Classes and Objects

```
#| edit: false
#| echo: false
# include the required modules

import numpy as np
import matplotlib.pyplot as plt

plt.rcParams.update({'font.size': 12,
                     'lines.linewidth': 1,
                     'lines.markersize': 10,
                     'axes.labelsize': 11,
                     'xtick.labelsize' : 10,
                     'ytick.labelsize' : 10,
                     'xtick.top' : True,
                     'xtick.direction' : 'in',
                     'ytick.right' : True,
                     'ytick.direction' : 'in',})
```

Object oriented programming

A very useful programming concept is object oriented programming. In all the programs we wrote till now, we have designed our program around functions i.e. blocks of statements which manipulate data. This is called the procedure-oriented way of programming.

There is another way of organizing your program which is to combine data and functionality and wrap it inside something called an object. This is called the object oriented programming paradigm, which will be useful especially for larger programs.

Classes and Objects

Object-oriented programming is built upon two fundamental concepts: classes and objects.

- A class is a blueprint or template that defines a new type of object. Think of it as a mold that creates objects with specific characteristics and behaviors.
- Objects are specific instances of a class. They have two main components:
  - **Properties** (also called attributes or fields): Variables that store data within the object
  - **Methods**: Functions that define what the object can do
- Properties come in two varieties:
  - **Instance variables**: Unique to each object instance (each object has its own copy)
  - **Class variables**: Shared among all instances of the class (one copy for the entire class)

For example, if you had a `Car` class: - Instance variables might include `color` and `mileage` (unique to each car) - Class variables might include `number_of_wheels` (same for all cars) - Methods might include `start_engine()` or `brake()`

Creating Classes

To define a class in Python, we use this basic syntax:

```
class ClassName:
    # Class content goes here
```

The definition starts with the `class` keyword, followed by the class name, and a colon. The class content is indented and contains all properties and methods of the class.

Here's a minimal example:

```
#| autorun: false
class Colloid:
    pass  # 'pass' creates an empty class with no properties or methods
```

To create an object (an instance) of this class:

```
#| autorun: false
particle = Colloid()
particle  # Prints the object's location in memory
```

Class Methods

Methods are functions that belong to a class. They define the behavior of the class and can operate on the class's properties.

> Understanding `self` in Python Classes
>
> Every method in a Python class automatically receives a special first parameter, conventionally named `self`. This parameter refers to the specific instance of the class that calls the method.
> Key points about `self`: - It's automatically passed by Python when you call a method - It gives the method access to the instance's properties - By convention, we name it `self` (though technically you could use any valid name) - You don't include it when calling the method
> Example:
>
> ```
> class Colloid:
>     def type(self):  # self is automatically provided
>         print('I am a plastic colloid')
>
> # Usage:
> particle = Colloid()
> particle.type()  # Notice: no argument needed for self
> ```
>
> In this example, even though `type()` appears to take no arguments when called, it actually receives the `particle` object as `self`.

```
#| autorun: false
class Colloid:
    def type(self):
        print('I am a plastic colloid')

p = Colloid()
p.type()

b=Colloid()
b.type()
```

The Constructor Method: `__init__`

The `__init__` method (called the constructor) is a special method that initializes a new object when it's created. It allows you to: - Set up initial values for the object's properties - Perform

any setup the object needs when it's created

The name has double underscores (dunders) at both ends: `__init__`

Here's an example:

```
#| autorun: false
class Colloid:
    def __init__(self, R):
        self.R = R  # Stores the radius as an instance variable

    def get_size(self):
        return self.R  # Method to retrieve the radius
```

Using the class:

```
#| autorun: false
# Create two colloids with different radii
particle1 = Colloid(5)  # radius = 5
particle2 = Colloid(2)  # radius = 2

# Get the size of particle1
print(f'Colloid radius is {particle1.get_size()} μm')
```

> **Note**
>
> Python also provides a `__del__` method (destructor) that's called when an object is deleted. We'll see this in action later.

The String Representation: `__str__` Method

The `__str__` method defines how an object should be represented as a string. Python automatically calls this method when: - You use `print(object)` - You convert the object to a string using `str(object)`

Here's an example:

```
#| autorun: false
class Colloid:
    def __init__(self, R):
        self.R = R  # Initialize radius
```

```
    def get_size(self):
        return self.R

    def __str__(self):
        # Define how the object should be displayed as text
        return f'I am a plastic colloid of radius {self.R:.1f}'
```

> Tip
>
> The `.1f` format specification means the radius will be displayed with one decimal place. You can customize this string representation to show whatever information about your object is most relevant.

Let's see it in action:

```
#| autorun: false
# Create a colloid with radius 15
particle = Colloid(15)

# Print the object - this automatically calls __str__
print(particle)
```

Understanding Class and Instance Variables

In Python classes, we can have two types of variables that store data:

Class Variables (Shared Data)

- Shared among all instances of a class
- Defined inside the class but outside any method
- All objects share the same copy of these variables
- Changes affect all instances
- Useful for tracking data common to all instances

Instance Variables (Individual Data)

- Unique to each instance/object
- Usually defined in `__init__`
- Each object has its own copy

- Changes only affect that specific instance
- Useful for object-specific properties

Here's a practical example:

```
#| autorun: false
class Colloid:
    # Class variable: tracks total number of particles
    total_particles = 0

    def __init__(self, R):
        # Instance variable: each particle has its own radius
        self.R = R
        # Increment counter when new particle is created
        Colloid.total_particles += 1

    def __del__(self):
        # Decrement counter when particle is deleted
        Colloid.total_particles -= 1
```

Let's see how it works:

```
#| autorun: false
# Create two particles
p1 = Colloid(3)   # Particle with radius 3
p2 = Colloid(12)  # Particle with radius 12

# Each particle has its own radius (instance variable)
print(f"Particle radii: p1 = {p1.R}, p2 = {p2.R}")

# Both share the same total_particles count (class variable)
print(f"Total particles: {Colloid.total_particles}")

# Delete one particle
del p2
print(f"After deletion, total particles: {Colloid.total_particles}")
```

> **Tip**
>
> Common uses for class variables:
>
> - Counters (like tracking total instances)

- Constants shared by all instances
- Configuration values for all objects