Introduction to Intermediate Level Python

Daniel Perrefort
Center for Research Computing
University of Pittsburgh
Today's Outline

1. Writing Object Oriented Python
2. Working with "Magic Methods"
   Break
3. Useful Python Design Patterns
4. Special Decorators for Classes
5. Inheritance
   Break
6. Introduction to SOLID
Writing Object Oriented Python
What is a Programming Paradigm?

A particular style of writing software, often enforced by a programming language.

This is where we will focus today
Procedural Programming

Computational steps are divided into reusable units and the code is executed serially

```python
my_list = [10, 20, 30, 40]

def sum_the_list(data):
    res = 0
    for val in data:
        res += val

    return res

print(sum_the_list(my_list))
```
Object Oriented Programming

Data and logic are combined into **objects** that reflect distinct logical constructs

```python
my_list = [10, 20, 30, 40]

class ListCalculator:
    
    def __init__(self, data):
        self.data = data

    def sum(self):
        return sum(self.data)

instance = ListCalculator(my_list)
print(instance.sum())
```

Numpy arrays and Pandas Dataframes are examples of objects.
Everything In Python Is An Object!
(Even if you are writing "procedural" software)
A big difference between procedural and OO is **encapsulation**:

"... the bundling of data with the mechanisms or methods that operate on the data, or the limiting of direct access to some data, such as an object's components."

- Wikipedia
The class Keyword

The class acts as a template for creating new objects.

Objects are instances of the class. Objects provide access to the data/logic. Classes define what the interface looks like.

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

r1 = Rectangle(4, 5)
r2 = Rectangle(100, 100)
print(r1.area())
```
This example defines three main things:

- The `Rectangle` class
- The `r1` and `r2` instances

```python
class Rectangle:
    width = 5
    height = 6

r1 = Rectangle()
r2 = Rectangle()
```
Encapsulating Data: Class Attributes

Example 1.1:

```python
# Class attributes can be accessed from instances
print(r1.width)
print(r2.width)
> 5
> 5

# Class attributes can also be accessed from the class itself
print(Rectangle.width)
print(Rectangle.height)
> 5
> 6
```

class Rectangle:
    width = 5
    height = 6

r1 = Rectangle()
r2 = Rectangle()
Encapsulating Data: Class Attributes

Example 1.2:

```python
print(Rectangle.width)
print(r1.width)
print(r2.width)

r1.width = 12
print(Rectangle.width)
print(r1.width)
print(r2.width)
```

```python
class Rectangle:
    width = 5
    height = 6

r1 = Rectangle()
r2 = Rectangle()
```
Encapsulating Data: Class Attributes

Example 1.2:

```python
class Rectangle:
    width = 5
    height = 6

r1 = Rectangle()
r2 = Rectangle()

print(Rectangle.width)
print(r1.width)
print(r2.width)
> 5
> 5
> 5

r1.width = 12
print(Rectangle.width)
print(r1.width)
print(r2.width)
> 5
> 12
> 5
```

Editing an instance only changes the properties of that instance.
Encapsulating Data: Class Attributes

Example 1.3:

```python
class Rectangle:
    width = 5
    height = 6

r1 = Rectangle()
r2 = Rectangle()

print(Rectangle.width)
print(r1.width)
print(r2.width)
> 5
> 5
> 5

Rectangle.width = 12
print(Rectangle.width)
print(r1.width)
print(r2.width)
> > >
```

```bash
> 5
> 5
> 5
```

```python
class Rectangle:
    width = 5
    height = 6

r1 = Rectangle()
r2 = Rectangle()
```
Encapsulating Data: Class Attributes

Example 1.3:

```python
class Rectangle:
    width = 5
    height = 6

r1 = Rectangle()
r2 = Rectangle()

print(Rectangle.width)
print(r1.width)
print(r2.width)

> 5
> 5
> 5

Rectangle.width = 12
print(Rectangle.width)
print(r1.width)
print(r2.width)

> 12
> 12
> 12

Editing a class changes the properties of all current and future instances.
```
Encapsulating Data: Instance Attributes

The `init` method is responsible for **instantiating** the class. It is called every time a new instance is made.

The `self` variable is always the first argument in the `init` method. It represents the instance being created.

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

r1 = Rectangle(4, 5)
r2 = Rectangle(100, 100)
```
Encapsulating Data: Instance Attributes

Example 2.1:

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

r1 = Rectangle(4, 5)
r2 = Rectangle(100, 100)

print(r1.width)
print(r1.height)
> 4
> 5

print(r2.width)
print(r2.height)
> 100
> 100
```
Encapsulating Data: Instance Attributes

Example 2.2:

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

r1 = Rectangle(4, 5)
r2 = Rectangle(100, 100)
```

```bash
print(Rectangle.width)
> 
```
Encapsulating Data: Instance Attributes

Example 2.2:

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

r1 = Rectangle(4, 5)
r2 = Rectangle(100, 100)
```

```python
print(Rectangle.width)
> AttributeError: type object 'Rectangle' has no attribute 'width'
```

Instance attributes are not accessible from the class.
Class VS Instance Attributes

**CLASS ATTRIBUTES**

- Defined under the `class` directive
- Values are shared across all instances
- Attributes can be accessed from the class

**INSTANCE ATTRIBUTES**

- Defined in the `__init__` method
- Values vary between instances
- Attributes do not exist for the class

When in doubt, you probably want an instance attribute.
Encapsulating Logic: Methods

Everything indented under the `class` keyword belongs to that class. This includes defining **methods** (i.e., functions)

The first argument is always be the instance* – traditionally called `self`.

*Technically there are exceptions to this rule. We will cover them later.
Encapsulating Logic: Methods

Example 3.1:

```python
# Returned value is given as 4 * 5 = 20
print(r1.area())
> 20

class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
r1 = Rectangle(4, 5)
r2 = Rectangle(100, 100)
```
Side Note: PEP8 Naming for Classes

• Class names are **CamelCase**
• Method and attribute names are **snake_case**
• Special cases:
  ◦ Variables and methods starting with an underscore are "private"
  ◦ "Dunder" methods starting and ending with double underscores extend built in functionality
• Other formatting notes:
  ◦ One space before methods (not two)
  ◦ Two spaces before classes

```python
class MyClass:
    def __init__(self):
        self._private_attr = "Don't Touch"

    def public_method(self):
        return "Hello World!"

    def _private_method(self):
        return "Not for public use"
```
Working With "Magic Methods"
### Dunder Methods

#### Lookups
- `__getattribute__`
- `__getattr__`
- `__delattr__`
- `__delitem__`
- `__delslice__`
- `__setattr__`
- `__setitem__`
- `__setslice__`
- `__missing__`
- `__getitem__`
- `__getitem__`
- `__getslice__`

#### Binary Operators
- `__add__`
- `__and__`
- `__divmod__`
- `__floordiv__`
- `__lshift__`
- `__matmul__`
- `__mod__`
- `__mul__`
- `__or__`
- `__pow__`
- `__rshift__`
- `__rtruediv__`
- `__xor__`
- `__radd__`
- `__rand__`
- `__rdivmod__`
- `__rfloordiv__`
- `__rlshift__`
- `__rmatmul__`
- `__rmul__`
- `__rmod__`
- `__rneg__`
- `__rpos__`
- `__rinvert__`

#### Unary Operators
- `__abs__`
- `__neg__`
- `__pos__`
- `__invert__`

#### Numeric Type Casting
- `__int__`
- `__bool__`
- `__nonzero__`
- `__complex__`
- `__float__`

#### Math
- `__index__`
- `__trunc__`
- `__floor__`
- `__ceil__`
- `__round__`

#### String and Repr
- `__str__`
- `__repr__`

#### Context Manager
- `__enter__`
- `__exit__`

#### Descriptor
- `__get__`
- `__set__`
- `__delete__`
- `__set_name__`

#### Async
- `__aenter__`
- `__aexit__`
- `__aiter__`
- `__anext__`
- `__await__`

#### Creation and Typing
- `__call__`
- `__class__`
- `__dir__`
- `__init__`
- `__init_subclass__`
- `__prepare__`
- `__new__`
- `__subclasses__`

#### Instance/Subclass Check
- `__instancecheck__`
- `__subclasscheck__`

#### Modules
- `__import__`

#### Others
- `__bytes__`
- `__fspath__`
- `__getnewargs__`
- `__reduce__`
- `__reduce_ex__`
- `__sizeof__`
- `__length_hint__`
- `__format__`
- `__cmp__`
Casting With __str__ (and __repr__)  

Example 4:
```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def __str__(self):
        return f"Square: {self.width} x {self.height}"

class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def __str__(self):
        return f"Square: {self.width} x {self.height}"
```

__str__ functions similarly to __repr__:
- __str__ should be human readable
- __repr__ provides technical information
- __repr__ is the fallback for __str__
Equality With __eq__

Example 5:

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def __eq__(self, other):
        same_width = self.width == other.width
        same_height = self.height == other.width
        return same_width and same_height

r1 = Rectangle(4, 5)
r2 = Rectangle(100, 100)
print(r1 == r2)  # False
r3 = Rectangle(100, 100)
print(r2 == r3)  # True
print(r2 is r3)  # False
```

Each equality operator is implemented separately (__eq__, __ne__, __gt__, __lt__, __ge__, __le__) and, or, *, //, @, etc.)
Indexing With `__getitem__`

Example 6:

```python
r1 = Rectangle(4, 5)
print(r1[0])
print(r1[1])
> 4
> 5

r1[0] = 100
print(r1[0])
> TypeError: 'type' object does not support item assignment
```

`__getitem__` and `__setitem__` are also responsible for dictionary style indexing.

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def __getitem__(self, index):
        return (self.width, self.height)[index]
```

```python
r1[0] = 100
print(r1[0])
> TypeError: 'type' object does not support item assignment
```
Callable Objects With __call__

Example 7:

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def __call__(self, value):
        print('You called this object with', value)

r1 = Rectangle(4, 5)
r1(16)
> You called with object with 16
```

Everything in Python is an object

- Functions are callable objects
- Functions implement the __call__ method
class Rectangle:

def __init__(self, width, height):
    self.width = width
    self.height = height

def __str__(self):
    return f"Square: {self.width} x {self.height}"

def __eq__(self, other):
    same_width = self.width == other.width
    same_height = self.width == other.width
    return same_width and same_height

def __getitem__(self, index):
    return (self.width, self.height)[index]

def area(self):
    return self.width * self.height
Useful Python Design Patterns
Context Managers

Context managers are used to automate setup and teardown tasks for cleaner development

```python
from pathlib import Path

with Path('my_file.txt').open() as file:
    print(file.closed)  # False

print(file.closed)  # True
```
Creating a Context Manager

class TemporaryFile:
    def __init__(self, path):
        self._path = Path(path)
        self._file = None
    def __enter__(self):
        self._path.touch()  # Make sure the file exists and open the file.
        self._file = self._path.open()
    def __exit__(self, *args):
        self._file.close()  # Close and delete the open file.
        self._path.unlink()  # Add methods as necessary to make the returned object useful
    def write(self, data):
        ...
Creating a Context Manager

Example 8:

```python
from pathlib import Path

file_path = Path('my_file.txt')
with TemporaryFile(file_path) as temp_file:
    temp_file.write('some_text')
```

```python
class TemporaryFile:
    def __init__(self, path):
        self._path = Path(path)
        self._file = None

    def __enter__(self):
        self._path.touch()
        self._file = self._path.open()
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self._file.close()
        self._path.unlink()

    def write(self, data):
        ...
```
Context Managers – Example Use Cases

• I/O Operations
• Remote Server Connections
• Database Sessions
• User Authentication
• Data caching (with cleanup)
• Resource locking
• Hardware interactions
Example 9.1

def perfect_squares(num):
    i = 0
    data = []
    while i < num:
        data.append(i * i)
        i += 1
    return data

for value in perfect_squares(10):
    ...

Q: How many iterations are performed for num=100?
A: 200 iterations
Generators with Functions

Generators are built using the yield keyword

```python
def perfect_squares(num):
    i = 0
    data = []
    while i < num:
        data.append(i * i)
        i += 1
    return data

for value in perfect_squares(10):
    ...
```

Generators are memory efficient. They are also useful for reducing runtime complexity.

Example 9.2

```python
def perfect_squares(num):
    i = 0
    while i < num:
        yield i * i
        i += 1

for value in perfect_squares(10):
    ...
```
Generators with Classes

Iteration relies on the `__iter__` and `__init__` methods behind the scenes

```
def perfect_squares(num):
    i = 0
    while i < num:
        yield i * i
        i += 1

    return data

for value in perfect_squares(10):
    ...
```

Python provides easier ways to write generators. Use dunders to expose more advanced behavior.

```
class PerfectSquares:
    def __init__(self, num):
        self._num = num
        self._i = None

    def __iter__(self):
        self._i = -1
        return self

    def __next__(self):
        self._i += 1
        if self._i < self._num:
            return self._i * self._i
        raise StopIteration()
```

Example 9.3
Generators with Comprehension

In practice, using comprehension is much simpler

```python
my_list = [i * i for i in range(10)]
> [1, 4, 9, 16, 25]

my_set = {i * i for i in range(10)}
> {1, 4, 9, 16, 25}

my_dict = {i: i * i for i in range(10)}
> {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

my_generator = (i * i for i in range(10))
> <generator object <genexpr> at 0x7f297815f5e0>
```
Decorators are callable objects that wrap other callable objects

The outer function accepts the unwrapped callable and returns a wrapped version

The inner function defines the new logic wrapped around the original callable

```
def outer(func):
    def inner(*args, **kwargs):
        print('I have been decorated!')
        func(*args, **kwargs)
    return inner
```

**Important:** Pay attention to the signature of the inner function. That will be the new signature of the wrapped function.
Decorators

Example 10.1:

```python
def my_function(x):
    print('I was given the number', x)

my_function(5)
> I was given the number 5

wrapped = outer(my_function)
wrapped(5)
> I have been decorated!
> I was given the number 5
```

```python
def outer(func):
    def inner(*args, **kwargs):
        print('I have been decorated!')
        func(*args, **kwargs)
    return inner
```

The @ syntax for Decorators

Example 10.2:

```python
@outer
def my_function(x):
    print('I was given the number', x)

my_function(5)
> I have been decorated!
> I was given the number 5
```

```python
def outer(func):
    def inner(*args, **kwargs):
        print('I have been decorated!')
        func(*args, **kwargs)
    return inner

def outer(func):
    def inner(*args, **kwargs):
        print('I have been decorated!')
        func(*args, **kwargs)
    return inner
```

```python
def outer(func):
    def inner(*args, **kwargs):
        print('I have been decorated!')
        func(*args, **kwargs)
    return inner
```
Decorators Example: @cache

```python
from functools import lru_cache

@lru_cache
def return_number(x):
    print('The function was called for', x)
    return x

print(return_number(5))
> The function was called for 5
> 5

print(return_number(5))
> 5
```
Building @cache from scratch

Q: How would you build @cache from scratch?

def cache(func):
    ...

Building @cache from scratch

Q: How would you build @cache from scratch for a function that takes a single argument?

```python
def cache(func):
    cached_data = dict()

    def wrapped(*args, **kwargs):
        key = str(args) + str(kwargs)
        if key not in cached:
            cached[key] = func(*args, **kwargs)

        return cached[key]

    return wrapped
```
Three layers can be used to define decorators that take arguments.

```python
def cache(max_size):
    def wrapper(func):
        cached_data = dict()

        def wrapped(*args, **kwargs):
            result = func(*args, **kwargs)
            if getsizeof(cached) + getsizeof(result) < max_size:
                ...

            return cached[arg]

        return wrapped

@cache(max_size=1000)
def return_number(x):
    ...
```
And yes... we can write it as a class

class Cache:

    def __init__(self, max_size):
        self._max_size = max_size

    def __call__(func):

        def wrapped(arg):
            result = func(arg)
            if getsizeof(cached) + getsizeof(result) < self._max_size:
                ...

            return cached[arg]

        return wrapped

@cache(max_size=1000)
def return_number(x):
    ...

Special Decorators for Classes
The getter/setter pattern is commonly used to modify object state

car = SelfDrivingVehicle()

# Get the current value
current_speed = car.get_speed()

# Change to a new value
car.set_speed(mph=15)
## Built In Decorators - `@property`

The property decorator turns methods into attributes

```python
car = SelfDrivingVehicle()

# Get the current value
current_speed = car.speed

# Change to a new value
car.speed = 15
```

Example 11.2

```python
class SelfDrivingVehicle:
    @property
    def speed(self):
        """Get the current speed"""

    @speed.setter
    def speed(self, mph):
        """Set the current speed"""
```
Built In Decorators - @property

If you don't provide a setter, an error is raised

car = SelfDrivingVehicle()

# Get the current value
current_speed = car.speed

# Change to a new value
car.speed = 15
>AttributeError: property 'speed' of 'SelfDrivingVehicle' object has no setter

Example 11.3

class SelfDrivingVehicle:

    @property
def speed(self):
        """Get the current speed"""
Class methods can access class attributes but not instance attributes

```python
class SelfDrivingVehicle:
    _top_speed = 80

@classmethod
def get_top_speed(cls):
    return cls._top_speed
```
Built In Decorators - @staticmethod

Static methods have no information concerning the parent class

class SelfDrivingVehicle:

    @staticmethod
    def print_version():
        print("Version 3.0.4")
Inheritance
What is Inheritance

• Inheritance allows classes to reuse logic from other classes
• Subclasses (child classes) inherit logic from parent classes
Single Inheritance

The **parent class** defines the basic level of behavior.

The **child class** adds/overwrites functionality as necessary.

The **super()** call provides access to the parent class.

class Rectangle:

```python
def __init__(self, length, width):
    self.length = length
    self.width = width

def area(self):
    return self.length * self.width
```

class Square(Rectangle):

```python
def __init__(self, length):
    super().__init__(length, length)
```
Single Inheritance

```python
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)

my_square = Square(5)
print(my_square.area())
> 25
```
Abstract Base Classes

Abstract classes are any class that implements abstract methods.

Child classes override abstract or they are also abstract.

```python
import abc

class Shape(metaclass=abc.ABCMeta):
    @abc.abstractmethod
def area(self):
        """Return the area"""

    @abc.abstractmethod
def perimeter(self):
        """Return the perimeter"""

class Rectangle:
    def area(self):
        print("This is overloaded")
```
import abc

class Shape(metaclass=abc.ABCMeta):
    @abc.abstractmethod
def area(self):
        """Return the area""
    @abc.abstractmethod
def perimeter(self):
        """Return the perimeter""

class Rectangle:
    def area(self):
        print("This is overloaded")

my_shape = Shape()
> Can't instantiate abstract class Shape with abstract methods area, perimeter

my_rectangle = Rectangle()
my_rectangle.area()
> This is overloaded
Multiple Inheritance

- You can inherit from multiple classes at once
- Python will search for methods/attributes in order of inheritance
- Inheritance is mostly a depth first search. Things get complicated when classes share parents.
Break
Introduction to SOLID
SOLID Design Principles

Object Oriented software should adhere to the normal principles:

- Keep It Simple (KISS)
- Principle of Least Surprise
- You Aren’t Going To Need It (YAGNI)
- Don’t Repeat Yourself (DRY)

It should also adhere to SOLID principles.

- S - Single-Responsibility Principle
- O - Open-Closed Principle
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle
Single Responsibility Principle (SRP)

• Every module, class, or function should be responsible for a single functionality, and it should encapsulate that part.

• In simpler terms:
  ◦ SRP applies at all levels of code (functions, classes, modules, packages)
  ◦ Each "unit of code" should be responsible for a single task
  ◦ Each unit should be properly encapsulated

• SRP does not argue for giant-monolithic structures. It’s the opposite!

"A class should have only one reason to change"
- Robert C. Martin
Question: Should the `authenticate` step be in its own class? Why?
Open/Closed

• Objects should be open for extension but closed for modification
  ◦ A class should be extendable without modifying the class itself

• Open/Closed benefits from:
  ◦ Clean inheritance structures (assuming SRP)
  ◦ Polymorphism in dependency classes
  ◦ Low coupling between classes
class Square:
    """Stores geometric properties for a square"""

def __init__(self, length):
    self.length = length

class Circle:
    """Stores geometric properties for a circle"""

def __init__(self, radius):
    self.radius = radius
class Square:
    """Stores geometric properties for a square"""

    def __init__(self, length):
        self.length = length

class Circle:
    """Stores geometric properties for a circle"""

    def __init__(self, radius):
        self.radius = radius

class Calculator:

    def total_area(self, shape_arr):
        """Return the total area for a collection of shapes"""

        total_area = 0
        for shape in shape_arr:
            if isinstance(shape, Square):
                total_area += shape.length ** 2
            elif isinstance(shape, Circle):
                total_area += pi * shape.radius ** 2

        return total_area
class Square:
    """Stores geometric properties for a square"""
    
    def __init__(self, length):
        self.length = length

    def area(self):
        return self.length ** 2

class Circle:
    
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return pi * self.radius ** 2

class AreaCalculator:

    def total_area(self, shape_arr):
        """Return the total area for a collection of shapes"""

        return sum(shape.area() for shape in shape_arr)

Notice how this solution also follows the SRP.
Liskov Substitution

Parent classes should be replaceable with their child classes

Note:
We don't actually expect random code substitutions. This is more of a "guiding principle" for designing good inheritance structures.

In practicality:
- Avoid child classes that have little in common with the parent class
- Aim for high cohesion
You have been tasked with writing two classes - one representing a `Square` and one representing a `Rectangle`.

Define these classes in a way that:
1. One class inherits from another
2. Each class has a method for the `area` of the shape
3. The classes obey Liskov Substitution
You have been tasked with writing two classes - one representing a `Square` and one representing a `Rectangle`.

Define these classes in a way that:
1. One class inherits from another
2. Each class has a method for the `area` of the shape
3. The classes obey Liskov Substitution
Liskov Substitution Example

You have been tasked with writing two classes - one representing a `Square` and one representing a `Rectangle`.

Define these classes in a way that:
1. One class inherits from another
2. Each class has a method for the `area` of the shape
3. The classes obey Liskov Substitution
You have been tasked with writing two classes - one representing a `Square` and one representing a `Rectangle`.

Define these classes in a way that:

1. One class inherits from another
2. Each class has a method for the `area` of the shape
3. The classes obey Liskov Substitution

```python
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width
```
Liskov Substitution Example

You have been tasked with writing two classes - one representing a `Square` and one representing a `Rectangle`.

Define these classes in a way that:
1. One class inherits from another
2. Each class has a method for the `area` of the shape
3. The classes obey Liskov Substitution

```python
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width
    def area(self):
        return self.length * self.width

class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)
```
Interface Segregation

• An interface is a set of abstractions:
  ◦ `Square.area()`
  ◦ `Square.perimeter()`
  ◦ `Square.width()`

• Clients should not be required to use interfaces they don’t need
  ◦ Most applicable to large projects
  ◦ Avoid giant, monolithic interfaces
  ◦ Rely on smaller, client specific interfaces
Interface Segregation Example
Interface Segregation Example

Client Group 1

Client Group 2

Interface

ML Model
Interface Segregation Example
Interface Segregation Example

Interfaces can still be subclasses of a shared (SOLID) parent class.
Dependency Inversion Principle

• High-level constructs should not rely on low level implementations
  ◦ Both should depend on abstractions (e.g., interfaces).

• Abstractions should not depend on details.
  ◦ Details (implementations) should depend on abstractions.

• In simple terms: Rely on abstractions
class AreaCalculator:
    """Return the total area for a collection of shapes"""
    def total_area(self, shape_arr):
        total_area = 0
        for shape in shape_arr:
            if isinstance(shape, Square):
                total_area += shape.length ** 2
            elif isinstance(shape, Circle):
                total_area += pi * shape.radius ** 2
        return total_area

class Square:
    """Stores geometric properties for a square"""
    def __init__(self, length):
        self.length = length

class Circle:
    """Stores geometric properties for a circle"""
    def __init__(self, radius):
        self.radius = radius
class Square:

    def __init__(self, length):
        self.length = length

    def area(self):
        return self.length ** 2

class Circle:

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return pi * self.radius ** 2

class AreaCalculator:

    def total_area(self, shape_arr):
        """Return the total area for a collection of shapes""
        return sum(shape.area() for shape in shape_arr)

Notice how this solution also follows the SRP and Open/Closed.