Python é uma linguagem de programação multi-paradigma. Um dos paradigmas suportados é o paradigma de orientação a objetos, e dessa maneira, possui diversos mecanismos de reuso de código. O exemplo abaixo mostra como uma classe é construída em Python, com métodos, atributos e seu construtor:

```
class MinhaClasse:
      def __init__(self, meuAtributo):
            print("Este é o construtor")
            self.meuAtributo = meuAtributo

      def metodo(self):
            print("Este é meu método")
            print(self.meuAtributo + " é meu atributo")
```

Python permite heranças de maneira simples (herdar de apenas uma classe) ou múltipla (várias classes), e também polimorfismos, como mostra os exemplos abaixo.

```
class Pai(object):
      def __init__(self):
      print('Construindo a classe Pai')

      class Filha(Pai):
            def __init__(self):
            super().__init__()
```

Aqui vemos como a herança é realizada, utilizando o método `super()` para invocar o construtor da classe pai. No caso de herança múltipla, a prioridade é definida da esquerda para a direita na declaração de classes pai.

```
class English(object):
      def greet(self):
      print(“Hi!”)

class Portuguese(object):
      def greet(self):
      print(“Oi!”)

class Bilingual(English, Portuguese):
      pass

if __name__ == ‘__main__’:
Bilingual().greet()
```

A composição de classes é uma ferramenta que podemos utilizar quando possuimos objetos que utilizam outros objetos em sua implementação, como se tivéssemos um grande objeto que engloba outros. O exemplo abaixo representa como seria uma possível implementação de uma calculadora que faz composição de quatro classes: Pilha, teclas, tela e unidade lógica aritmética (ULA).

```
class Calculadora():

      def __init__(self):
            self.pilha = Pilha()
            self.entrada = Teclas()
            self.ula = ULA()
            self.tela = Tela()

      def novaOperacao(self, valor1,valor2):
            self.entrada.valorEntrada(valor1,valor2)
            self.pilha.uso()

      def soma(self):
            soma = self.ula.soma(self.entrada.getValor())
            self.tela.mostraTexto(soma)
            self.bateria.uso()
```

No que se trata de métodos estáticos (que não precisam que seu objeto seja instanciado para ser invocado), existe uma função pronta em Python que pode converter um método comum para um método estático, a função `staticmethod`. Outra solução é usar o decorador `@staticmethod` antes de alguma função para torná-la estática. Segue um exemplo que demonstra ambos os casos:

```
      class Mathematics:

            def addNumbers(x, y):
                  return x + y

            @staticmethod
            def subNumbers(x, y):
                  return x - y


      Mathematics.addNumbers = staticmethod(Mathematics.addNumbers)

      print('The sum is:', Mathematics.addNumbers(5, 10))
      print('The sub is:', Mathematics.subNumbers(5, 10))
```

Por último, métodos abstratos podem ser utilizados com a biblioteca ABS, a *Abstract Base Classes,* conforme representado no exemplo final:

```
from abc import ABCMeta, abstractmethod

class Abstract(metaclass=ABCMeta):
     def use_concrete_implementation(self):
          print(self._concrete_method())

     @abstractmethod
     def _concrete_method(self):
          pass

class Concrete(Abstract):

     def _concrete_method(self):
          return 2 * 3
```