Methods:

Array of parameters

```
def sum_numbers(*args):
    total = 0
    for num in args:
        total += num
    return total

# Using the sum_numbers method with different numbers of arguments
print(sum_numbers(1, 2, 3))
# Output: 6
print(sum_numbers(1, 2, 3, 4, 5))
# Output: 15
print(sum_numbers(10, 20, 30, 40, 50))
# Output: 150
```

Class methods

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display_info(self):
        return f"Name: {self.name}, Age: {self.age}"

    @classmethod
    def from_string(cls, string):
        name, age = string.split(',')
        return cls(name.strip(), int(age.strip()))

# Using the class method to create Person objects
person1 = Person.from_string("Alice, 30")
person2 = Person.from_string("Bob, 25")

# Displaying information of created Person objects
print(person1.display_info())
# Output: Name: Alice, Age: 30
print(person2.display_info())
# Output: Name: Bob, Age: 25
```

In/Out parameters

```
def double_numbers(numbers):
    for i in range(len(numbers)):
        numbers[i] *= 2
    return numbers

# Original list of numbers
original_numbers = [1, 2, 3, 4, 5]

# Calling the method with the original list
modified_numbers = double_numbers(original_numbers)

# Displaying the modified list
print("Modified Numbers:", modified_numbers)
# Output: Modified Numbers: [2, 4, 6, 8, 10]

# Original list remains unchanged
print("Original Numbers:", original_numbers)
# Output: Original Numbers: [1, 2, 3, 4, 5]
```

Multiple return values

```
import math

def get_circle_info(radius):
    area = math.pi * radius**2
    circumference = 2 * math.pi * radius
    return area, circumference

# Calling the method and unpacking the returned tuple
circle_area, circle_circumference = get_circle_info(5)

# Displaying the results
print("Circle Area:", circle_area)
# Output: Circle Area: 78.53981633974483
print("Circle Circumference:", circle_circumference)
# Output: Circle Circumference: 31.41592653589793
```

Optional parameter values

```
def greet(name, message="Hello"):
    return f"{message}, {name}!"

# Calling the method with and without providing a custom message
print(greet("Alice"))
# Output: Hello, Alice!
print(greet("Bob", "Hi there"))
# Output: Hi there, Bob!
```

Variable parameters

```
def sum_numbers(*args):
    total = 0
    for num in args:
        total += num
    return total

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

# Using the sum_numbers method with different numbers of positional
arguments
print("Sum:", sum_numbers(1, 2, 3))
# Output: Sum: 6
print("Sum:", sum_numbers(1, 2, 3, 4, 5))
# Output: Sum: 15
print("Sum:", sum_numbers(10, 20, 30, 40, 50))
# Output: Sum: 150

# Using the print_info method with different numbers of keyword
arguments
print_info(name="Alice", age=30)
# Output: name: Alice, age: 30
print_info(name="Bob", age=25, city="New York")
# Output: name: Bob, age: 25, city: New York
```

With return value

```
def add_numbers(a, b):
    return a + b

# Calling the method and storing the returned value
result = add_numbers(3, 5)

# Displaying the returned value
print("Result:", result)  # Output: Result: 8
```

Without any parameters

```
from datetime import datetime

def get_current_year():
    return datetime.now().year

# Calling the method
current_year = get_current_year()

# Displaying the current year
print("Current Year:", current_year)
```

Without any return value

```
def print_message(message):
    print("Message:", message)

# Calling the method
print_message("Hello, World!")
```

Nested class

```
class Outer:
    def __init__(self, name):
        self.name = name
        self.inner = self.Inner()

    def display_outer(self):
        print("Outer Name:", self.name)

    class Inner:
        def display_inner(self):
            print("Inner Class")

# Creating an instance of the outer class
outer_obj = Outer("Outer Object")

# Accessing methods of the outer class
outer_obj.display_outer()
# Output: Outer Name: Outer Object

# Accessing methods of the inner class
inner_obj = outer_obj.inner
inner_obj.display_inner()
# Output: Inner Class
```

Properties:

Computed properties

```python
import math

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

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

# Creating an instance of Circle
circle = Circle(5)

# Accessing the computed property
print("Radius:", circle.radius)
# Output: Radius: 5
print("Area:", circle.area)
# Output: Area: 78.53981633974483
```

Lazy properties

```
import math

class LazyProperty:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, owner):
        if instance is None:
            return self
        value = self.func(instance)
        setattr(instance, self.func.__name__, value)
        return value

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

    @LazyProperty
    def area(self):
        print("Calculating area...")
        return math.pi * self.radius ** 2

# Creating an instance of Circle
circle = Circle(5)

# Accessing the lazy property
print("Radius:", circle.radius)
# Output: Radius: 5
print("Area:", circle.area)
# Output: Calculating area... \n Area: 78.53981633974483
print("Area:", circle.area)
# Output: Area: 78.53981633974483 (no re-calculation)
```

Read-Only properties: Computed properties

```
import math


class Circle:
    def __init__(self):
        self.radius = 0

    @property
    def area(self):
        return math.pi * pow(self.radius, 2)


circle = Circle()
circle.radius = 2
# circle.area is 12.566370614359172

print(circle.area)
```

Read-Only properties: Stored properties

```
class FilmList:
    def __init__(self):
        self.__count = 10

    @property
    def count(self):
        return self.__count


filmList = FilmList()
count = filmList.count

print(count) # count is  10
```

Stored properties

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Creating an instance of Person
person = Person("Alice", 30)

# Accessing stored properties
print("Name:", person.name)  # Output: Name: Alice
print("Age:", person.age)    # Output: Age: 30

# Modifying stored properties
person.name = "Bob"
person.age = 25

# Displaying modified properties
print("Modified Name:", person.name)
# Output: Modified Name: Bob
print("Modified Age:", person.age)
# Output: Modified Age: 25
```

Type properties

```
class Circle:
    pi = 3.14159

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

    def calculate_area(self):
        return Circle.pi * self.radius ** 2

# Creating instances of Circle
circle1 = Circle(5)
circle2 = Circle(10)

# Accessing the type property
print("Value of pi:", Circle.pi)  # Output: Value of pi: 3.14159

# Calculating areas using type property
print("Area of circle 1:", circle1.calculate_area())
# Output: Area of circle 1: 78.53975
print("Area of circle 2:", circle2.calculate_area())
# Output: Area of circle 2: 314.159
```

Subscripts (indexer methods):

With generic parameter

```
class MyList:
    def __init__(self):
        self.data = {}

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

    def __setitem__(self, index, value):
        self.data[index] = value

# Creating an instance of MyList
my_list = MyList()

# Using integer indices
my_list[0] = 'a'
my_list[1] = 'b'
print("Element at index 0:", my_list[0])
# Output: Element at index 0: a
print("Element at index 1:", my_list[1])
# Output: Element at index 1: b

# Using string keys
my_list['first'] = 10
my_list['second'] = 20
print("Element with key 'first':", my_list['first'])
# Output: Element with key 'first': 10
print("Element with key 'second':", my_list['second'])
# Output: Element with key 'second': 20
```

With multiple parameter

```
class Matrix:
    def __init__(self, rows, columns):
        self.rows = rows
        self.columns = columns
        self.data = [[0] * columns for _ in range(rows)]

    def __getitem__(self, indices):
        row, column = indices
        return self.data[row][column]

    def __setitem__(self, indices, value):
        row, column = indices
        self.data[row][column] = value

# Creating an instance of Matrix
matrix = Matrix(3, 3)

# Setting values using multiple indices
matrix[0, 0] = 1
matrix[1, 1] = 2
matrix[2, 2] = 3

# Getting values using multiple indices
print("Value at position (0, 0):", matrix[0, 0])
# Output: Value at position (0, 0): 1
print("Value at position (1, 1):", matrix[1, 1])
# Output: Value at position (1, 1): 2
print("Value at position (2, 2):", matrix[2, 2])
# Output: Value at position (2, 2): 3
```