PD
PENTEST DIARIES
HTTPS://WWW.SAKSHAMDIXIT.COM

Public, Private & Protected Access Specifiers in Python:

In Python, access control is a bit different compared to languages like Java or C++. Python doesn't have explicit keywords like "public", "private", or "protected" to specify access levels. However, it does have a convention and some mechanisms for controlling access to attributes and methods.

Public: In Python, all attributes and methods are public by default. This means they can be accessed from outside the class.

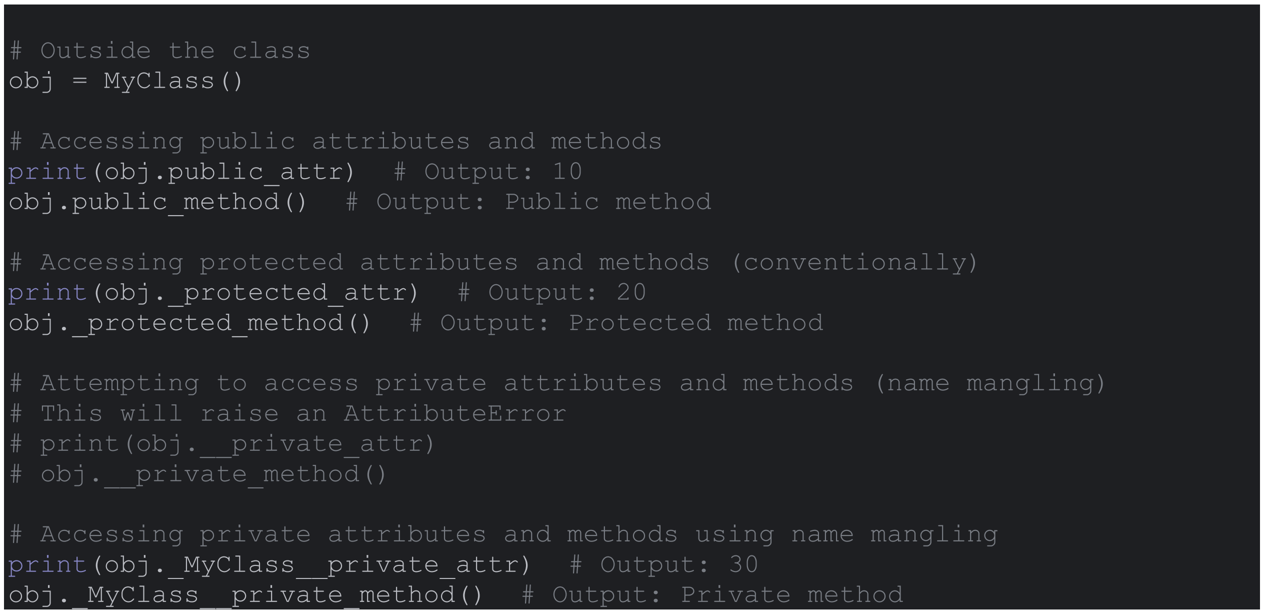
Private: Python uses name mangling to create a kind of "private" attribute or method. If an attribute or method name starts with two underscores (__), Python will internally rename it to `_ClassName__attribute` or `_ClassName__method`. This makes it difficult to access from outside the class, but it's not truly private since it can still be accessed with the mangled name. However, it serves as a convention to indicate that the attribute or method is intended for internal use by the class.

Protected: Python doesn't have a true "protected" access specifier like some other languages. Conventionally, attributes or methods prefixed with a single underscore (_) are considered protected. This is more of a hint to other developers that the attribute or method is not intended to be used outside the class, but there's nothing in Python to enforce this.

Here's a simple example to illustrate these concepts:

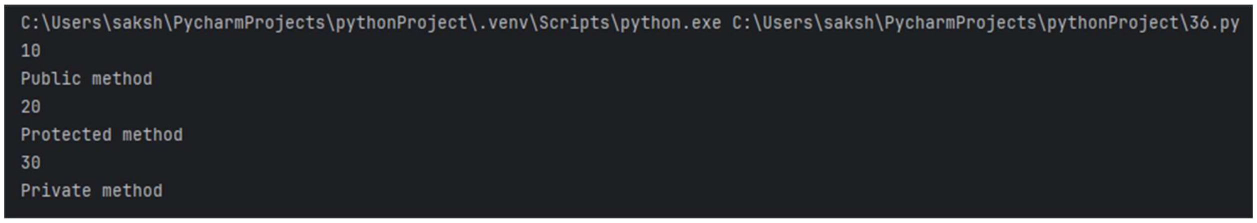
CODE:

```
class MyClass:
    def __init__(self):
        self.public_attr = 10
        self._protected_attr = 20
        self.__private_attr = 30

    def public_method(self):
        print("Public method")

    def _protected_method(self):
        print("Protected method")

    def __private_method(self):
        print("Private method")
```



```
# Outside the class
obj = MyClass()

# Accessing public attributes and methods
print(obj.public_attr)  # Output: 10
obj.public_method()  # Output: Public method

# Accessing protected attributes and methods (conventionally)
print(obj._protected_attr)  # Output: 20
obj._protected_method()  # Output: Protected method

# Attempting to access private attributes and methods (name mangling)
# This will raise an AttributeError
# print(obj.__private_attr)
# obj.__private_method()

# Accessing private attributes and methods using name mangling
print(obj._MyClass__private_attr)  # Output: 30
obj._MyClass__private_method()  # Output: Private method
```

Output:



```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\36.py
10
Public method
20
Protected method
30
Private method
```

Remember, in Python, the emphasis is on readability and simplicity, so it's considered more Pythonic to trust developers not to access attributes or methods that are conventionally marked as private or protected.

Polymorphism In Python:

Polymorphism in Python refers to the ability of different objects to respond to the same message or method invocation in different ways. It allows objects of different types to be treated as objects of a common superclass.

Method Overriding:

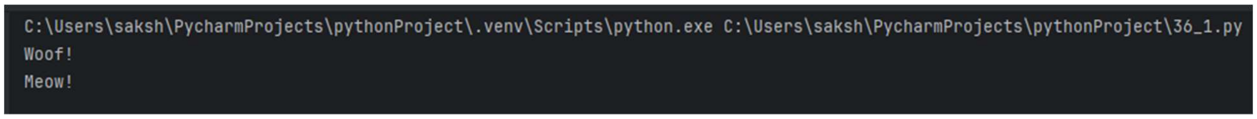
Method overriding in Python refers to the ability to redefine a method in a subclass that already exists in its superclass. When a method is overridden in a subclass, the subclass provides a new implementation of the method that overrides the behavior of the method in the superclass.

Here's an example of method overriding in Python:

CODE:

```python
class Animal:
    def make_sound(self):
        print("Some generic sound")

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")

# Create instances of Dog and Cat
dog = Dog()
cat = Cat()

# Call make_sound method on instances
dog.make_sound()   # Output: Woof!
cat.make_sound()   # Output: Meow!
```

Output:



```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\36_1.py
Woof!
Meow!
```

In this example, the `Animal` class has a method called `make_sound()`, which prints a generic sound. Both the `Dog` and `Cat` classes inherit from the `Animal` class. However, they provide their own implementations of the `make_sound()` method, which override the behavior of the method in the `Animal` class.

When you call the `make_sound()` method on instances of `Dog` and `Cat`, Python uses the overridden methods defined in the respective subclasses (`Dog` and `Cat`), rather than the method defined in the superclass (`Animal`).

Duck Typing:

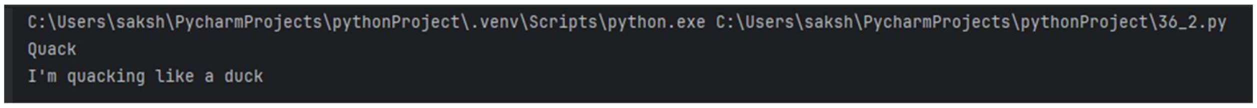
Duck typing in Python is a concept where the type or class of an object is less important than the methods it defines. It's a way of thinking about interfaces in a dynamic language like Python. The term comes from the saying, "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

In Python, you often don't need to explicitly specify the type of an object. Instead, you can rely on whether an object supports the necessary methods or attributes. For example:

CODE:

```
class Duck:
    def quack(self):
        print("Quack!")

class Person:
    def quack(self):
        print("I'm quacking like a duck!")

def make_it_quack(entity):
    entity.quack()

duck = Duck()
person = Person()

make_it_quack(duck)    # Output: Quack!
make_it_quack(person) # Output: I'm quacking like a duck!
```

Output:



```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\36_2.py
Quack
I'm quacking like a duck
```

In the example above, both the `Duck` class and the `Person` class have a `quack()` method. So, even though they are different classes, as long as they "quack" like a duck, they can be passed to the `make_it_quack()` function.

This flexibility is one of the reasons why Python is considered a highly dynamic and expressive language. It allows you to focus more on what objects can do rather than what they are.

Operator Overloading:

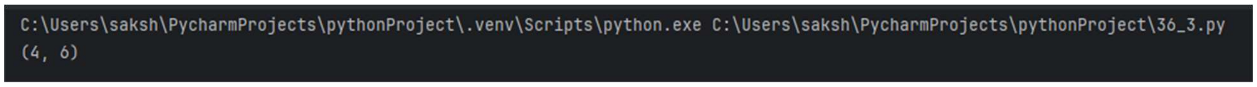
Operator overloading in Python refers to the ability to define custom behavior for built-in operators like `+`, `-`, `*`, `/`, `==`, `<`, `>`, and others. This allows objects of user-defined classes to work with these operators in a way that makes sense for the class.

Here's a simple example of operator overloading in Python:

CODE:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"


# Create two Point objects
point1 = Point(1, 2)
point2 = Point(3, 4)

# Use the '+' operator with Point objects
result = point1 + point2

print(result)  # Output: (4, 6)
```

Output:



```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\36_3.py
(4, 6)
```

In this example, the `Point` class defines the `__add__` method to overload the `+` operator. When you use the `+` operator with two `Point` objects (`point1 + point2`), Python calls the `__add__` method of the left-hand operand (`point1`) with the right-hand operand (`point2`) as an argument.

You can similarly overload other operators such as `__sub__` for `-`, `__mul__` for `*`, `__eq__` for `==`, `__lt__` for `<`, and so on.

Operator overloading can make your code more expressive and intuitive, allowing your custom objects to behave more like built-in types. However, it's important to use operator overloading judiciously and follow Pythonic conventions to avoid confusion and maintainability issues.

Method Overloading:

In Python, method overloading typically refers to the ability to define multiple methods with the same name but with different numbers or types of parameters. However, unlike some other programming languages like Java or C++, Python does not support method overloading in the traditional sense.

In languages like Java, method overloading allows you to define multiple methods with the same name but different parameter lists, and the appropriate method is chosen by the compiler based on the arguments passed at the call site.

In Python, since you can define default parameter values and variable-length argument lists, you can achieve similar functionality without needing to explicitly overload methods. Here's an example:

CODE:

```python
class Animal:
    def make_sound(self):
        print("Some generic sound")

class Dog(Animal):
    def make_sound(self):
        print("Woof!")


class Calculator:
    def add(self, x, y):
        return x + y

    def add(self, x, y, z):
        return x + y + z


# Create an instance of Calculator
calc = Calculator()

# Call the overloaded add method
print(calc.add(2, 3))  # This will raise an error
print(calc.add(2, 3, 4))  # Output: 9
```

Output:

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\36_4.py
9
```

In the above example, the second definition of the `add()` method overrides the first one. However, if you try to call `calc.add(2, 3)`, it will raise an error because Python does not support method overloading based on the number of arguments.

While you can't directly achieve method overloading based on the number or type of parameters in Python, you can use default parameter values or variable-length argument lists to create functions or methods that behave differently depending on the arguments passed. This flexibility is one of Python's strengths, as it allows you to write concise and expressive code without relying on language features like method overloading.