

CS2006 Python Project 1 - Classes and Iterators in Python

Matriculation Numbers: 160001362, 160014817, 160013384 (Group 14)

06/03/2018

Contents

1	Summary of Functionality	3
1.1	Basic Specification:	3
1.2	Additional Requirements:	3
1.2.1	Easy	3
1.2.2	Medium	4
1.2.3	Hard	4
1.3	Further Features:	5
2	Design and Implementation	5
3	Evidence of Testing	7
4	Known Problems	7
5	Problems Overcome	8
6	Summary of Provenance	8
6.1	Code Implemented by the Group	8
6.2	Code Modified From the Provided Example Code	8
6.3	Code Sourced From Elsewhere	8
7	Conclusion	8

1 Summary of Functionality

This practical specified the development of a mathematical system to explore discrete mathematical structures using the programming language Python.

The provided README file gives detailed instructions on what the solution can do and how to run each command.

The following functionality has been implemented:

1.1 Basic Specification:

All requirements from the basic specification have been implemented. They are as follows:

- An implementation of the twisted integers data structure which supports addition and multiplication given by the following rules:
 - $a \oplus b = (a + b) \bmod n$
 - $a \otimes b = (a + b + a \cdot b) \bmod n$
- Exceptions that are used to check that user input is valid.

1.2 Additional Requirements:

From the suggested additional requirements, all of the easy, medium and hard requirements have been implemented:

1.2.1 Easy

- **Easy Requirement 1** - The function 'mulEqualToOne(n)' has been developed in the checker.py file which calculates for a given n all elements $x \in \mathbb{Z}_n$ such that $x \otimes x = 1$, where $1 \in \mathbb{Z}_n$
- **Easy Requirement 2** - For a given n functions (in checker.py) have been developed to check whether the following properties hold for all $x, y, z \in \mathbb{Z}_n$ (each of which returns a boolean signifying whether the properties hold):

- $x \oplus y = y \oplus x$ - 'isCommutativeAdd(n)'
- $x \otimes y = y \otimes x$ - 'isCommutativeMul(n)'
- $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ - 'isAssociativeAdd(n)'
- $(x \otimes y) \otimes z = x \otimes (y \otimes z)$ - 'isAssociativeMul(n)'
- $(x \oplus y) \otimes z = (x \otimes y) \oplus (y \otimes z)$ - 'isDistributive(n)'

The first two of the above five functions have been optimised to avoid redundant checks.

- **Easy Requirement 3** - The file `twisted_integers.py` contains a class `TwistedIntegers` which implements a data structure representing \mathbb{Z}_n with respect to the operations $a \oplus b = (a \oplus b) \bmod n$ and $a \otimes b = (a \otimes b) \bmod n$ and contains the methods `__init__`, `__str__` and `size` where `size` returns the number of elements in \mathbb{Z}_n .

1.2.2 Medium

- **Medium Requirement 1** - In the `twisted_integers.py` file (along with the aforementioned `TwistedIntegers` class) the `IteratorOfTwistedIntegers` class has been implemented to iterate over instances of `TwistedIntegers` and contains an `__init__` function (to initialise it with a given instance of `TwistedIntegers`), a `hasNext()` boolean function (to indicate if the iterator has a next value) and a `next()` function (to return the next value in the iterator).
- **Medium Requirement 2** - The function `'findValAdd(n)'` in the `twisted_integers.py` file finds for a given n all elements τ of \mathbb{Z}_n such that $\tau \oplus x = x$ for all $x \in \mathbb{Z}_n$.
- **Medium Requirement 3** - The function `'findValMul(n)'` in the `twisted_integers.py` file finds for a given n all elements ε of \mathbb{Z}_n such that $\varepsilon \otimes x = x$ for all $x \in \mathbb{Z}_n$.

1.2.3 Hard

- **Hard Requirement 1** - In the `twisted_int_matrix.py` file the class `TwistedIntMatrix` has been implemented which represents a matrix of `TwistedInt` objects. The `__init__` function of the class takes an x and y dimension as well as a list of `TwistedInt` objects and creates a matrix containing the values in the list. The `__mul__` function implements matrix multiplication (between two given matrices) using the row by column rule and the \oplus and \otimes operators. This is achieved using the `calcDotProduct` function which calculates the dot product between a row and a column of the two matrices (which itself uses the `getCol` function to access a column of the second matrix).

For the same requirement, an algorithm has been implemented using the function `getPossibleMatrices(matrices)` which takes a list of M matrices and calculates all of the possible unique matrices that may be obtained by multiplying these matrices, i.e. the set $\{g_1 \ g_2 \ \dots \ g_k \mid k \in \mathbb{N}, g_i \in M\}$.

- **Hard Requirement 2** - The level of testing and documentation has been enhanced by including docstrings for every function and class, as well as example doctests within these docstrings that can be extracted and run to verify the correctness of the functions which they document.

1.3 Further Features:

The follow additional features have also been implemented:

- An iterator class, `IteratorOfTwistedIntMatrix`, has been implemented in `twisted_int_matrix.py` file which iterates over an instance of `TwistedIntMatrix` from top left to bottom right across the `TwistedInt` elements. As with the `IteratorOfTwistedIntegers` iterator, this iterator has a `hasNext()`, `next()` and `__init__` functions.
- TODO

2 Design and Implementation

- **Basic Specification - TwistedInt Data Structure (`twisted_int.py`)**
 - The `TwistedInt` data structure is implemented in `twisted_int.py`. It is initialised by passing an integer value 'val' and mod value 'n' into `__init__`. The constructor of the `TwistedInt` class will raise a `TypeError` exception if 'val' or 'n' are not integers and will raise the custom `InvalidModError` or `InvalidValError` if the value of n passed in is less than 0 or if val is not in \mathbb{Z}_n , 0 to (n - 1). These exceptions can be handled by any implementing code to validate the inputs of the user.

The printing of `TwistedInt` objects has be modified in the function `__str__` to printed `TwistedInt` instances in the format "<object:n>".

`TwistedInt` multiplication is implemented by the `__mul__` function which takes two `TwistedInts` and compares their mod n values - raising a custom `MismatchedModError` exception if they are not equal - and otherwise performs the operation $(a + b + a \cdot b) \bmod n$ and assigns the result to the value of a new `TwistedInt` (as well as the same mod n value) and returns this object.

Similarly the `__add__` function performs the operation $(a + b) \bmod n$ with two given `TwistedInt` instances and returns a `TwistedInt` instance with the result as the integer value.

- **Easy Requirements 1 and 2 (`checker.py`)** - The '`mulEqualToOne(n)`' function implements easy requirement 1 of the specification. If the value of n passed into the function is either a string or is less than 0 then a `TypeError` exception or `InvalidModError` exception is raised respectively. If the value of n is valid then all possible values of $x \in \mathbb{Z}_n$ are iterated over and checked to see whether the condition $x \otimes x = 1$ holds. If the condition holds then the value of x is added to a list of valid ints which are returned at the end of the function.

For the second easy requirement each of the mathematical properties

is tested using the respective functions `isCommutativeAdd`, `isCommutativeMul`, `isAssociativeAdd`, `isAssociativeMul` and `isDistributive`. Each of the functions iterate over all possible values of x , y , and z using for loops and evaluate their respective conditions. The first two functions (having only two variables to test) have been optimised to avoid redundant checks using the function `combinations_with_replacement` from the `itertools` from the Python standard library.

- **Easy Requirement 3 (`twisted_integers.py`)** - The `TwistedIntegers` class implements the `TwistedIntegers` data structure that represents \mathbb{Z}_n for a given value of n . If the value n given is a string or is less than 0 then `TypeError` and `InvalidModError` exceptions are raised. Otherwise the class iterates over all possible values in \mathbb{Z}_n and assigns a `TwistedInt` instance for each into a list attribute. The `__str__` function generates output of all the `TwistedInt` instances using the `IteratorOfTwistedIntegers` iterator implemented for Medium Requirement 1. The `size` function returns the value of n of the `TwistedIntegers` instance.
- **Medium Requirement 1 (`twisted_integer.py`)** - The `IteratorOfTwistedIntegers` class implements the ability to iterate over a `TwistedIntegers` instance. The `hasNext` function in the iterator returns true if the current index of the iterator is less than the size of the `TwistedIntegers` instance in the iterator. The `next` function checks if the `hasNext` is false and if this is the case then an `IndexError` is raised. Otherwise the element at the current index is returned and the index of the iterator is incremented.
- **Medium Requirement 2 and 3 (`twisted_integers.py`)** - The function `findValAdd` finds for a given n all elements τ of \mathbb{Z}_n such that $\tau \oplus x = x$ for all $x \in \mathbb{Z}_n$. It does this by creating two `IteratorOfTwistedIntegers` instances and iterates over all values of τ in the outer loop and all possible values of x in the inner loop, each time checking whether the condition is true for those values of τ and x . If the condition holds for all values of x then the value of τ is added to a list of valid integers, otherwise the inner loop breaks to the outer loop. The list is returned at the end of the function.

Similarly using the same technique the function `findValMul` finds for a given n all elements ε of \mathbb{Z}_n such that $\varepsilon \otimes x = x$ for all $x \in \mathbb{Z}_n$.

- **Hard Requirement 1 (`twisted_int_matrix.py`)** - The `TwistedIntMatrix` class implements the functionality to create a matrix of `TwistedInt` instances by specifying an x dimension, y dimension and list of `TwistedInts` to populate the matrix. The constructor function for the class raises a custom `MatrixIndexError` exception if the size of the x or y dimensions are less than 0. It also raises a `MatrixIndexError` if the size of the matrix is smaller than the number of elements in the list of `TwistedInts`. Finally it throws as a `MismatchedModError` exception if not all of the `TwistedInts` have the same mod value of n . If it is a valid instantiation then

the class iterates over a matrix attribute and appends the values from the TwistedInt list to the correct index in the matrix.

The `__str__` function allows the matrix to be printed as it uses an iterator to traverse the matrix and produce a string output of the elements within.

The `__mul__` function allows traditional matrix row-column multiplication using the \oplus and \otimes operations. This is achieved by iterating over the matrices to be multiplied and calculating the dot product of each pair of rows and columns using the `calcDotProduct` function and the `getCol` functions. Appropriate exceptions are thrown (`TypeError`, `ValueError` and `MismatchedModError`) if the type of the second argument is not a matrix, the x and y dimensions of the matrices do not allow for multiplication, or the mod value of the two matrices are not equal, respectively.

The `getPossibleMatrices` function takes a list of matrices and uses the `permutations` function from `itertools` from the Python standard library to find all unique possible orderings of list indices for the matrix list. Each permutation is then iterated over and for each index in the permutation, the matrix at that index is accessed and multiplied by the following matrices in that same permutation. The result of this is added to a list of generated matrices.

- **Hard Requirement 2** - Extensive docStrings have been added to every function of the code. These can be seen inside the actual code, as well as inside the documentation folder, as HTML files. These also contain docTests - while not as extensive as unit testing, these provide an example of how to use every function, and test that the basic functionality is correct. These can be run in the terminal. The outputs of running all these tests has been stored in `docTest results.txt`.

3 Evidence of Testing

4 Known Problems

For the algorithm of hard requirement 1 that aims to return a set of unique matrices produced as a result of multiplying a list of matrices in different permutations, the list of matrices produced contain repeated values. This is because for some reason when the `contains` function of the `twisted_int_matrix`

5 Problems Overcome

6 Summary of Provenance

6.1 Code Implemented by the Group

6.2 Code Modified From the Provided Example Code

6.3 Code Sourced From Elsewhere

7 Conclusion