C9 Classes

Class

1. is a self-defined class which allow you to write all the relevant methods in the same class

```
class Duck:
    sound = 'Quack quack.'
    movement = 'Walks like a duck.'

    def quack(self):
        print(self.sound)

    def move(self):
        print(self.movement)
```

Constructors

1. constructor by using init which allow user to define private variable (only can be called within it class) but Python do not have it, so that it is just an idea to remind the user to avoid any changes on it

```
class Animal:
    #the first arg is self which means it is an object method that references to itself
    def __init__(self, type, name, sound):
        #return the object variable in _name format (private variable but Python don't have it)
        self._type = type
        self._name = name
        self._sound = sound

    #another way to assign the paramter without following the orders
    def __init__(self, **kwargs):
        #defined the default value for each variable
        self._type = kwargs["type"] if "type" in kwargs else "kitten"
        self._name = kwargs["name"] if "name" in kwargs else "fluffy"
        self._sound = kwargs["sound"] if "sound" in kwargs else "rawr"

    def type(self):
        return self._type #accessor

    def name(self):
        return self._name

    def sound(self):
        return self._sound

def print_animal(o):
    if not isinstance(o, Animal):
        raise TypeError('print_animal(): requires an Animal')
    print('The {} is named "{}" and says "{}".'.format(o.type(), o.name(), o.sound()))


def main():
    a0 = Animal('kitten', 'fluffy', 'rwar')
    a1 = Animal('duck', 'donald', 'quack')
    #by using kwagrs
    a2 = Animal(type = "dog", name = "bert", sound = "wou")
```

```
    print_animal(a0)
    print_animal(a1)
    print_animal(a2)
    print_animal(Animal('velociraptor', 'veronica', 'hello'))

if __name__ == '__main__': main()
```

Methods

1. also known as function in order to conduct any process
2. can be defined by using def keyword

```
def __str__(self):
        return f'The {self.type()} is named "{self.name()}" and says "{self.sound()}".'
```

Inheritance

1. allow to extends classes by driving the variables from parent class

```
class Animal:
    def __init__(self, **kwargs):
        if 'type' in kwargs: self._type = kwargs['type']
        if 'name' in kwargs: self._name = kwargs['name']
        if 'sound' in kwargs: self._sound = kwargs['sound']

    def type(self, t = None):
        if t: self._type = t
        #exception
        try: return self._type
        except AttributeError: return None

    def name(self, n = None):
        if n: self._name = n
        try: return self._name
        except AttributeError: return None

    def sound(self, s = None):
        if s: self._sound = s
        try: return self._sound
        except AttributeError: return None

#duck and kitten class are inherit from the parent class which is Animal
#super is the keyword that both classes are using the variables from the Animal class
class Duck(Animal):
    def __init__(self, **kwargs):
        self._type = 'duck'
        if 'type' in kwargs: del kwargs['type']
        super().__init__(**kwargs)

class Kitten(Animal):
    def __init__(self, **kwargs):
        self._type = 'kitten'
        if 'type' in kwargs: del kwargs['type']
        super().__init__(**kwargs)

    def kill(self, s):
        print(f"{self.name()} will now kill all {s}!")

def print_animal(o):
```

```
    if not isinstance(o, Animal):
        raise TypeError('print_animal(): requires an Animal')
    print(f'The {o.type()} is named "{o.name()}" and says "{o.sound()}".')

def main():
    a0 = Kitten(name = 'fluffy', sound = 'rwar')
    a1 = Duck(name = 'donald', sound = 'quack')
    print_animal(a0)
    print_animal(a1)
    a0.kill("humans")

if __name__ == '__main__': main()
```

Iterators

1. an object that contains a countable number of values
2. is an object which implements the iterator protocol, which consist of the methods **iter**() and **next**()

```
class inclusive_range:
    def __init__(self, *args):
        numargs = len(args)
        self._start = 0
        self._step = 1

        if numargs < 1:
            raise TypeError(f'expected at least 1 argument, got {numargs}')
        elif numargs == 1:
            self._stop = args[0]
        elif numargs == 2:
            (self._start, self._stop) = args
        elif numargs == 3:
            (self._start, self._stop, self._step) = args
        else: raise TypeError(f'expected at most 3 arguments, got {numargs}')

        self._next = self._start

    def __iter__(self):
        return self

    def __next__(self):
        if self._next > self._stop:
            raise StopIteration
        else:
            _r = self._next
            self._next += self._step
            return _r

def main():
    for n in inclusive_range(25):
        print(n, end=' ')
    print()

if __name__ == '__main__': main()
```