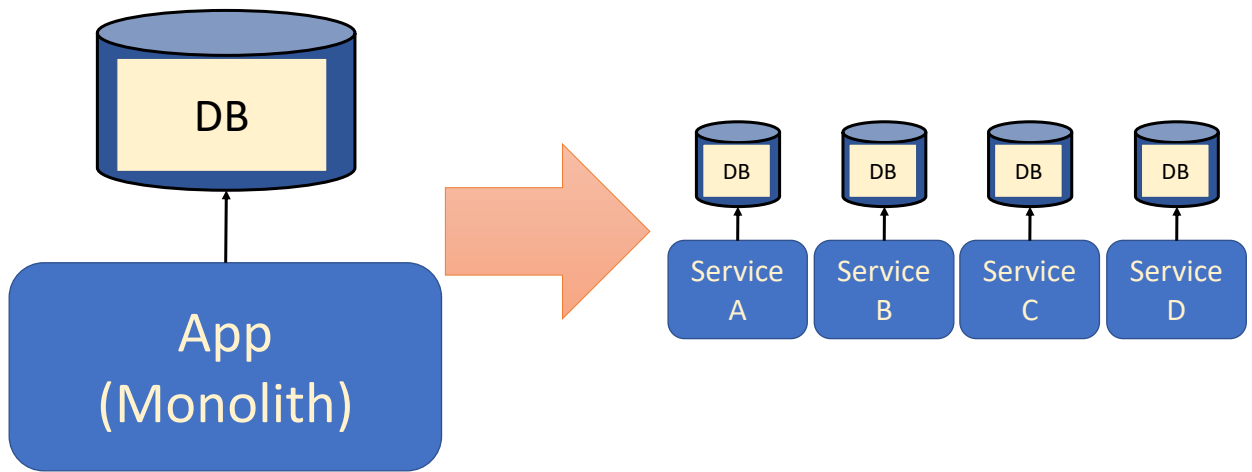
The rules of this architecture state that inner layers do not "know about" outer layers, thus buffering them from changes. Since business entities change less frequently (once established) than infrastructure choices, they are shielded from these changes.

*Use Cases* - abstract the data access layers from business logic

*Adapters* - the logic between your use cases and external dependencies such as databases.

These should not contain any business logic, but instead they only handle the requests from the use cases to the external devices.

## Microservice Architectures

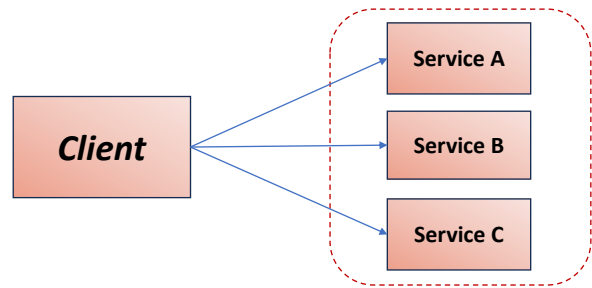


Microservices tackle the issue of a monolith's growing complexity by breaking down the complexity into smaller systems. These smaller systems then become more manageable since they are independent of each other. They own their own data, and they share this data via a smaller (usually RESTful) API.

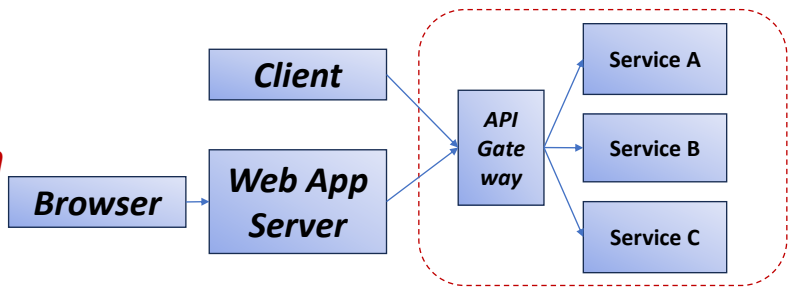
Jeff Bezos is famous for saying, that no meeting should be so large that 2 pizzas can't feed the whole group. Teams developing microservices can be thought of the same way. They are smaller and able to grasp the complexity of their entire system. Microservices are made up of many vertical slices, each potentially implementing their own (version of) layered architecture.

## Client-to-Server Microservice Communication

### *Direct Access*



### *Gateway Pattern*

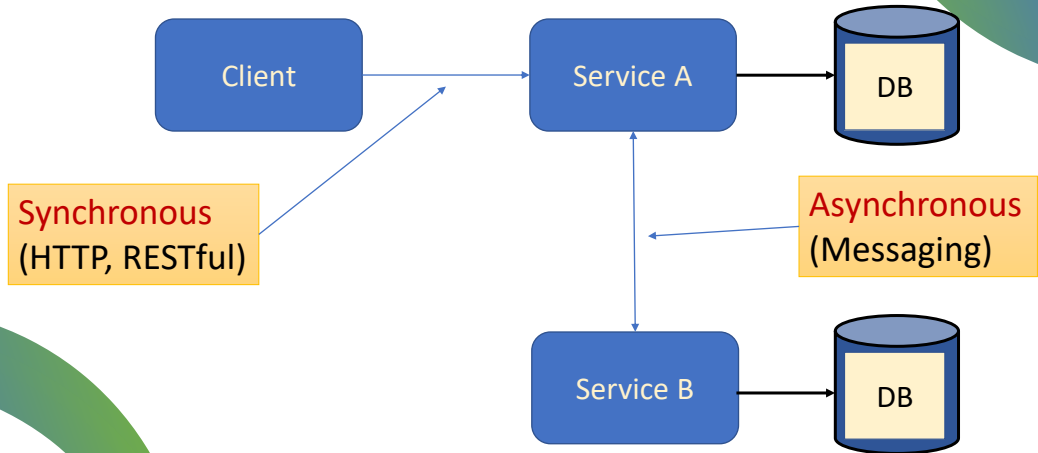


In a microservices architecture, a client may need data from several services. By directly accessing the data from each service several issues may arise. Making requests to each service may require authentication on each server, or latencies may accumulate causing poor performance.

An API gateway can aggregate data into a single response. It is similar to a façade pattern but for microservices. Apigee, Kong, Mulesoft, are some of the available API gateways. While API gateways can help solve some issues, they aren't without problems in other areas. For example, a gateway becomes a single point of failure.



## Communication *Between* Microservices



Microservices typically have a requirement to share data between other microservices. Communication *between* these services can be either synchronous or asynchronous. A common design approach is to let clients (particularly external clients) access the RESTful API interface--the synchronous interface. Synchronous calls must wait for the response to come back before continuing.

The asynchronous approach often involves a form of messaging where the sender of the message usually doesn't wait for a response before continuing. This is the focus of discussions in our last (next) chapter.

## Handling Change in Microservices: API Versioning

Examples of how to handle versioning

`http://apiv2.sample.com/api/celebrities`

`http://localhost:8051/api/v2/celebrities`

`http://localhost:8051/api/celebrities?version=2`

POST /celebrities HTTP/1.1

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)

Host: www.sample.com

Content-Type: application/x-www-form-urlencoded

...

Accept-version: 2

name=Fred%20Savage&pay=3.0&category=Actors

- When microservices change, it can affect the consumers of the service
- **Versioning** helps buffer changes from clients
- Versioning is required when
  - The API returns a *new data format*
  - APIs are *removed*
- Versioning is not required when an *implementation* detail changes
  - E.g., change to the db implementation
  - New resources (endpoints) are added
  - New attributes (properties) are added to the response

Changes within microservices can often be accomplished faster due to the smaller size of the overall system involved. When other applications make use of these services those changes can affect them. One way to deal with change in a microservice is via *versioning*.

Examples above depict ways to use a versioning strategy. URL endpoints that support the new changes can be created so that clients may use the existing, original APIs while the newly updated APIs become available to clients as they update their applications slowly over time.



## CH 4 SUMMARY

Numerous high-level architectures exist that influence the topology for the entire system

RESTful API architectures are easy and popular to implement in Python

Microservices rely on API to allow its different components to successfully communicate



# Event-Driven Architectures

# Event-Driven Architectures Overview

- Event-Driven Architectures
- Mediator
- Observer
- Event Bus
- Microservice Asynchronous Communication

## Event-Based Solutions

- Event-driven architectures are typically *asynchronous*
  - Messages are sent by the sender without waiting for acknowledgement that the receiver got it
- They are useful in:
  - Distributed systems, such as *microservices*
  - Handling overloaded systems
  - Processing jobs and tasks
  - Facilitating communication between UI components
  - In concurrency models

Python's **asyncio** module does this!

Event-driven solutions provide a way for systems to interoperate while at the same time remaining decoupled. They often interact through an exchange of messages, and in some patterns, the sender may not even know who is receiving the messages!

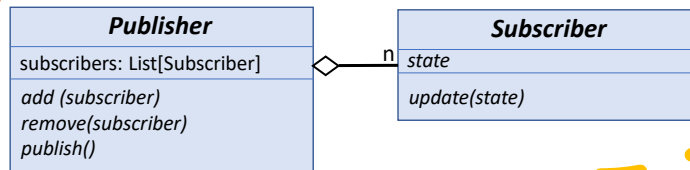
## Patterns and Tools for Event-Driven Solutions

- Patterns used in message-based systems include:
  - Observer
  - Event-Bus
  - Message Broker
- Tools used in message-based implementation:
  - RabbitMQ
  - Apache Kafka
  - Redis
  - Many others

Event-driven solutions are often supported by messaging applications such as Apache Kafka and RabbitMQ. These tools implement the architectures to facilitate communications between distributed systems.

# Observer

- An **observer pattern** facilitates communication with many components by centralizing control and providing **add/remove** subscriber capability
- Sometimes referred to as a **publisher-subscriber** pattern
- Analogy: signing up for a newsletter
  - All those signed up receive it automatically without having to go look for it



An observer pattern is used when an object needs to notify other objects of an event (message). It decouples the sender of a message from the receiver. The pattern is often referred to as a publisher-subscriber (pub-sub) pattern as well.

Though the class diagram doesn't show it, the subscriber can optionally inherit from an abstract class that enforces the behavior of having an `update()` method.



# Observer Implementation

```
class AbstractSubscriber(abc.ABC):
    @abc.abstractmethod
    def __init__(self):
        pass

    @abc.abstractmethod
    def update(self, event: Event):
        pass

class Subscriber(AbstractSubscriber):
    def __init__(self):
        self.state = {}

    def update(self, event: Event):
        self.state = event.state
        print(f'{self.__class__.__name__}: {event}')
```

```
class Publisher:
    def __init__(self):
        self.subscribers:
            List[AbstractSubscriber] = []

    def add(self, observer:
AbstractSubscriber):
        self.subscribers.append(observer)

    def remove(self, observer):
        try:
            self.subscribers.remove(observer)
        except ValueError:
            pass

    def publish(self, event: Event):
        for subscriber in self.subscribers:
            subscriber.update(event)
```

ch05\_event/observer.py

The purpose of the empty, abstract `__init__()` method is to prevent instantiation of that class.

Here, we can see our Subscriber and Publisher class definitions. The Publisher manages a collection of subscribers. Subscribers don't know anything about Publishers or other Subscribers. This communication occurs only in a one-way direction.

## Observer Implementation *(continued)*

```
if __name__ == '__main__':  
  
    publisher = Publisher()  
  
    subscriber1 = Subscriber()  
    publisher.add(subscriber1)  
  
    subscriber2 = Subscriber()  
    publisher.add(subscriber2)  
  
    publisher.publish(Event('The next newsletter has arrived!'))  
    publisher.remove(subscriber1)  
  
    publisher.publish(Event('Another newsletter will be released soon!'))
```

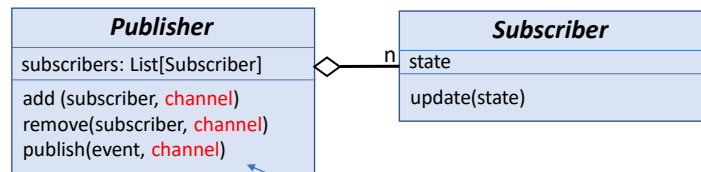
```
Subscriber: The next newsletter has arrived!  
Subscriber: The next newsletter has arrived!  
Subscriber: Another newsletter will be released soon!
```

ch05\_event/observer.py

The main code creates and adds two subscribers to the Publisher. After a while, the first subscriber is unsubscribed causing only one remaining subscriber to receive the state updates.

## Task 5-1

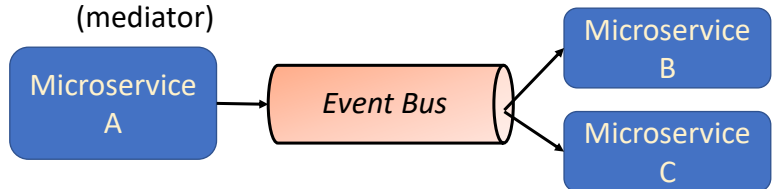
- Modify the Observer pattern example so that it supports communicating over specific channels
- A channel supports different sets of subscribers
- Work from the provided `ch05_event/task5_1_starter.py` file



Add channel support to the *Publisher* by modifying these three methods

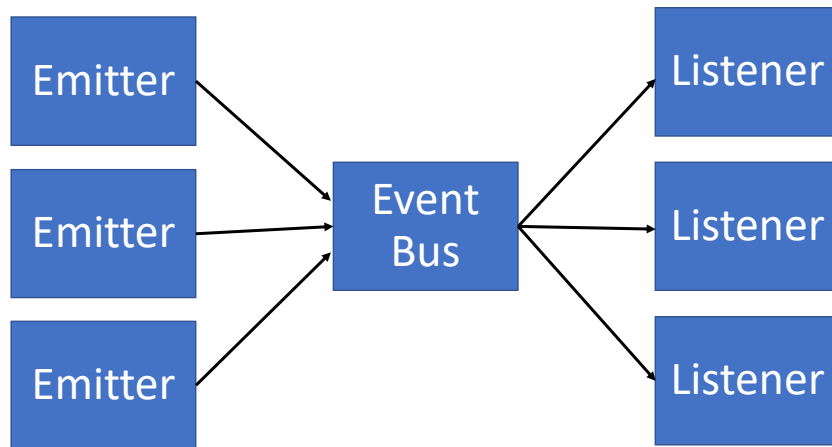
## Event Bus

- The **event bus** allows for pub/sub style communication
- An event bus pattern is a combination of the gang of four *singleton*, *observer*, and *mediator* patterns
  - It is typically implemented as a singleton
  - It tracks subscribers and publishers (like observer) but usually does so on a larger scope (application-level) than a simple observer
  - It manages communication between subscribers (mediator)



An evolution of the observer pattern is an event bus. The event bus typically still implements a publisher-subscriber pattern. It usually provides multiple channels (sometimes called topics) to send messages to/from. Subscribers to a topic will receive the message if they are subscribed.

## Event Bus Implementation In Python



An event bus can be implemented using tools such as RabbitMQ. Python does not have a powerful, distributed event-bus tool. Celery comes close but is more of a job/task queue than a message broker (in fact it typically employs a message broker like RabbitMQ) for messaging services.

This lack of messaging service in Python is notable, but there are Python interfaces for RabbitMQ and Apache Kafka.

A simplified event bus is presented here. This bus *does not* work in a distributed environment, however.

## Event Bus Implementation

- Event bus implementations are somewhat easy to create, but this would incur a lot of reinventing of existing tools
- Python provides numerous small scale event bus implementations including one called `event_bus` (appropriately)

```
pip install event-bus
```

```
from event_bus import EventBus

bus = EventBus()

@bus.on('default')
def subscriber(msg):
    print(msg)

def publish(msg: str, channel: str = 'default'):
    bus.emit(channel, msg)

publish('This is an emitted message.')
```

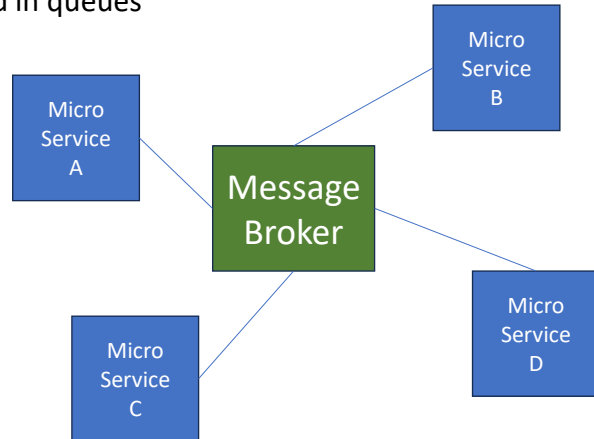
ch05\_event/event\_bus\_framework.py

This framework is best for smaller, single-node implementations. Learn more about it at: <https://github.com/seanpar203/event-bus>

It is not based on Python 3's `asyncio` APIs.

### Message Broker

The **message broker** pattern is typically a centralized *hub-and-spoke* implementation where messages are stored in queues



A message broker is a more centralized system that facilitates communication between microservices by using message queues. The broker is implemented in more of a hub-and-spoke architecture as shown. The broker is the hub in the center of multiple services.

A bus tends to decentralize control over the message management while brokers maintain routing control of messages and can ensure delivery to the correct channels.

## CH 5 SUMMARY

- ▶ Event-driven architectures help decouple components and create more scalable solutions
- ▶ They are particularly important in microservices to allow for communication between services
- ▶ Use an event bus to register unlimited listeners that can receive communications from a source





# Course Wrap-Up



What Did We Learn?

Overview of patterns and architectures

Python idioms

Object-oriented essentials

SOLID Principles

Domain modeling

Type Hints (Annotations)

Dataclasses

Repository Pattern

SQLAlchemy

Data Mapper

Unit of Work

Behavioral Patterns such as

*Chain of Responsibility*

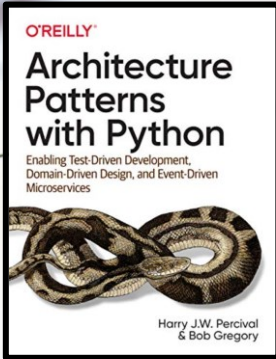
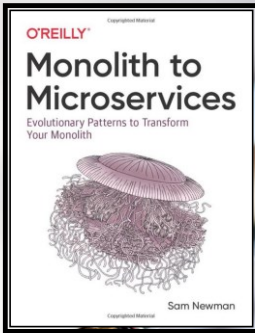
Sentinels and Global variables

Creational patterns such as *Abstract Factory*

and *Factory Method*

RESTful Architectures

Event-based patterns with Observers and Buses



## RECOMMENDED SOURCES

[https://mypy.readthedocs.io/en/stable/cheat\\_sheet\\_py3.html](https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html)

<https://python-patterns.guide/>

<https://martinfowler.com/>

<https://www.cosmicpython.com/book/introduction.html>

<https://packaging.python.org/en/latest/overview/>



# Software Architecture with Python

Exercise Workbook

# Task 1-1

## Setting Up Your Environment



### Overview

This task sets up your development environment by configuring your Python interpreter for use within an IDE. You will also configure the IDE to work with your student files.



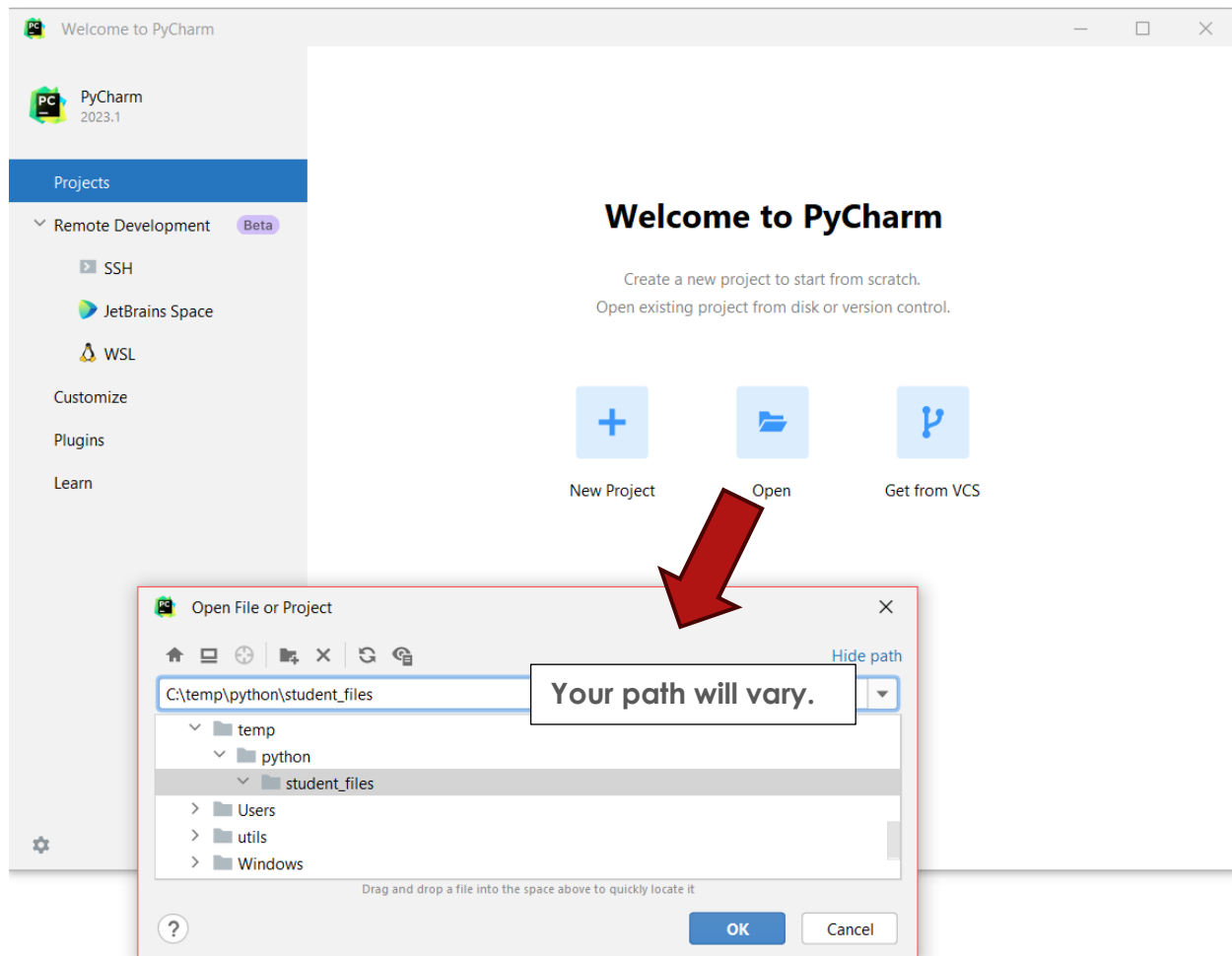
### Launch PyCharm / Create a Virtual Environment

The tool we'll use for working through the exercises is JetBrains PyCharm. PyCharm has the ability create a *Python virtual environment* and *install needed dependencies* (third party packages).

Begin by launching PyCharm.

Ensure that the student files have been placed on your computer and have already been extracted from the original zip file. Take note of the location of these student files.

At the initial screen, choose "Open" and browse to the location of your student files.



When PyCharm starts, one of two things will happen. Follow the approach that matches what happens to you:

1. PyCharm prompts you to create a virtual environment.
2. PyCharm didn't prompt you to create a virtual environment.

### 1. **PyCharm Prompts you to Create a Virtual Environment**

When PyCharm opens, if it prompts you to create a virtual environment do so. It will likely suggest the directory called **venv** within the `student_files` directory. Select **Ok**. That's it. Wait for it to finish.

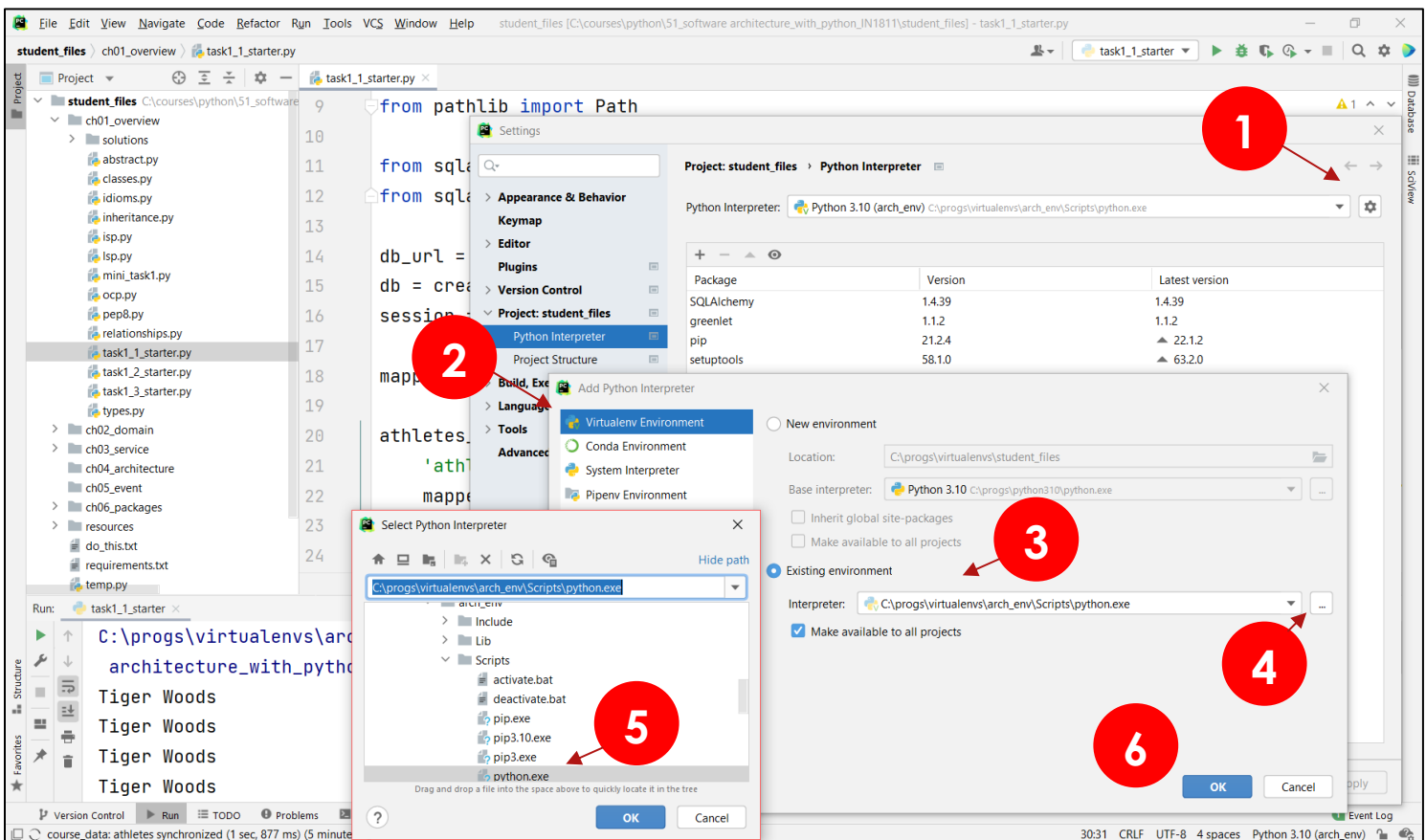
## 2. PyCharm Didn't Prompt you to Create a Virtual Environment

If PyCharm never prompted you to create a virtual environment, you will need to create your own virtual environment manually. Proceed to the *Settings* to establish which interpreter to use.

OS X: **PyCharm > Preferences** Or **PyCharm > Settings**

Windows: **File > Settings**

(Refer to the screenshot below.)

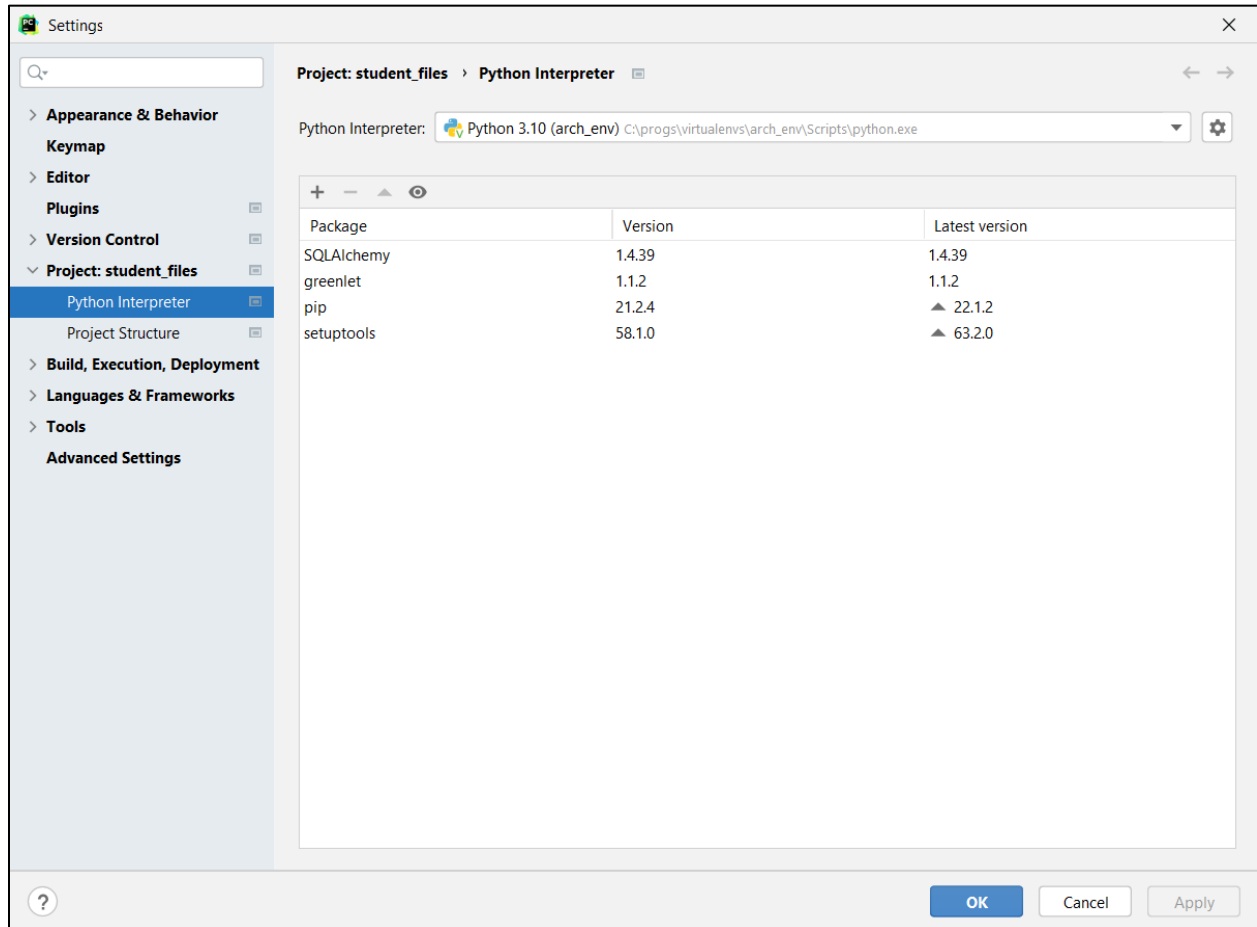


- 1- Select the Add Interpreter... option (or gear) symbol.
- 2- Leave selected "Virtualenv Environment."
- 3- Select the "Existing Environment" radio button.

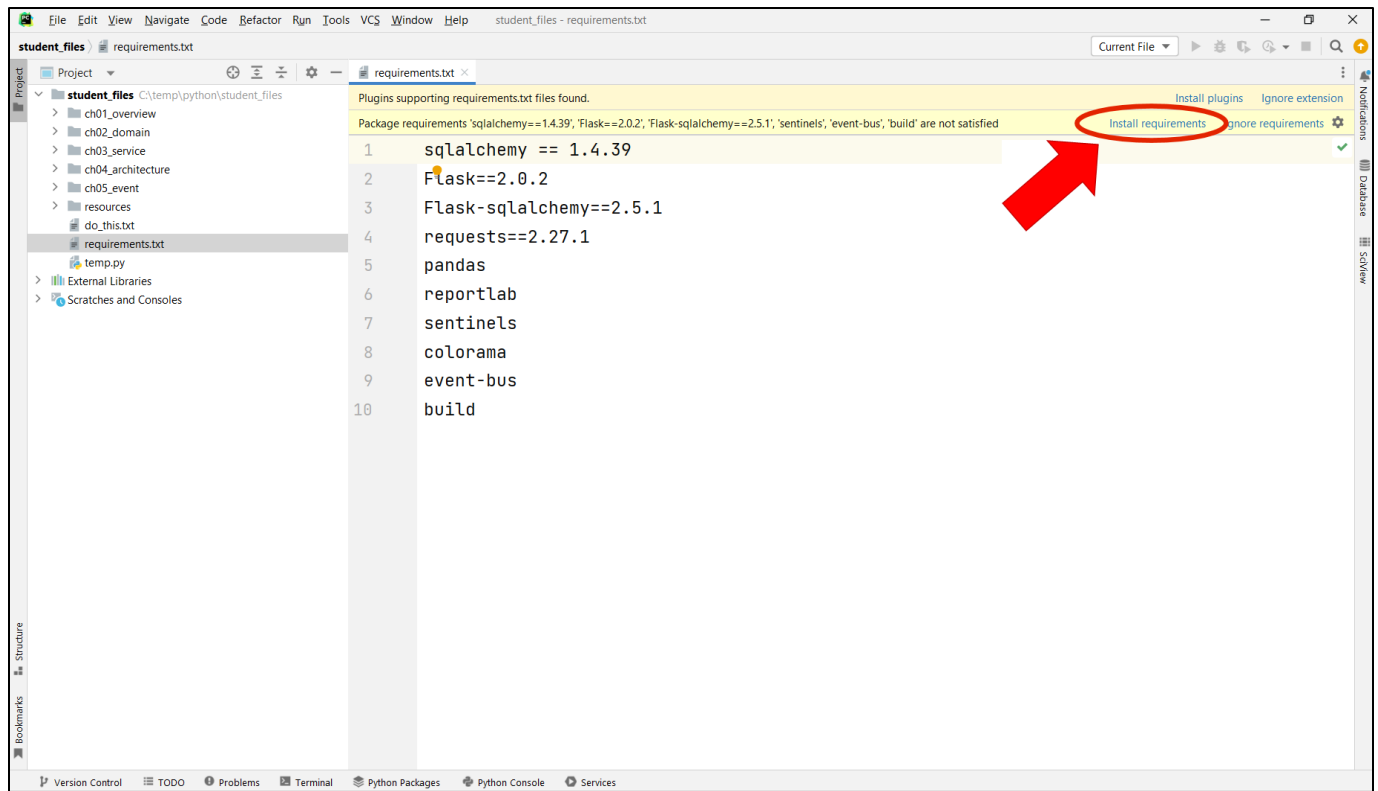


- 4- Select the ... (ellipsis button) (to the right) to find your interpreter.
- 5- Browse to the new Interpreter location (*python.exe* on Windows, *python* or *python3.x* on OS X)
- 6- Click OK twice.

You might see a few packages listed here.



Back in the main window, double-click the file called **requirements.txt**. When it opens, select the option in the yellow bar that says, "**Install Requirements.**" You may have to wait about 30 seconds for it to install the dependencies (refer to screenshot on the next page).



Test out the selected virtual environment by running the file from ch01\_overview called **task1\_1\_starter.py**.

To run it, right-click in the source code and select **Run task1\_1\_starter**.

That's it! If it runs, you're done!

# Task 1-2

## Python Inheritance and Abstract Classes



### Overview

This task provides practice working with inheritance and abstract classes in Python.

Work from the provided starter file, **task1\_2\_starter.py**, found in the `ch01_overview` directory.

In this task, you will create a class hierarchy. At the top of the hierarchy is an abstract Athlete class. A concrete class, called Player, inherits from Athlete. Finally, the Team contains a list of Players (not Athletes).



### Create the Athlete Class

Create an Athlete class. Make it abstract. Provide an `__init__()` method. Make this method abstract. Provide a `full_name` (str) and `age` (int) to be passed and attached to self.

```
class Athlete(abc.ABC):
    @abc.abstractmethod
    def __init__(self, full_name: str, age: int):
        self.full_name = full_name
        self.age = age
```



## Create the Player Class

Create a Player class that inherits from Athlete. Pass in a full\_name, age, team\_id (int), and position (str). Call the base class \_\_init\_\_() using super().

```
class Player(Athlete):
    def __init__(self, full_name: str, age: int, team_id: int,
position: str):
        super().__init__(full_name, age)
        self.team_id = team_id
        self.position = position
```



## Create the Team Class

Create the Team class. Provide a name and id as parameters. Within the \_\_init\_\_() create a roster (an empty list of Players for now).

```
class Team:
    def __init__(self, name: str, id: int):
        self.name = name
        self.id = id
        self._roster: List[Player] = []
```

Add to the class the following two things:

- 1) A property called roster.

```

class Team:
    def __init__(self, name: str, id: int):
        self.name = name
        self.id = id
        self._roster: List[Player] = []

    @property
    def roster(self) -> List[Player]:
        return self._roster

```

2) A method called add\_player().

```

class Team:
    def __init__(self, name: str, id: int):
        self.name = name
        self.id = id
        self._roster: List[Player] = []

    @property
    def roster(self) -> List[Player]:
        return self._roster

    def add_player(self, player: Player) -> None:
        if not any(plr for plr in self._roster if player.full_name ==
plr.full_name):
            self._roster.append(player)

```

That's it!

Clean up any errors. The code at the bottom of the starter file should work with your classes.

Test it out by running the file.

# Task1-3

## Designing for the Dependency Inversion Principle



### Overview

This task will refactor the previous task, modifying it to use the DIP SOLID principle discussed in the notes.

Work from the provided starter file, **task1\_3\_starter.py**, found in the `ch01_overview` directory.



### Create an InjuredPlayer Class

Create an `InjuredPlayer` class that inherits from `Athlete`. Create an `__init__()` that accepts a `full_name`, `age`, `team_id`, and `return_date`. Invoke the parent `__init__()`.

```
class InjuredPlayer(Athlete):
    def __init__(self, full_name: str, age: int,
                  team_id: int, return_date: date):
        super().__init__(full_name, age)
        self.team_id = team_id
        self.return_date = return_date
```



### Modify the Team's add\_player() Method

Change the `add_player()` method to be `add_athlete()`. Have this method accept an `athlete` parameter.

```

class Team:
    def __init__(self, name: str, id: int):
        self.name = name
        self.id = id
        self._roster: List[Athlete] = []

    @property
    def roster(self):
        return self._roster

    def add_athlete(self, athlete: Athlete):
        if not any([ath for ath in self._roster if athlete.full_name
== ath.full_name]):
            self._roster.append(athlete)

```



### Modify add\_to\_roster()

Modify the provided **add\_to\_roster()** method to accept an Athlete object instead of a Player (DIP principle). Change the team.add\_player() call to add\_athlete().

```

    def add_athlete(self, athlete: Athlete):
        if not any(ath for ath in self._roster if athlete.full_name ==
ath.full_name):
            self._roster.append(athlete)

def add_to_roster(team: Team, athlete: Athlete):
    team.add_athlete(athlete)

if __name__ == '__main__':
    bulls_id = 1
    ...

```



## Create an InjuredPlayer Object

Create an InjuredPlayer object. Supply values for it. Example values can be: Scottie Pippin, 28, bulls\_id, and date(1998, 4, 11).

```
if __name__ == '__main__':
    bulls_id = 1
    bulls = Team('Bulls', bulls_id)

    players = [Player('Michael Jordan', 33, bulls_id, 'Guard'),
                InjuredPlayer('Scottie Pippin', 28, bulls_id,
                              date(1998, 4, 11))]

    for plr in players:
        add_to_roster(bulls, plr)
```



## Modify the Type Hint

Locate the self.\_roster line within the Team's \_\_init\_\_() to be of type List[Athlete]

```
class Team:
    def __init__(self, name: str, id: int):
        self.name = name
        self.id = id
        self._roster: List[Athlete] = []
```

That's it!

Test it out to ensure it works!



# Task 2-1

## Implementing the Repository Pattern



### Overview

This task builds an implementation of the repository pattern.

Work from the provided starter folder, **task2\_1\_starter**, found in the `ch02_domain` directory.



### Define a Player Dataclass

Open **models.py**. Create a Player Dataclass containing a `full_name` (str), `age` (int), `position` (str) and `team_id` (int) fields. Optionally, create a `__str__()` method, though what it returns is not critical.

```
@dataclass
class Player:
    full_name: str
    age: int
    position: str
    team_id: int

    def __str__(self):
        return f'{self.full_name} ({self.age}), {self.position}'
```



### Define a Team Dataclass

Below the Player dataclass definition within `models.py`, create a Team model class containing a `common_name` (str) and `country` (str). No need to add an `__str__()` method here.

```
@dataclass
class Team:
    common_name: str
    country: str
```



## Add Abstract Methods to the Store Class

Open **base\_store.py**. Within this module, locate the Store class. Add 3 empty methods: `connect()`, `add()`, and `get()`. Do this as follows:

```
class Store(abc.ABC):
    def __init__(self):
        try:
            self.conn = self.connect()
            ...

    @abc.abstractmethod
    def connect(self):
        pass

    @abc.abstractmethod
    def add(self, model):
        pass

    @abc.abstractmethod
    def get(self, id):
        pass

    def __enter__(self):
        return self

    def __exit__(self, typ, msg, tb):
```

```
self.close()

...
```



## Inherit TeamStore from Store

Open **team\_store.py**. In the class declaration line, have TeamStore inherit from Store.

```
class TeamStore(Store):
```



## Implement connect() within TeamStore

Override the connect() method from the parent. It should invoke and return the sqlite3 connect() method. Pass the class' db\_url variable into it.

```
class TeamStore(Store):
    db_url = Path(__file__).parents[2] /
                'resources/course_data.db'

    def connect(self):
        return sqlite3.connect(TeamStore.db_url)
```



## Override the add() Method in TeamStore

Override the add() method from the parent. The code would normally contain SQL code to INSERT and object into the database.

```
class TeamStore(Store):
    db_url = Path(__file__).parents[2] /
                                     'resources/course_data.db'

    def connect(self):
        return sqlite3.connect(TeamStore.db_url)

    def add(self, team: Team) -> int:
        try:
            cursor = self.conn.cursor()
            cursor.execute(
                'INSERT INTO teams (common_name, country) VALUES(?, ?)',
                (team.common_name, team.country))

            row_id = cursor.lastrowid
        except Exception as err:
            raise StoreException('Error adding team.') from err

        return row_id
```



## Override the get() Method within TeamStore

Override the get() method from the parent. It will accept a team\_id and return a Team object.

```
class TeamStore(Store):
    db_url = Path(__file__).parents[2] / 'resources/course_data.db'

    def connect(self):
        return sqlite3.connect(TeamStore.db_url)

    def add(self, team: Team) -> int:
```

```

...

def get(self, team_id) -> Team:
    try:
        cursor = self.conn.cursor()
        cursor.execute(
            'SELECT common_name, country FROM teams WHERE id = ?',
            (team_id,))
        results = cursor.fetchone()
    except Exception as err:
        raise StoreException('Error retrieving team.') from err

    return Team(*results)

```



## PlayerStore should inherit from Store

Open **player\_store.py**. In the class declaration line, have PlayerStore inherit from Store.

```
class PlayerStore(Store)
```



## Create the connect() method for PlayerStore

Repeat the connect() call that was done in step 5.

```

class PlayerStore(Store):
    db_url = Path(__file__).parents[2] /
                'resources/course_data.db'

    def connect(self):
        return sqlite3.connect(PlayerStore.db_url)

```



## Override add() within the PlayerStore

Have this method accept a Player model object and return the id of the new object inserted.

```
class PlayerStore(Store):
    db_url = Path(__file__).parents[2] / 'resources/course_data.db'

    def connect(self):
        return sqlite3.connect(PlayerStore.db_url)

    def add(self, player: Player):
        try:
            cursor = self.conn.cursor()
            cursor.execute('INSERT INTO players (full_name, age,
position, team_id) VALUES(?, ?, ?, ?)',
                           (player.full_name, player.age,
player.position, player.team_id))
            row_id = cursor.lastrowid

        except Exception as err:
            raise StoreException('Error adding player.') from err

        return row_id
```



## Override get() from the Store

Override the get() method from the Store as done in Step 7. Supply a player\_id to the method to be used in the SQL SELECT statement.

```
class PlayerStore(Store):
    db_url = Path(__file__).parents[2] / 'resources/course_data.db'

    def connect(self):
        return sqlite3.connect(PlayerStore.db_url)

    def add(self, player: Player):
        ...

    def get(self, player_id) -> Player:
```

```

try:
    cursor = self.conn.cursor()
    cursor.execute('SELECT full_name, age, position, team_id
FROM players WHERE id = ?', (player_id,))
    results = cursor.fetchone()
except Exception as err:
    raise StoreException('Error retrieving player.') from err

return Player(*results)

```



## Import the Concrete Store Classes

Open **driver.py**. Add the two imports.

```

from models import Team, Player

from base_store import StoreException
from player_store import PlayerStore
from team_store import TeamStore

```



## Use the PlayerStore

Add a player (or several) to the database via the store.

```
from models import Team, Player

from base_store import StoreException
from player_store import PlayerStore
from team_store import TeamStore

try:
    with TeamStore() as ts:
        team_id = ts.add(Team('Depp', 'Owensboro, KY, USA'))

        with PlayerStore() as ps:
            row_id = ps.add(Player('Iron Man', 12, 'Defender', team_id))

            with PlayerStore() as ps:
                print(ps.get(row_id))
except StoreException as err:
    print(err)
```

That's it!

Clean up any errors and test it out by running the driver.



# Task 2-2

## Repository Pattern and SQLAlchemy



### Overview

This task will implement a repository pattern that uses the popular ORM tool called SQLAlchemy.

In this task, you will complete the `player_store.py` file by implementing the `add()` and `get()` methods.

Work from the provided starter folder, **task2\_2\_starter**, found in the `ch02_domain` directory.



### Override and Complete the `add()` Method

This `add()` method accepts a `Player` object (as before). Open **task2\_2\_starter/repository/player\_store.py**. Implement the `add()` method using SQLAlchemy.

```
class PlayerStore(Store):
    db_url = Path(__file__).parents[3] / 'resources/course_data.db'
    db = create_engine('sqlite:/// ' + str(db_url), echo=True)

    def connect(self):
        try:
            return sessionmaker(bind=PlayerStore.db)()
        except SQLAlchemyError as err:
            raise StoreException('Error getting session.') from err

    def add(self, player: Player):
        try:
            with self.connect() as session:
```

```

        playerDTO = PlayerDTO(player.full_name, player.age,
player.position, player.team_id)
        session.add(playerDTO)
        session.flush()
        session.refresh(playerDTO)
        id = playerDTO.id
        session.commit()
    except SQLAlchemyError as err:
        raise StoreException('Error adding team.') from err

    return id

```



## Override and Complete the get() Method

Within the same file, implement the get() method using SQLAlchemy.

```

class PlayerStore(Store):
    db_url = Path(__file__).parents[3] / 'resources/course_data.db'
    db = create_engine('sqlite:/// ' + str(db_url), echo=True)

    def connect(self):
        ...

    def add(self, player: Player):
        ...

    def get(self, player_id) -> Player:
        try:
            with self.connect() as session:
                playerDTO = session.query(PlayerDTO).get(player_id)
                player = Player(playerDTO.full_name, playerDTO.age,
playerDTO.position, playerDTO.team_id)
            except Exception as err:
                raise StoreException('Error retrieving team.') from err

        return player

```

That's it!

# Task 3-1

## Unit of Work



### Overview

This task will implement the unit of work pattern discussed in the student manual by completing the `publish()` method.

Work from the provided starter folder, **task3\_1\_starter**, found in the `ch03_service` directory. For this task you will only need to complete the `acquire_roster` module using the unit of work pattern.



### Create the `acquire_roster()` Function

Open `ch03_service\task3_1_starter\services\acquire_roster.py`.  
Begin by building the `acquire_roster()` method as shown below.

```
def acquire_roster(team_name: str) -> List:  
    # more will go here in the next 3 steps
```



### Create (Instantiate) a `TeamStore`, Invoke `find_by_name()`

Instantiate a `TeamStore()` object and call its newly created `find_by_name()` method.

```
def acquire_roster(team_name: str) -> List:  
    ts = TeamStore()  
    team_id = ts.find_by_name(team_name)
```



## Instantiate a PlayerStore, Invoke find\_by\_team()

Do the same thing for the PlayerStore, except pass the team\_id (returned from find\_by\_name) into the PlayerStore's find\_by\_team() method.

```
def acquire_roster(team_name: str) -> List:
    ts = TeamStore()
    team_id = ts.find_by_name(team_name)

    ps = PlayerStore()
    players = ps.find_by_team(team_id)
```



## Return the List of Players

Return the list of Players from the method.

```
def acquire_roster(team_name: str) -> List:
    ts = TeamStore()
    team_id = ts.find_by_name(team_name)

    ps = PlayerStore()
    players = ps.find_by_team(team_id)

    return players
```

That's it! Test it out by running driver.py.

# Task 4-1

## Implement the Flask Service Layer



### Overview

This task will enable a services layer in a layered architecture using the Flask application server.

Work from the provided starter folder, **task4\_1\_starter**, found in the `ch04_architecture` directory. You will work with two files: `app.py` and `task4_1_client.py`. Both can be found directly inside the `task4_1_starter` folder.



### Add the `route()` to the `get_roster()` Method

Open `ch04_architecture\task4_1_starter\app.py`. Complete the `get_roster()` method as shown.

```
@app.route('/api/team_roster/<name>', methods=['GET'])  
  
def get_roster(name):  
    pass
```



### Call the `publish()` Service within the Function

Remove the `pass` statement. Call the `publish()` service within the `get_roster()` function.

```

@app.route('/api/team_roster/<name>', methods=['GET'])
def get_roster(name):
    status = 200
    team = publish(name)
    resp = jsonify(team=team, name=name)

    return Response(resp.data, mimetype='application/json',
                    status=status)

```



## Add Exception Handling

Errors should cause StoreExceptions to propagate back to this method. Handle exceptions by setting an error message and returning a 404.

```

@app.route('/api/team_roster/<name>', methods=['GET'])
def get_roster(name):
    status = 200
    try:
        team = publish(name)
        resp = jsonify(team=team, name=name)
    except Exception as err:
        resp = jsonify(team=[], name=err.args[0])
        status = 404

    return Response(resp.data, mimetype='application/json',
                    status=status)

```

Careful adding the exception handling in. You may have to adjust your indenting.



## Write the Client Code

Open **task4\_1\_client.py** and work on writing the client code segment.

```
r = requests.get(f'{base_url}{path}{team_name}')
results = r.json()
print(f'URL: {r.url}')
print(results)
```

That's it! Test it out by running first the *app.py* file and then running the *task4\_1\_client.py* file. You can also type the following URL into the browser:

**`http://localhost:8051/api/team_roster/Arsenal`**

# Task 5-1

## Modify the Observer



### Overview

This task will take an observer pattern and implement it in a way that it supports having multiple channels.

Work from the provided **ch05\_event\task5\_1\_starter.py** file. Only the Publisher class needs to be modified.



### Change the Subscriber List in the `__init__()`

Open **ch05\_event\task5\_1\_starter.py**. Locate the Publisher class. This is the only class that needs to be edited. Modify the `__init__()` to have a dictionary instead of a list for its subscribers, as follows:

```
class Publisher:
    def __init__(self):
        self.subscribers: defaultdict = defaultdict(list)

    ...
```



### Refactor the `add()` Method

Still within the Publisher class, modify **`add()`** to take a second parameter for the channel. Also, revise the one line within this function to support a channel, as follows:



```

class Publisher:
    def __init__(self):
        self.subscribers: defaultdict = defaultdict(list)

    def add(self, observer: AbstractSubscriber,
           channel: str = 'default'):
        self.subscribers[channel].append(observer)

```



## Refactor the remove() Method

Still within the Publisher class, modify **remove()** in almost the exact way we did for **add()** except call **remove()** instead of **append()**. Have **remove()** take a second parameter for the channel. Also, revise the one line within this function to support a channel, as follows:

```

class Publisher:
    def __init__(self):
        self.subscribers: defaultdict = defaultdict(list)

    def add(self, observer: AbstractSubscriber, channel: str =
'default'):
        self.subscribers[channel].append(observer)

    def remove(self, observer, channel: str = 'default'):
        try:
            self.subscribers[channel].remove(observer)
        except ValueError:
            pass

```



## Refactor the publish() Method

Still within the Publisher class, modify **publish()** to take a second parameter for the channel. It should be a str type and have the default value of 'default' as we've seen in steps 2 and 3. Also, revise the for-loop within this method as follows:

```
class Publisher:
    def __init__(self):
        self.subscribers: defaultdict = defaultdict(list)

    def add(self, observer: AbstractSubscriber,
            channel: str = 'default'):
        self.subscribers[channel].append(observer)

    def remove(self, observer, channel: str = 'default'):
        try:
            self.subscribers[channel].remove(observer)
        except ValueError:
            pass

    def publish(self, event: Event, channel: str = 'default'):
        for subscriber in self.subscribers[channel]:
            subscriber.update(event)
```

That's it! Test it out!