Classes

Classes

A **class** provides a way to **bundle** data and functionality together.

It creates a new `type` of object

and each `instance` of that `object` can have its own `attributes` and `methods`.

Classes

In python a class is defined using the `class` keyword.

```python
class ClassName:
    <statements>
    <statements>
    <statements>
    <statements>
    <statements>
    <statements>
```

Similar to a function, the body of a class is *indented*

Note that like a function, when a class is defined, no code in the body is executed. And a new namespace is created for the class.

Class Objects

```python
class Myclass:
    i = 12345

    def f(self):
        return 'hello world'
```

In this example, `Myclass` is a class object with an attribute `i` and a method `f`.

This object has two operations, **attribute references** and **instantiation**.

Attribute References

```python
class Myclass:
    i = 12345

    def f(self):
        return 'hello world'
```

In attribute references, we use dot notation `obj.name` to access the attribute or method.

```python
Myclass.i
Myclass.f
```

returns `12345` and a `function object` respectively.

Instantiation

```python
class Myclass:
    i = 12345

    def f(self):
        return 'hello world'
```

In instantiation, we create a new instance of the class by calling the class object.

```python
x = Myclass()
```

This returns a new **instance** of `Myclass`.

An instance is an object that contains the data defined and functions in the class.

At it’s core, think of the class as a blueprint for creating objects.

Initialization

During initialization, we usually want to set up the instance with **some initial values**.

We can do this by defining a special method `__init__()` in the class.

```python
class Myclass:
    current = 0

    def __init__(self, start, increment=1):
        self.start = start
        self.increment = increment
        current = start


    def inc(self):
        self.current += self.increment
        return self.current
```

Initialization

So if we instantiate the class with

```
x = Myclass(5) # Myclass + ()
```

and call

```
x.inc()
```

`6` gets returned.

And if we make another instance (in the same program)

```
y = Myclass(10, 2)
```

and call

```
y.inc()
```

`12` gets returned.

Note that these instances (`x` and `y`) have their own separate data.

The self parameter

In the method definitions, the first parameter is always `self`.

This parameter refers to the **instance object itself**.

And it's automatically passed by Python when we call a method on an instance.

```python
class Dog:
    kind = "canine"

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

d = Dog("Fido")
e = Dog("Buddy")
```

In this example, `d` and `e` are two different instances of the `Dog` class.

When we call

```python
d.name
e.name
```

```python
d.kind
e.kind
```

Note that both `d` and `e` share the same **class attribute** `kind`, but have different **instance attributes** `name`.

The self parameter

If we wanted to define a method that makes use of the instance's data, we can do so using the `self` parameter.

```python
class Dog:
    kind = "canine"

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

    def get_kind(self):
        return self.kind

    def bark(self):
        return f"{self.name} says woof!"
```

Note that you **cannot** access instance attributes or methods without using `self`. Even class attributes should be accessed using `self` to ensure proper resolution.

Codechum Time

Create a class `Counter` that has the following methods:

- `__init__(self, start=0)` : initializes the counter with a starting value (default is 0)
- `increment(self, amount=1)` : increases the counter by the specified amount (default is 1)
- `decrement(self, amount=1)` : decreases the counter by the specified amount (default is 1)
- `get_value(self)` : returns the current value of the counter
- `reset(self)` : resets the counter to the initial starting value