1. SOLID Principles (OOP)

Solid Object-Oriented Programming (OOP) principles are a set of guidelines that help developers design and implement maintainable, scalable, and robust software systems. These principles promote the creation of flexible and loosely coupled code, making it easier to modify, extend, and test the application. The SOLID acronym stands for five fundamental principles:

1.1 Single Responsibility Principle (SRP) [Separate Classes]

> *A class should have only one reason to change*

It should have a single responsibility. This principle helps maintain clear separation of concerns and makes classes easier to understand and maintain.

```
# Bad example: A class with multiple responsibilities
class Employee:
    def calculate_salary(self):
        # ...

def calculate_attendance(self):
        # ...

    def save_to_database(self):
        # ...
```

```
# Good example: Separate classes for each responsibility
class Employee:
    def calculate_salary(self):
        # ...

def calculate_attendance(self):
        # ...

class EmployeeDatabase:
    def save_to_database(self, employee):
        # ...
```

1.2 Open/Closed Principle (OCP) [Inheritance]

> *Software entities (classes, modules, functions) should be open for extension but closed for modification.*

This principle encourages designing classes in a way that allows adding new functionality without changing the existing code.

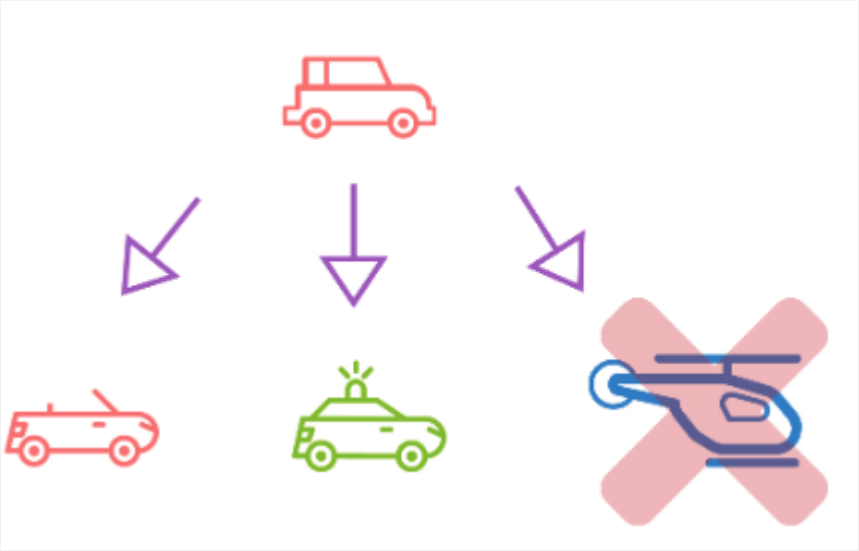
```
# Bad example: A class that requires modification to add new shapes
class AreaCalculator:
    def calculate_area(self, shape):
        if isinstance(shape, Rectangle):
            # calculate area for Rectangle
        elif isinstance(shape, Circle):
            # calculate area for Circle
```

```
# Good example: Using polymorphism to add new shapes without modifying the class
class Shape:
    def calculate_area(self):
        pass

class Rectangle(Shape):
    def calculate_area(self):
        # ...

class Circle(Shape):
    def calculate_area(self):
        # ...

# Now, we can easily add new shapes without modifying AreaCalculator
```

1.3 Liskov Substitution Principle (LSP): [Avoid jpt Inheritance]

> *Subtypes should be substitutable for their base types.*

In simpler terms, derived classes should be able to replace their base classes without affecting the correctness of the program.



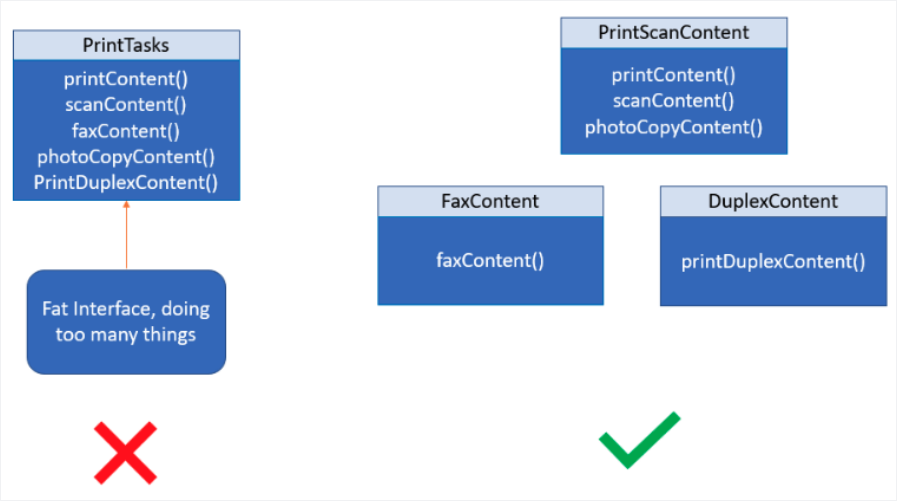
```
# Bad example: Derived class violates the contract of the base class
# Penguins are not birds, and can't fly ofcourse
class Bird:
    def fly(self):
        pass

class Penguin(Bird):
    def fly(self):
        raise NotImplementedError("Penguins cannot fly.")
```

```
# Good example: Derived class honors the contract of the base class
class Bird:
    def fly(self):
        pass

class Sparrow(Bird):
    def fly(self):
        # Sparrows can fly
```

1.4 Interface Segregation Principle (ISP):

> *Clients should not be forced to depend on interfaces they do not use.*

In other words, it is better to have multiple smaller interfaces specific to the needs of clients rather than a single large interface.



- If I have a class called *HPLaserJetPrinter* implementing this interface, I can implement all the method from this interface as *HPPrinter* provides all these facilities. So no problem seen here. Suppose if , I have another class called *CannonPrinter*, implementing this *PrintTasks* interface. This Cannon printer cannot perform fax operation. Still it is forced to implement the *faxContent* method. This is the major problem we face with fat interfaces.

```
# Bad example: A single large interface (AllInOnePrinter)
# In this case, AllInOnePrinter MUST implement 'Fax', which is not what it does
class Printer:
    def print(self):
        pass

class Fax:
    def fax(self):
        pass

class Scanner:
    def scan(self):
        pass

class AllInOnePrinter(Printer, Fax, Scanner):
    pass
```

```
# Good example: Separate interfaces for printing and fax
class Printer:
    def print(self):
        pass

class Fax:
```

```
    def fax(self):
        pass

class Scanner:
    def scan(self):
        pass

class AllInOnePrinter(Printer, Fax):
    pass

class FaxScanner(Fax, Scanner):
    pass
```

1.5 Dependency Inversion Principle (DIP) [Don't initialize a class inside another class]

> *High-level modules should not depend on low-level modules; both should depend on abstractions.*

Abstractions should not depend on details; details should depend on abstractions. This principle promotes loose coupling and flexibility.

```
# Bad example: High-level class directly depends on low-level class
class BackendService:
    def retrieve_data(self):
        # retrieve data from the backend

class FrontendService:
    def __init__(self):
        self.backend_service = BackendService()
```

```
# Good example: High-level class depends on abstractions
class BackendService:
    def retrieve_data(self):
        # retrieve data from the backend

class FrontendService:
    def __init__(self, backend_service):
        self.backend_service = backend_service
```

2. Python Exceptions

2.1 Exception Blocks

- `try`: The code within this block is executed. If an exception occurs during execution, the control is transferred to the corresponding `except` block.
- `except`: This block catches and handles the specific exception(s) raised in the `try` block. The exception is assigned to a variable (in this case, `e`) for further examination or logging.
- `else`: This block is executed only if there is no exception in the `try` block. It allows you to specify the code that should run when no exception occurs.
- `finally`: This optional block always runs, regardless of whether an exception occurred or not. It is typically used for cleanup operations, such as closing files or releasing resources.

Example of exceptions handling with all exception blocks:

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError as e:
        print("Error: Cannot divide by zero!")
    except TypeError as e:
        print(f"Error: {e} - Please provide valid numeric values.")
    else:
        print(f"The result is: {result}")
    finally:
        print("Division operation completed.")
```

```
# Example usage:
divide(10, 2)       # Output: The result is: 5.0 / Division operation completed.
divide(10, 0)       # Output: Error: Cannot divide by zero! / Division operation
completed.
divide(10, 'hello') # Output: Error: unsupported operand type(s) for /: 'int' and 'str'
- Please provide valid numeric values. / Division operation completed.
```

2.2 Commonly used Exceptions in Python

Exception	Description
`BaseException`	The base class for all built-in exceptions in Python.
`Exception`	The most common base class for non-system-exiting exceptions.
`TypeError`	Raised when an operation or function is applied to an inappropriate type.
`ValueError`	Raised when a function receives an argument of the correct type but inappropriate value.
`NameError`	Raised when a local or global name is not found or not defined.

`ZeroDivisionError`	Raised when division or modulo operation is performed with zero as the divisor.
`IndexError`	Raised when a sequence subscript is out of range.
`KeyError`	Raised when a dictionary key is not found.
`FileNotFoundError`	Raised when a file or directory is requested but cannot be found.
`IOError`	Raised when an input/output operation fails.
`AttributeError`	Raised when an attribute reference or assignment fails.
`NotImplementedError`	Raised when an abstract method or function is not implemented in a subclass.
`ImportError`	Raised when a module or package is not found or cannot be imported.
`StopIteration`	Raised when an iterator has reached its end.
`KeyboardInterrupt`	Raised when the user interrupts program execution, typically by pressing Ctrl+C.
`MemoryError`	Raised when an operation runs out of memory.
`OverflowError`	Raised when a numerical calculation exceeds the limits of its data type.
`RuntimeError`	A generic exception raised when a runtime error occurs.
`ArithmeticError`	The base class for arithmetic exceptions, including `ZeroDivisionError` and `OverflowError`.
`AssertionError`	Raised when an `assert` statement fails.
`EOFError`	Raised when the input stream has reached the end-of-file.
`SystemExit`	Raised when the `sys.exit()` function is called to exit the program.

3. Python Comprehensions

Python comprehensions are concise and expressive ways to create data structures like lists, sets, and dictionaries in a single line of code.

3.1 List Comprehensions

```
# Example
numbers = [1, 2, 3, 4, 5]
squares = [num ** 2 for num in numbers if num % 2 == 0]
# Output: [4, 16]
```

3.2 Set Comprehensions

```
numbers = [1, 2, 3, 4, 5]
even_squares = {num ** 2 for num in numbers if num % 2 == 0}
# Output: {16, 4}
```

3.3 Dictionary Comprehensions

```
numbers = [1, 2, 3, 4, 5]
squared_dict = {num: num ** 2 for num in numbers}
# Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

3.4 Generator Comprehensions

```
multiples_gen = (i for i in range(30) if i % 3 == 0)
print(multiples_gen)
# Output: <generator object <genexpr> at 0x7fdaa8e407d8>
for x in multiples_gen:
  print(x)
  # Outputs numbers
```

4. Python Collections

- `defaultdict`
- `OrderedDict`
- `Counter`
- `deque`
- `namedtuple`

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")

print(perry)
# Output: Animal(name='perry', age=31, type='cat')

print(perry.name)
# Output: 'perry'
```

- `enum.Enum`

```
from collections import namedtuple
from enum import Enum

class Species(Enum):
    cat = 1        #####
    dog = 2        #####
    horse = 3
    aardvark = 4
    butterfly = 5
    owl = 6
    platypus = 7
    dragon = 8
    unicorn = 9
    # The list goes on and on...

    # But we don't really care about age, so we can use an alias.
    kitten = 1      ######
    puppy = 2        #####

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="Perry", age=31, type=Species.cat)
drogon = Animal(name="Drogon", age=4, type=Species.dragon)
tom = Animal(name="Tom", age=75, type=Species.cat)
charlie = Animal(name="Charlie", age=2, type=Species.kitten)

# And now, some tests.
>>> charlie.type == tom.type
True
>>> charlie.type
<Species.cat: 1>
```

5. Decorators

Python decorators enables us to modify or extend the behavior of functions or methods. Decorators are essentially functions that wrap another function or method, allowing you to add functionality before, after, or around the wrapped function's execution.

Decorators are commonly used for tasks, such as:

1. **Logging:** Adding logging statements before and after function execution.
2. **Authorization:** Checking user permissions before allowing access to a function.
3. **Timing:** Measuring the time taken to execute a function.

4. **Caching:** Storing the results of expensive function calls to avoid re-computation.
5. **Input Validation:** Checking the validity of function arguments.

Python also provides built-in decorators, such as `@staticmethod`, `@classmethod`, and `@property`, which offer specific behavior for methods in classes.

Example:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

6.Ternary Operators

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = [num ** 2 if num % 2 == 0 else num for num in numbers]
print(squared_numbers)  # Output: [1, 4, 3, 16, 5]
```

7. Map, Filter and Reduce

7.1 Map

The `map` function applies a specified function to all elements of an iterable and returns a new iterable with the results.

```
numbers = [1, 2, 3, 4, 5]
squares = map(lambda x: x ** 2, numbers)
print(list(squares))  # Output: [1, 4, 9, 16, 25]
```

7.2 Filter

The `filter` function filters elements from an iterable based on a specified function that returns `True` or `False`.

```
numbers = [1, 2, 3, 4, 5]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # Output: [2, 4]
```

7.3 Reduce

The `reduce` function performs a cumulative computation on elements of an iterable from left to right, reducing the iterable to a single result.

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
sum_of_numbers = reduce(lambda x, y: x + y, numbers)
print(sum_of_numbers)  # Output: 15 (1 + 2 + 3 + 4 + 5)
```

8. Generators

In Python, a generator is a special type of iterable, similar to lists or tuples, but with a crucial difference. While lists store all their elements in memory at once, generators generate elements on-the-fly as you iterate over them. This feature makes generators memory-efficient and useful for handling large datasets or infinite sequences.

Generators are best for calculating large sets of results (particularly calculations involving loops themselves) where you don’t want to allocate the memory for all results at the same time.

Generators are defined using functions with the `yield` keyword instead of `return`. When a function with a `yield` statement is called, it returns a generator object. Each time the generator's `next()` method is called, the function runs up to the `yield` statement, yields the value, and pauses its execution. The next

time `next()` is called, the function resumes from where it was paused and continues until it encounters another `yield` statement or reaches the end of the function.

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Using the generator
fib_gen = fibonacci()
for _ in range(10):
    print(next(fib_gen), end=' ')  # Output: 0 1 1 2 3 5 8 13 21 34
```

In this example, the `fibonacci` generator keeps yielding the next Fibonacci number indefinitely.

9. *args and **kwargs

*args and **kwargs allow functions to be more flexible and accommodate different numbers of input arguments.

9.1 *args (Positional Arguments):

The `*args` syntax allows a function to accept a variable number of positional arguments. When you use `*args` as a parameter in a function definition, it allows the function to receive any number of positional arguments passed to it. The `*args` parameter collects these arguments into a tuple.
Example:

```
def sum_numbers(*args):
    total = sum(args)
    return total
```

```
print(sum_numbers(1, 2, 3))  # Output: 6
print(sum_numbers(1, 2, 3, 4, 5))  # Output: 15
```

In this example, the `sum_numbers` function accepts any number of positional arguments and calculates their sum.

9.2 **kwargs (Keyword Arguments):

The `**kwargs` syntax allows a function to accept a variable number of *keyword arguments.* When you use `**kwargs` as a parameter in a function definition, it allows the function to receive any number of keyword arguments passed to it. The `**kwargs` parameter collects these arguments into a dictionary. Example:

```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_kwargs(name="John", age=30, city="New York")
# Output:
# name: John
# age: 30
# city: New York
```

In this example, the `print_kwargs` function accepts any number of keyword arguments and prints their key-value pairs.

9.3 Both *args and **kwargs

```
def print_args_and_kwargs(*args, **kwargs):
    print("Positional arguments:")
    for arg in args:
        print(arg)

    print("Keyword arguments:")
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_args_and_kwargs(1, 2, 3, name="Alice", age=25)
# Output:
# Positional arguments:
# 1
# 2
# 3
# Keyword arguments:
# name: Alice
# age: 25
```