Assignment 3 v. 1.1

COMP9021, Session 1, 2016

1 General matter

1.1 Aims

The purpose of the assignment is to:

- gain some acquaintance with some of the specific features of Python, hidden methods in particular;
- design and implement an interface based on the desired behaviour of an application program;
- design and implement a strategy to explore a large problem space;
- develop problem solving skills.

1.2 Submission

Your program will be stored in two files, one being named `ring.py`, the other being named `tangram.py`. It can be useful to let the second one import the first one. Upload your file using WebCMS. Assignments can be submitted more than once: the last version is marked. Your assignment is due by June 5 11:59pm.

1.3 Assessment

The assignment consists of two parts. The first part is worth 5 marks, the second part 4 marks (though the second part might require more efforts than the first part).

For each test, the automarking script will let your program run for 30 seconds.

For the second part, as all output is `True` or `False`, each test will actually consist of a number of subtests of similar difficulty, and passing one test will mean passing all associated subtests; failing at least one of the subtests of a test will result in scoring 0 to that test.

Up to one mark will reward good comments, good choice of names for identifiers and functions, readability of code, simplicity of statements, compactness of functions. This will be determined manually.

Late assignments will be penalised: the mark for a late submission will be the minimum of the awarded mark and 10 minus the number of full and partial days that have elapsed from the due date.

1.4 Reminder on plagiarism policy

You are permitted, indeed encouraged, to discuss ways to solve the assignment with other people. Such discussions must be in terms of algorithms, not code. But you must implement the solution on your own. Submissions are routinely scanned for similarities that occur when students copy and modify other people's work, or work very closely together on a single implementation. Severe penalties apply.

2 The ring class (5 marks)

The main aim is to modify the meaning of slices so that we can select parts of a sequence in such a way that wrapping around is possible. For that purpose, we define a `ring` module.

```
$ python3
...
>>> from ring import *
```

The class we are defining can operate on sequences, and on sequences only, including objects of class `Ring`, with an empty list as default.

```
>>> ring = Ring({1, 2, 3})
Traceback (most recent call last):
...
TypeError: 'set' object is not subscriptable
>>> ring = Ring(1)
...
TypeError: 'int' object is not subscriptable
>>> ring = Ring([1, 2, 3])
>>> ring
Ring([1, 2, 3])
>>> print(ring)
[1, 2, 3]
>>> ring = Ring((1, 2, 3))
>>> ring
Ring((1, 2, 3))
>>> print(ring)
(1, 2, 3)
>>> ring = Ring()
>>> ring
Ring([])
>>> ring = Ring([1, 2, 3])
>>> Ring(ring)
Ring([1, 2, 3])
>>> ring = Ring((1, 2, 3))
>>> Ring(ring)
Ring((1, 2, 3))
```

We can use indexes or slices, with the usual error messages when indexes or arguments of slices are not integers:

```
>>> ring = Ring([1, 2, 3])
>>> ring['a']
Traceback (most recent call last):
...
TypeError: ring indices must be integers or slices, not str
>>> ring['a': 1: 2]
Traceback (most recent call last):
...
TypeError: slice indices must be integers or None or have an __index__ method
>>> ring[0: : 'a']
Traceback (most recent call last):
...
TypeError: slice indices must be integers or None or have an __index__ method
```

We can use any integer as index, computing modulo the length of the sequence.

```
>>> ring = Ring([1, 2, 3])
>>> len(ring)
3
>>> ring[-5], ring[-4], ring[-3], ring[-2], ring[-1]
(2, 3, 1, 2, 3)
>>> ring[0], ring[1], ring[2], ring[3], ring[4], ring[5]
(1, 2, 3, 1, 2, 3)
```

With slices, the second argument is included, not excluded, and we wrap around when needed:

```
>>> ring = Ring(tuple(range(5)))
>>> ring[2: 2]
(2,)
>>> ring[2: 3]
(2, 3)
>>> ring[2: 4]
(2, 3, 4)
>>> ring[2: 5]
(2, 3, 4, 0)
>>> ring[2: 6]
(2, 3, 4, 0, 1)
>>> ring[2: 7]
(2,)
>>> ring[3: 2]
(3, 4, 0, 1, 2)
>>> ring[4: 2]
(4, 0, 1, 2)
>>> ring[5: 2]
(0, 1, 2)
>>> ring[6: 2]
(1, 2)
>>> ring[7: 2]
(2,)
>>> ring[8: 2]
(3, 4, 0, 1, 2)
```

The first argument of a slice defaults to 0. The second argument defaults to the first argument minus 1 when step is positive, and to the first argument plus 1 when step is negative. The third argument defaults to 1. Also, a step of 0 is acceptable, which always returns an empty sequence.

```
>>> ring[: ]
(0, 1, 2, 3, 4)
>>> ring[: :-1]
(0, 4, 3, 2, 1)
>>> ring[: 3]
(0, 1, 2, 3)
>>> ring[: 3: -1]
(0, 4, 3)
>>> ring[3: : ]
(3, 4, 0, 1, 2)
>>> ring[3: : -1]
(3, 2, 1, 0, 4)
>>> ring[2: : 0]
()
```

More examples, with a step different to 1 or -1:

```
>>> ring = Ring(list(range(5)))
>>> ring[3: 1: 2]
[3, 0]
>>> ring[2: 3: -2]
[2, 0, 3]
>>> ring[2: 1: 3]
[2, 0]
>>> ring[2: 3: -3]
[2, 4]
```

We can also change some `Ring` objects in place. Our interface is similar to the standard one when we deal with indexes:

```
>>> ring = Ring(tuple(range(3)))
>>> ring[1] = 8
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
>>> ring = Ring(list(range(5)))
>>> ring[2] = -2
>>> ring
Ring([0, 1, -2, 3, 4])
>>> ring[21] = -1
>>> ring
Ring([0, -1, -2, 3, 4])
```

But it is very different to the standard one when we deal with slices. First, we accept a step of 0, in which case we replace simple slices. Note that when we replace part of the end and the beginning of a sequence, the replacement is always placed at the beginning of the sequence.

```
>>> ring = Ring(list(range(5)))
>>> ring[2: 4: 0] = [-1]
>>> ring
Ring([0, 1, -1])
>>> ring = Ring(list(range(5)))
>>> ring[4: 4: 0] = [-1, -2]
>>> ring
Ring([0, 1, 2, 3, -1, -2])
>>> ring = Ring(list(range(5)))
>>> ring[3: 2: 0] = (-1,)
>>> ring
Ring([-1])
>>> ring = Ring(list(range(5)))
>>> ring[3: 1: 0] = (-1, -2, -3)
>>> ring
Ring([-1, -2, -3, 2])
```

For strictly positive steps, we *insert* data, between the start and the end, in a number of slots that has to be equal to the number of data to insert:

```
>>> ring = Ring(list(range(5)))
>>> ring[2: 4] = [-1]
Traceback (most recent call last):
...
ValueError: attempt to insert sequence of size 1 to extended slice of size 2
>>> ring[2: 4] = [-1, -2, -3]
Traceback (most recent call last):
...
ValueError: attempt to insert sequence of size 3 to extended slice of size 2
```

The following illustrates. Note that when we insert a datum between the last and the first element of a sequence, that datum is always placed at the beginning of the sequence.

```
>>> ring = Ring(list(range(5)))
>>> ring[0: 3] = -1, -2, -3
>>> ring
Ring([0, -1, 1, -2, 2, -3, 3, 4])
>>> ring = Ring(list(range(5)))
>>> ring[2: 1] = -1, -2, -3, -4
>>> ring
Ring([-3, 0, -4, 1, 2, -1, 3, -2, 4])
>>> ring = Ring(list(range(5)))
>>> ring[2: 0] = -1, -2, -3
>>> ring
Ring([-3, 0, 1, 2, -1, 3, -2, 4])
>>> ring = Ring(list(range(5)))
>>> ring[4: 1] = -1, -2
>>> ring
Ring([-1, 0, -2, 1, 2, 3, 4])
>>> ring = Ring(list(range(5)))
>>> ring[4: 0] = (-1,)
>>> ring
Ring([-1, 0, 1, 2, 3, 4])
>>> ring = Ring(list(range(5)))
>>> ring[3: 3] = -1, -2, -3, -4, -5
>>> ring
Ring([-2, 0, -3, 1, -4, 2, -5, 3, -1, 4])
>>> ring = Ring(list(range(10)))
>>> ring[1: 7: 2] = -1, -2, -3
>>> ring
Ring([0, 1, -1, 2, 3, -2, 4, 5, -3, 6, 7, 8, 9])
>>> ring = Ring(list(range(10)))
>>> ring[7: 2: 2] = -1, -2, -3
>>> ring
Ring([-2, 0, 1, -3, 2, 3, 4, 5, 6, 7, -1, 8, 9])
```

For strictly negative steps, we *replace* data, between the start and the end, that have to be equal in number to the number of data to replace:

```
>>> ring = Ring(list(range(5)))
>>> ring[2: 4: -1] = [-1, -2]
Traceback (most recent call last):
...
ValueError: attempt to replace sequence of size 2 to extended slice of size 3
>>> ring[2: 4: -1] = [-1, -2, -3, -4]
Traceback (most recent call last):
...
ValueError: attempt to replace sequence of size 4 to extended slice of size 3
```

The following illustrates.

```
>>> ring = Ring(list(range(5)))
>>> ring[0: 3: -1] = -1, -2, -3, -4
>>> ring
Ring([-1, -2, -3, -4, 4])
>>> ring = Ring(list(range(5)))
>>> ring[2: 1: -1] = -1, -2, -3, -4, -5
>>> ring
Ring([-4, -5, -1, -2, -3])
>>> ring = Ring(list(range(5)))
>>> ring[2: 0: -1] = -1, -2, -3, -4
>>> ring
Ring([-4, 1, -1, -2, -3])
>>> ring = Ring(list(range(5)))
>>> ring[4: 1: -1] = -1, -2, -3
>>> ring
Ring([-2, -3, 2, 3, -1])
>>> ring = Ring(list(range(5)))
>>> ring[4: 0: -1] = (-1, -2)    # Note: fixed from original version
>>> ring
Ring([-2, 1, 2, 3, -1])
>>> ring = Ring(list(range(5)))
>>> ring[3: 3: -1] = -1          # Note: fixed from original version
>>> ring
Ring([0, 1, 2, -1, 4])
>>> ring = Ring(list(range(10)))
>>> ring[1: 7: -2] = -1, -2, -3, -4
>>> ring
Ring([0, -1, 2, -2, 4, -3, 6, -4, 8, 9])
>>> ring = Ring(list(range(10)))
>>> ring[7: 2: -2] = -1, -2, -3
>>> ring
Ring([0, -3, 2, 3, 4, 5, 6, -1, 8, -2])
```

Finally, we want `Ring` objects to be generators—they can be used only once. Starting point and increment, set by default to 0 and 1, respectively, can be changed, and iteration will proceed till we reach an element that has already been generated:

```
>>> ring = Ring(list(range(5)))
>>> for i in ring:
...     print(i)
...
0
1
2
3
4
>>> ring = Ring(tuple(range(5)))
>>> ring.start = 2
>>> ring.step = 2
>>> for i in ring:
...     print(i)
...
2
4
1
3
0
>>> ring = Ring(range(12))
>>> ring.start = 5
>>> ring.step = -3
>>> for i in ring:
...     print(i)
...
5
2
11
8
>>> ring = Ring(range(7))
>>> ring.start = 10
>>> ring.step = 4
>>> for i in ring:
...     print(i)
...
3
0
4
1
5
2
6
```

3 Solving tangrams (4 marks)

As expected and promised...

The aim is of course not to, for a given set of pieces and a shape, come up with a particular solution, but come up with *some* solution.

```
$ python3
...
>>> from tangram import *
>>> file = open('pieces_A.xml')
>>> coloured_pieces = available_coloured_pieces(file)
>>> file = open('shape_A_1.xml')
>>> shape = available_coloured_pieces(file)
>>> solution = solve(coloured_pieces, shape)
>>> if solution:
...     output_file = open('tangram_A_1_a.xml', 'w')
...     write_coloured_pieces_to_file(solution, output_file)
...     output_file.close()
... else:
...     print('No solution could be found.')
...
... file = open('tangram_A_1_a.xml')
>>> tangram = available_coloured_pieces(file)
>>> is_solution(tangram , shape)
True
```