Data Structures and Algorithms with Python

Workshop 3

In this workshop you'll start implementing your own data structures in Python. You'll be doing this using an object-oriented programming approach, creating new **classes** to represent new types of data in your program. Object-oriented programming essentially provides a new way of structuring your data — for example pocket monsters — and the procedures associated with them.

The workshop has two main parts:

1. Monsters
2. Extra: Queues

A 'skeleton' code for these exercises is provided in the file available on the Hub. Open this file in Spyder and you're ready to go.

1 Monsters

We started building a data structure for pocket monsters in a game where you collect monsters and use them to fight other monsters. A pocket monster has a number of attributes, like a name, a type, hit points and combat points. We could represent a pocket monster by creating variables like this:

```
monsterName = 'Pikachu'
combatPoints = 30
hitPoints = 13
type = 'lightning'
```

However, if we want to have many monsters, we would need too many variables. We could then try storing the monsters in a dictionary:

```
monsters = {'Pikachu':[20,53,'electric'],'Squirtle':[82,90,'water'],/
            'Mew':[1940,599,'ridiculous']}
```

But this also quickly becomes unwieldy. What if we want to add different kinds of attacks for each of the monsters? These attacks would have their own characteristics (name, damage, etc.). We would likely end up with dictionaries nested within lists within dictionaries! We're better off defining monsters (and attacks) as **classes**. A class defines a new type of object for Python. Other types of objects we have seen include lists, floats, and dictionaries — now you're adding your own.

```
class Monster():
    def __init__(self,name,monsterType,combatPoints,hitPoints):
        self.name = name
        self.monsterType = monsterType
        self.combatPoints = combatPoints
        self.hitPoints = hitPoints
        self.health = hitPoints # current health
```

The `__init__` function is a special function which is called when you create a new instance of the monster object. It will take the inputs you give to it and use them to create a new monster with these attributes. The keyword `self` is also special and used within the object to conveniently refer to itself. You will have to use it similarly in all functions you define within the class. Creating a monster works like this.

```
pika = Monster('Pikachu','electric',100,80)
```

The class also includes a host of other functions. Many of these simply return the different attributes of the monster. It is good programming practice to use functions like this instead of directly referring to a monster's attributes.

```
pika = Monster('Pikachu','electric',100,80)
print(pika.getMonsterType()) # good practice
```

```
print(pika.monsterType) # not as good practice
```

In more complex projects, using such functions would help make sure that these values are not accidentally changed. Indeed, some programming languages have this kind of walls around object attributes by default. We are, however, not going to worry about these practices too much in this module.

Your first task is to complete the methods in the monster object. In class, we built a function called `hurt`, which reduces a monster's health. Implement the method `heal`, which increases its health. However, a monster's health cannot exceed its hit points. Your method should also print out the monster's health status similarly to the `hurt` function.

Next, you'll add attacks to the monsters. Following the design of the monster class, design a class `Attack` which defines attacks. An attack has the following attributes:

- `name`, a string
- `attackType`, a string
- `damage`, an integer.

It should have the following functions:

- `__init__` is used when a new `Attack` object instance is created. It stores the input attributes.
- `getAttackType` returns the attack type.
- `getDamage` returns the attack damage.
- `__str__` returns the name (this will be called if you use print(attack))

Now you have designed a data structure to capture the attributes of attacks. But the monsters don't know how to use them yet... next, update your implementation of the `Monster` class so that one of a monster's attributes is a list of attacks. Furthermore, implement a function `useAttack(self,attack,otherMonster)` within the `Monster` class that uses an attack on another monster. This function should `hurt` the other monster with the current monster's `damage`.

Use your method to have monsters combat each other. Of course, the actual damage from an attack might depend on various factors such as the relative combat points of the two monsters. A

strong monster attacking a weak opponent may be more effective. To capture this, you might rewrite the function by multiplying the attack damage by the ratio of the attacker's and the defender's combat points. What else would you need to make the game more interesting?

There are many other things we could add to our monster objects. For example, it is possible that there would be many instances of the same monster species, eg two Squirtles. In such a case, we might want to consider creating a separate class out of each species. The species would be special cases of the `Monster` class. This is referred to as *inheritance*: a `Squirtle` object would inherit all the characteristics of a `Monster`, but could have other attributes and methods beyond those.

```
1 class Squirtle(Monster):
2     #code...
```

We won't go further into the topic of inheritance here, but it is a useful topic to know: see the textbook's Chapter 8.

An interesting exercise to think through at home would be making a list of different objects you would have to model to design an entire pocket monster game.

2 Extra: Queues

We now move from pocket monsters back to efficient algorithm design. In the lecture, we looked briefly into the mechanics of implementing linear data structures, specifically lists. There are various ways of doing this, the simplest of which are an array and a linked list. We saw that the efficiency of list operations depends on its implementation. The difference is that an array keeps track of the absolute positions of elements while a linked list only tracks their relative positions. Adding an element to a linked list is therefore constant time, but in an array the complexity depends on the position of the element. If the element is appended to the end of the list, this is still constant time, but adding to the beginning is $O(n)$ as all other elements need to be moved forward by one position. However, accessing a linked list by index is $O(n)$ while any position in an array can be accessed in $O(1)$.

In this exercise, you'll implement another linear data structure: a queue. Queues work much as you would expect from your experience in the college cafe: you add items to one end of the queue, and remove them from the other end. Queues are not only everywhere in the world (especially in

London), but their design is crucial to operations in many service industries. Furthermore, we will see in the following workshops that a queue is a useful abstraction in algorithm design. Its advantage is that when efficiently implemented, its operations (adding and removing items) are both constant $O(1)$ running time.

The Python file provides two implementations of a queue. The first, `listQueue`, uses the Python `list` to build a queue. It has four methods.

- `__init__` is used when a new `listQueue` object is created. It creates an empty list of items in the queue.
- `isEmpty` simply checks whether the queue is empty.
- `enqueue` adds an item to the queue (in the front of the list)
- `dequeue` removes an item from the queue (from the end of the list)

Note how the methods again refer to the `listQueue` object itself using the keyword `self`. Check that the methods work as expected.

```
class ListQueue:
    """ Queue using list
        Supports inserting and deleting items
    """
    def __init__(self):
        self.items=[]

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()
```

The problem with the above implementation is that adding items to the beginning of the list is computationally expensive. Your task is to implement a more efficient queue class without using a Python list. You'll do this by creating a queue of *nodes*. A `Node` implementation is given below. Essentially, a node is a simple data structure that stores unspecified data as `stuff`, as well as information on what other node (if any) it is connected to.

```
class Node:
    """ Node class: contains unspecified data in stuff and link to next Node
    """
    def __init__(self, stuff=None, next=None):
        # this will run when you create a new Node
        self.stuff = stuff
        self.next  = next

    def __str__(self): # this will run when you use print() on a node
        return str(self.stuff)
```

Try creating a few nodes with string inputs.

The Python file contains a skeleton implementation of a `Queue` class. Your task is to complete it so that it performs the same operations as the `listQueue` class but creating the queue from `Node` objects by linking them together. In order to quickly add and remove elements, your class should keep track of which node is first and last in the queue, the length of the queue, as well as the `next` node for each node. Here's how it works.

- Suppose we start with an empty queue, and we add a node a. Now a is both the first and the last item in the queue, but since there are no other items, a does not have a `next` node. The length of our queue is one.
- Now let's add another node b. The node a now needs to point to b as the `next` node. Node a is still first in the queue, but the last item needs to be updated to be b. The last node does not point to another node. The length of the queue is increased to two.
- Let's remove a node. This should return the `stuff` inside the first node, a. Now b becomes the first node, and the length of the queue is decreased to one.

Note you can pass `stuff` directly as input when adding to the queue, as the `Node` is created within the queue. This kind of abstraction is often useful: the user does not have to worry about the internal operations of your data structure.

The class also contains a `__str__` method for you to complete. This is useful for printing out your queue status. After you have completed the class, try out the queue and compare the speed of the enqueueing operations of the two classes.

Python has a built-in data structure `deque` which implements a slightly more complicated data type called *double-ended queue*, giving an efficient way to use queues. Look for more information on double-ended queues on eg Wikipedia.