Informatics 2 – Introduction to Algorithms and Data Structures

Lab Sheet 3: Classes and more

In this lab we will be introducing *classes* in Python, and using them for a few examples that will be relevant to other aspects of the course. While Python isn't a fundamentally *object-oriented* language (in the way that Java or C# is, for example), classes may be defined and are useful for some purposes.

1 Defining and using classes

If you've never heard of classes or object-oriented programming, here is a quick introduction. As the name suggests, the fundamental notion in object-oriented programming is that of *object*. Objects may represent anything: a person, a company, an algorithm ... but what is important is that all objects are instances of *classes*. Classes define both a *type* to which all its instances belong (analogous to the int, String or floating point number types) **and** a fundamental structure (a *template*, you could say) for what objects of that class look like.

For example, in a company management program, we may wish to define a `Person` class, indicating their `name` and their `job` (the *attributes* of the `Person` class).
In Python, a class is defined with the `class` keyword. For instance:

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

  def getEmployeeDescription(self):
    return self.name + " is a " + self.job
```

There are quite a few things to explain here.

1. The `__init__` and `getEmployeeDescription` functions are called *methods* of the `Person` class. They can only be called on its instances, and their attributes and other methods are available within the method. Unlike Java or C#, in Python you need to explicitly indicate this by adding the `self` argument to every method of a class, that refers to the object (the Person) on which the method is called.

2. To create an object of a class, we use the name of the class as if it were a function, with the arguments of the `__init__` method, except for the `self` argument[1]:

```
>>> p = Person("Freddie","Algorithm Designer")
>>> p.name
>>> p.job
```

3. The `__init__` method is called the *constructor* of the class. It is automatically called whenever a new element of the class is created, and it is used to initialize objects. Often, we need to make sure that all instances of a class satisfy some invariant or constraint, and we cannot do this if we do not run some code every time an object of the class is created.

4. To call a method of an object (for example `getEmployeeDescription`), append `.methodName(arguments)` to the object. Note that the `self` argument does not need to be provided because it is the object itself. For example:

```
>>> p.getEmployeeDescription()
```

The attractions of classes include (but are not limited to) packaging a data structure under a single common name that can be used everywhere, and implementing a series of common operations on these data structures, allowing us to separate out their implementation from their use.

Python does not have explicit *access modifiers* like Java (e.g. `public` or `private`), but we can (almost) achieve this by prepending a method or attribute name with a double underscore (`__`) to indicate that they are private.

2 Encapsulating useful data structures

One useful application of classes is to implement *abstract data structures*. One of the simplest examples of these is the *queue*, which we'll discuss in Lecture 6. Like an ordinary queue of people, a queue is a kind of list, where new elements can be added to the tail of the queue and removed from the head. (Queues are sometimes known as *first-in-first-out lists*, because the order in which elements join the tail is the same as the order in which they reach the head.) For our purposes, a queue will be an object with methods to add (`push`) a new element, to extract (`pop`) an existing element from the head, and to check whether the queue is empty. The method names and arguments of a Queue class could look like this:

[1] If you are writing the Person class definition in a separate file (as you should be), remember to import it into the interpreter using *import filename*. Then, you would need to prepend the *Person* call with *filename.*

```
class Queue:
  # Returns True or False:
  def isEmpty(self):
    ...

  # Does not return anything:
  def push(self,x):
    ...

  # Returns the element on the head of the queue, if there is one.
  def pop(self):
    ...
```

But what is the right way to implement these methods? The punchline is, there is more than one. In Java, Queue is an *interface*, defining only the method names and signatures, and this interface has several different implementations. In Python there is not an equivalent notion of interface, but we can achieve a similar result with ABCs (Abstract Base Casses) and *inheritance*.

Let's talk about inheritance first. We say that a class A extends another class B (inherits from it) if every instance of A is *also* an instance of B. Moreover, when we specify the structure of A, we depart from already having all the structure of B and being able to use it, including both attributes and methods.

ABCs take this one step further by allowing us to define classes that are never meant to be directly instantiated, but instead only provide a common structure for several subclasses with different specific behaviours. Inheritance can take place between regular classes, but when defining an ABC, it *must* be extended.

We can define Queue as an ABC, as follows:

```
import abc

class Queue(abc.ABC):
  @abc.abstractmethod
  def isEmpty(self):
    pass

  @abc.abstractmethod
  def push(self,x):
    pass

  @abc.abstractmethod
  def pop(self):
    pass
```

The `@abc.abstractmethod` annotation indicates that the method does not have a standard implementation in the ABC: Subclasses must provide it.

Let's provide a queue implementation: Linked Lists.[2] Remember that a Linked List is formed of individual cells that are linked referentially, so that adding or removing a cell at a position we've already located is easy, but searching for elements in the list is hard. This is perfect for queues, since we only need to manipulate data at the front and back of the queue. To extend a class, we add an "argument" to the class name indicating its superclass (in this case `Queue`). Note too that this implementation uses an auxiliary class `LL_Queue_Cell` for individual cells.

```
class LL_Queue(Queue):
    def __init__(self):
        self.tail = LL_Queue_Cell()
        self.head = self.tail

    def isEmpty(self):
        return self.head.value == None

    def push(self, value):
        self.tail.value = value
        self.tail.next = LL_Queue_Cell()
        self.tail = self.tail.next

    def pop(self):
        if self.isEmpty():
            raise IndexError
        else:
            value = self.head.value
            self.head = self.head.next
            return value

class LL_Queue_Cell:
    def __init__(self):
        self.next = None
        self.value = None
```

Try it! Fetch the file `queue.py` from the course website,(or else copy the above code for `Queue` and `LL_Queue` into a new file `queue.py`), then run the following:

```
>>> import queue
>>> q = queue.LL_Queue()
>>> q.isEmpty()
>>> q.push(1)
>>> q.isEmpty()
>>> q.pop()
>>> q.isEmpty()
>>> q.push(2)
>>> q.push(3)
>>> q.pop()
>>> q.isEmpty()
```

[2]This implementation is due to Franz Miltz, an Inf2-IADS student in 2020–21.

```
>>> q.push(4)
>>> q.pop()
```

Exercise 1:
A different implementation of a queue is a *circular buffer*. A circular buffer is a sequence of memory cells (an array) of *fixed length* that we use to store the values of the queue in. In addition, it stores two indices, pointing to cells in the buffer, indicating the head and the tail of the current state of the queue. The twist is that the tail can be an earlier index than the head, and in such a case we consider the queue to *wrap around* from the end of the buffer to the beginning.
Here are some examples. Assume a buffer of size 5, and in all cases with the following data in the memory cells:

`|'a'|'b'|'c'|'d'|'e'|`

Then, if:

- The head is 0 and the tail is 3: the queue contains three elements: `'a'`, `'b'` and `'c'`, has `'a'` as top element and `'c'` as bottom element. The next `pop` will return `'a'`, and the next `push` will replace the content of the cell with `'d'`.
- The head is 2 and the tail is 2: the queue is empty.
- The head is 1 and the tail is 0: the queue contains four elements: `'b'`, `'c'`, `'d'` and `'e'`.
- The head is 4 and the tail is 4: the queue is empty.
- The head is 4 and the tail is 0: the queue contains one element: `'e'`.
- The head is 3 and the tail is 1: the queue contains three elements in the following order from top to bottom: `'d'`, `'e'` and `'a'`. The next `pop` will return `'d'`, and the next `push` will replace the content of the cell with `'b'`.

Implement a subclass of Queue that is a circular buffer. Call it `CircBuffer_Queue`. Leave the size of the buffer as a constructor parameter.

Hint: The modulus operation (`%`) can be particularly useful to implement wrap-around operations.

Exercise 2:
Do you see any important limitations in the circular buffer implementation? Does the linked list implementation have the same limitation? Does it have other disadvantages?

Can you think of how to get rid of the limitation in the circular buffer implementation?

Optional (if you have the time): Modify your circular buffer implementation so that it does not have the identified limitation.

2.1 Dynamic vs. strictly typed languages and type safety

As a final note on classes in Python, note that, compared to strictlier typed languages like Java or C#, in a dynamic language like Python, because variables do not have types (only values have types), inheritance and abstract data types are slightly less powerful / slightly less necessary: we do not need an abstract data type in order to have a queue of unknown implementation stored in a variable or passed as an argument, any regular variable is already able to do that and we are able to use anything as if it were a queue regardless of its type (it will just fail sometimes). The abstract base class approach is a way to make this more precise, but it does not carry as many functional advantages as interfaces do in Java, for example.
There is, however, one small functional advantage of ABCs: the `isinstance` function. `isinstance` takes as arguments an object and a class, and returns True if and only if the object is an instance of the class *transitively*. That is, if it is a direct instance of the class or of any subclass of it.

```
>>> ll_q = queue.LL_Queue()
>>> cb_q = queue.CircBuffer_Queue(50)
>>> isinstance(ll_q,queue.LL_Queue)
>>> isinstance(ll_q,queue.Queue)
>>> isinstance(ll_q,queue.CircBuffer_Queue)
>>> isinstance(cb_q,queue.LL_Queue)
>>> isinstance(cb_q,queue.Queue)
>>> isinstance(cb_q,queue.CircBuffer_Queue)
```

This allows us to provide some type safety on runtime to our usage of abstract data types, providing the necessary error control in the cases where types do not match in an ordered manner.

3 Files and using data structures

In this section we will briefly introduce you to using files in Python, but quickly wrap this around our own data structures. We might be using these data structures in the first coursework of the course.

Similarly to most programming languages and most operating systems, file access in Python consists mainly of two pairs of fundamental operations:

- `open` and `close`, that need to be done before and after a file is read/written. This informs the operating system of our intent to perform operations with the file, so that it will check permissions and perhaps do some additional operations like deal with concurrent access locks. You should *always* explicitly close a file that you explicitly open, and when in write mode, the file will usually not be properly written until you close it. When a file is opened, it is usually opened either in *read or in write mode*, but not both.

  In Python, the basic open and close functions are:

```
reader_or_writer = open(filename,mode,encoding)
reader_or_writer.close()
```

where `mode` is either `'r'` for read or `'w'` for write. While specifying the encoding is optional, it is useful to specify it to ensure that the files are written or read in the character encoding that we intend them to. Normally we will want to use `encoding='utf-8'`.

- `read` and `write`. You can only read from a file opened in read mode and write to a file opened in write mode. In Python, to read and write text, these are implemented via:

```
lines = reader.readlines(n)
writer.writelines(lines)
```

where `n` is the number of lines we wish to read and `lines` is an array of strings.

3.1 Buffered reading and writing

It is usual, however, to want to read or write a file one line at a time. Unfortunately, this is not very efficient when it comes to disk usage. Reading or writing n lines of text is usually almost as fast as writing just 1, for n not too big. This is due to how most hard drives' hardware works, where *sectors* comprise entire files that need to be *sought* before reading or writing, which takes as much or more time as the actual reading or writing operation.
This is not an unusual situation when it comes to algorithm and data structure design: the way in which a resource is accessed *most efficiently* is not the way in which it is accessed *most conveniently*. Fortunately, in many cases, including this one, we can obtain the best of both worlds by wrapping the resource in the adequate data structure: *buffered readers and writers*.

The idea is the following. We don't really *need* to read or write one line at a time, it just makes our code simpler if we can do so. When the lines are actually read or written is often not important, and we don't mind if they are in practice read and written in chunks, ahead of time or slightly after. We just don't want to have to do that manually every time in code dedicated to other algorithms. Buffered readers and writers are structures that read or write from files in chunks (many lines at a time), but store the results in memory, providing methods to read or write one line at a time from them.

Here is an implementation of classes `BufferedInput` and `BufferedOutput`[3].

```
class BufferedInput:
    def __init__(self,filename,memory):
        self.reader = open(filename,'r',encoding='utf-8')
        self.buffer = self.reader.readlines(memory)
        self.memory = memory
        self.pos = 0
```

[3]We might be using these in the coursework, so take a close look at them.

```
    def readline(self):
        if self.buffer == []:
            return ''
        else:
            result = self.buffer[self.pos]
            self.pos += 1
            if self.pos == len(self.buffer):
                self.buffer = self.reader.readlines(self.memory)
                self.pos = 0
            return result
    def close(self):
        self.reader.close()

class BufferedOutput:
    def __init__(self,filename,size):
        self.writer = open(filename,'w',encoding='utf-8')
        self.buffer = ['' for i in range(size)]
        self.lines = size
        self.pos = 0
    def writeline(self,str):
        self.buffer[self.pos] = str
        self.pos += 1
        if self.pos == self.lines:
            self.writer.writelines(self.buffer)
            self.pos = 0
    def flush(self): # flushes buffer and closes file
        self.writer.writelines(self.buffer[0:self.pos])
        self.writer.close()
```

Note that the size provided to either of them is the size of the *buffer*, not the size of the file or the number of lines we want to read: when the buffer is full, it is emptied (either by replacing it with new values or by flushing it to the output) and more lines are read and/or output.

Try them. At this point, you should pause to download the files `numbers1.txt`, `numbers2.txt`, `numbers3.txt`, `numbers4.txt`, `numbers5.txt` from the course website, and make sure these are in the same folder as your Python program (which we assume is called `files.py` here). Now run the following:

```
>>> import files
>>> reader = files.BufferedInput("numbers1.txt",50)
>>> writer = files.BufferedOutput("test_output.txt",100)
>>> # Read the first line, but do nothing with it.
... reader.readline()
>>> # Read the second and write it to output.
... writer.writeline(reader.readline())
>>> # Same for third.
... writer.writeline(reader.readline())
>>> reader.close()
>>> writer.flush()
```

Open the output file `test_output.txt` and verify that the output is correct.

In order to read from a file until no more lines remain, in Python we use an endless `while` loop with a check for nothing returned inside and a `break`:

```
>>> reader = files.BufferedInput("numbers1.txt",50)
>>> while True:
...   line = reader.readline()
...   if not line:
...     break
...   print(line)
```

Exercise 3:
You may have noticed that the file `numbers1.txt` contains numbers prepended by `N:`. Define a function `readnumbers(filename)` that reads a file, and for each line that starts with `N:`, reads the number that comes afterwards, storing all of them into a list that is returned as a result of the function. Any lines that do not start with `N:` must be ignored.

Notes:

- Remember to close the file (using the wrapper class function) after finishing with it.
- Use any appropriate buffer size that you think. Try with different values. It should work equally with all of them.

Exercise 4:
Look at the file `numbers2.txt`. It contains more numbers, but also other file names prepended by `I:` (for "import"). Modify your `readnumbers(filename)` function so that it *recursively* reads numbers from files:

- When it finds a `N:` line, it reads the number as before.
- When it finds a `I:` line, it reads all the numbers in the referenced file, via a recursive call.
- Any other lines must be ignored.

To check the correctness of your implementation, run it with `numbers2.txt` as parameters. Notice that this file transitively imports `numbers3.txt`, `numbers5.txt` and `numbers1.txt`, but **not** `numbers4.txt`. Verify your function worked correctly: the result should include numbers $0, 40, 47, 500$ and 3, among others, but **not** 1000 or 2000.

Hint: You may find the `extend` method of lists useful. It can be used to concatenate a list to another existing one.
Hint: You may run into errors for including the end of line character `\n` into the filename. Remember to remove that character from the filename when you make the recursive call.

Exercise 5:
Recursion is great, but is sometimes dangerous or too much. In particular, our recursive implementation of `readnumbers` keeps an unknown number of files open at the same time. While this is not necessarily a problem, it is often good practice to be moderate when using operating system resources like files, memory, keyboard, screen, etc.

Queues happen to be a great way to remove recursion *when the order is not relevant*. Use queues to read the files and their imported files, one at a time. To do this, you will need to remove the recursive call and use a `while` loop, while using a queue to store the files that are still pending to be read.

The order of the numbers in the output will be different, but the individual numbers should be the same. Verify it. Try both using a `LL_Queue` and a `CircBuffer_Queue`. The results should be the same, so long as the circular buffer is big enough (100 should be way more than enough).