Started on	Thursday, 28 September 2023, 6:50 PM
State	Finished
Completed on	Saturday, 30 September 2023, 6:13 PM
Time taken	1 day 23 hours
Marks	6.00/6.00
Grade	**1.00** out of 1.00 (**100**%)

Information

Introduction

Finding patterns in sequences of numbers is considered an intellectually-challenging problem. These problems are commonly used in "IQ" tests. For example given the sequence 3, 2, 3, 6, 11, 18, the question asks for the next number in the sequence.

Being challenging for humans, they are also considered good benchmarks for AI algorithms. Finding patterns in sequences is in fact a special case of the more-general problem of modelling and predicting time series with a wide range of applications from weather forecast to finding trends in markets.

The objective of this quiz is to use ideas from tree-representation of expressions, CSPs, optimisation, and population-based and evolutionary search techniques to create a program that given a sequence of integers, it can 'predicts' the rest of the sequence.

This super-quiz has multiple questions. Many of the questions ask for writing functions that can later be used in other parts and in the final predicting program.

Introduction to the representation and the search strategy

Consider the very simple sequence of numbers 1, 2, 3, 4, 5, 6. What is the most obvious pattern in this sequence and what is the next number? It is fair to say that the next number is 7 and that the most obvious (simplest) pattern is that the numbers are increasing one at a time. [There is a philosophical problem (known as generalisation fallacy) in this argument but we put that aside in this quiz.]

So how can we represent patterns? A good answer would be *by functions*. For example if *a(i)* denotes the number at position *i* in the sequence where *i* starts from zero, then one function (pattern) that matches the above sequence is *a(i) = a(i-1) + 1* which is a recursive function. The sequence can also be explained by function *a(i) = i + 1*. Therefore in order to automatically recognise a pattern in a sequence, the program must perform a search over the space of possible functions to find one that matches the given sequence. So what is a good representation for functions? When functions are mathematical expressions, the *prefix notation* provides a very useful and robust representation. For example the above pattern can be represented by the expression `(+ i 1)`.

Because of the complexity of the neighbourhood in the space of expressions, CSP or greedy local search algorithms are not applicable. A good choice is using evolutionary algorithms where each individual is an expression tree (prefix notation). In this assignment you will implement a tree-based (prefix notation) representation for expressions. This representation is powerful enough that will allow you to use a simple random search to solve many instances the problem of recognising patterns in sequence of numbers. You are welcome (but not required) to extend the search part of the assignment to a complete evolutionary algorithm.

Technical Notes

- Although it might be possible to skip some of the questions, it is recommended that you answer the questions in order. Some question specs are referred to in later questions.
- Similar to the previous quizzes, in order to avoid accidental break of dependency or forgetting import statements, you can have all of your functions in one file and every time just copy and paste the entire file in the answer box. However you have to make sure that the code in the body of your program (module) does not interfere with the output and tests (e.g. it does not print stuff etc.)

Information

Representation of expressions

Mathematical expressions can be seen as trees. For example the expression `4*y - 3` can be seen as the following tree.

```
      -
     / \
    /   \
   *     3
  / \
 /   \
4     y
```

We need an easy way of representing these trees in Python. A natural choice is to use nested lists where each list is a function application in prefix notation. For example the above tree can be represented as `['-', ['*', 4, 'y'], 3]`

Specification

We define an object to be an expression if it is either:

- a constant leaf: a Python integer (for example `3`);
- a variable leaf: a Python string (for example `'y'`) from a pre-specified set of *leaf symbols*;
- a Python list such that:
  - the list has exactly 3 elements; and
  - the first element is a string (for example `'*'`) from a pre-specified set of *function symbols*; and
  - the remaining two elements of the list are expressions themselves; these two serve as the arguments of the function.

Note that for simplicity we are assuming that all functions in these expressions are binary; that is, they take exactly two arguments.

Question 1

Correct

Mark 1.00 out of 1.00

Write a function of the form `is_valid_expression(object, function_symbols, leaf_symbols)` that takes an object as its first argument and tests whether it is a valid expression according to our definition of expressions in this assignment. The function must return `True` if the given object is valid expression, `False` otherwise.

The parameters of the function are:

- `object`: any Python object
- `function_symbols`: a collection (list, set, ...) of strings that are allowed to be used in function positions (internal nodes of the tree).
- `leaf_symbols`: a collection of strings that are allowed to be used as variable leaves.

Notes

1. This function needs to be recursive. The base cases are for leaf nodes.
2. This function can be written in less than 10 lines of code.
3. The built-in function `type` can be useful here.

For example:

Test	Result

`from student_answer import is_valid_expression`
 	
`function_symbols = ['f', '+']`
`leaf_symbols = ['x', 'y']`
`expression = 1`
 	
`print(is_valid_expression(expression, function_symbols, leaf_symbols))`
	`True`

`from student_answer import is_valid_expression`
 	
`function_symbols = ['f', '+']`
`leaf_symbols = ['x', 'y']`
`expression = 'y'`
 	
`print(is_valid_expression(`
`        expression, function_symbols, leaf_symbols))`
	`True`

`from student_answer import is_valid_expression`
 	
`function_symbols = ['f', '+']`
`leaf_symbols = ['x', 'y']`
`expression = 2.0`
 	
`print(is_valid_expression(`
`        expression, function_symbols, leaf_symbols))`
	`False`

`from student_answer import is_valid_expression`
 	
`function_symbols = ['f', '+']`
`leaf_symbols = ['x', 'y']`
`expression = ['f', 123, 'x']`
 	
`print(is_valid_expression(`
`        expression, function_symbols, leaf_symbols))`
	`True`

`from student_answer import is_valid_expression`
 	
`function_symbols = ['f', '+']`
`leaf_symbols = ['x', 'y']`
`expression = ['f', ['+', 0, -1], ['f', 1, 'x']]`
 	
`print(is_valid_expression(`
`        expression, function_symbols, leaf_symbols))`
	`True`

Test	Result

`from student_answer import is_valid_expression`
 	
`function_symbols = ['f', '+']`
`leaf_symbols = ['x', 'y']`
`expression = ['+', ['f', 1, 'x'], -1]`
 	
`print(is_valid_expression(`
`        expression, function_symbols, leaf_symbols))`
	`True`

`from student_answer import is_valid_expression`
 	
`function_symbols = ['f', '+']`
`leaf_symbols = ['x', 'y', -1, 0, 1]`
`expression = ['f', 0, ['f', 0, ['f', 0, ['f', 0, 'x']]]]`
 	
`print(is_valid_expression(`
`        expression, function_symbols, leaf_symbols))`
	`True`

`from student_answer import is_valid_expression`
 	
`function_symbols = ['f', '+']`
`leaf_symbols = ['x', 'y']`
`expression = 'f'`
 	
`print(is_valid_expression(`
`        expression, function_symbols, leaf_symbols))`
	`False`

`from student_answer import is_valid_expression`
 	
`function_symbols = ['f', '+']`
`leaf_symbols = ['x', 'y']`
`expression = ['f', 1, 0, -1]`
 	
`print(is_valid_expression(`
`        expression, function_symbols, leaf_symbols))`
	`False`

`from student_answer import is_valid_expression`
 	
`function_symbols = ['f', '+']`
`leaf_symbols = ['x', 'y']`
`expression = ['x', 0, 1]`
 	
`print(is_valid_expression(`
`        expression, function_symbols, leaf_symbols))`
	`False`

`from student_answer import is_valid_expression`
 	
`function_symbols = ['f', '+']`
`leaf_symbols = ['x', 'y']`
`expression = ['g', 0, 'y']`
 	
`print(is_valid_expression(`
`        expression, function_symbols, leaf_symbols))`
	`False`

Answer: (penalty regime: 0, 15, ... %)

```python
def is_valid_expression(object, function_symbols, leaf_symbols):
    """takes an object as its first argument and tests whether it is a valid
    expression according to our definition of expressions in this assignment.
    """
    #base case
    if type(object) is not list:
        if type(object) is str and object in leaf_symbols:
            return True
        if type(object) is int:
            return True
        return False
    if len(object) != 3:
        return False
    if object[0] not in function_symbols:
        return False

    #recursive
    return min(is_valid_expression(object[1], function_symbols, leaf_symbols),
               is_valid_expression(object[2], function_symbols, leaf_symbols))
```

	Test	Expected	Got	

✓	`from student_answer import is_valid_expression`
	
	`function_symbols = ['f', '+']`
	`leaf_symbols = ['x', 'y']`
	`expression = 1`
	
	`print(is_valid_expression(expression, function_symbols, leaf_symbols))`
		`True`	`True`	✓

✓	`from student_answer import is_valid_expression`
	
	`function_symbols = ['f', '+']`
	`leaf_symbols = ['x', 'y']`
	`expression = 'y'`
	
	`print(is_valid_expression(`
	`        expression, function_symbols, leaf_symbols))`
		`True`	`True`	✓

✓	`from student_answer import is_valid_expression`
	
	`function_symbols = ['f', '+']`
	`leaf_symbols = ['x', 'y']`
	`expression = 2.0`
	
	`print(is_valid_expression(`
	`        expression, function_symbols, leaf_symbols))`
		`False`	`False`	✓

✓	`from student_answer import is_valid_expression`
	
	`function_symbols = ['f', '+']`
	`leaf_symbols = ['x', 'y']`
	`expression = ['f', 123, 'x']`
	
	`print(is_valid_expression(`
	`        expression, function_symbols, leaf_symbols))`
		`True`	`True`	✓

✓	`from student_answer import is_valid_expression`
	
	`function_symbols = ['f', '+']`
	`leaf_symbols = ['x', 'y']`
	`expression = ['f', ['+', 0, -1], ['f', 1, 'x']]`
	
	`print(is_valid_expression(`
	`        expression, function_symbols, leaf_symbols))`
		`True`	`True`	✓

✓	`from student_answer import is_valid_expression`
	
	`function_symbols = ['f', '+']`
	`leaf_symbols = ['x', 'y']`
	`expression = ['+', ['f', 1, 'x'], -1]`
	
	`print(is_valid_expression(`
	`        expression, function_symbols, leaf_symbols))`
		`True`	`True`	✓

✓	`from student_answer import is_valid_expression`
	
	`function_symbols = ['f', '+']`
	`leaf_symbols = ['x', 'y', -1, 0, 1]`
	`expression = ['f', 0, ['f', 0, ['f', 0, ['f', 0, 'x']]]]`
	
	`print(is_valid_expression(`
	`        expression, function_symbols, leaf_symbols))`
		`True`	`True`	✓

	Test	Expected	Got	

✓	`from student_answer import is_valid_expression`
	
	`function_symbols = ['f', '+']`
	`leaf_symbols = ['x', 'y']`
	`expression = ['f', 1, 0, -1]`
	
	`print(is_valid_expression(`
	`        expression, function_symbols, leaf_symbols))`
		False	False	✓

✓	`from student_answer import is_valid_expression`
	
	`function_symbols = ['f', '+']`
	`leaf_symbols = ['x', 'y']`
	`expression = ['x', 0, 1]`
	
	`print(is_valid_expression(`
	`        expression, function_symbols, leaf_symbols))`
		False	False	✓

✓	`from student_answer import is_valid_expression`
	
	`function_symbols = ['f', '+']`
	`leaf_symbols = ['x', 'y']`
	`expression = ['g', 0, 'y']`
	
	`print(is_valid_expression(`
	`        expression, function_symbols, leaf_symbols))`
		False	False	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Question 2

Correct

Mark 1.00 out of 1.00

Write a function `depth(expression)` that takes an expression (that follows our definition of expression) and returns the depth of the expression tree. The depth of a tree is the depth of its deepest leaf.

Notes

1. This is a recursive function and can be written in 5 lines of code (or even fewer).
2. The depth of an expression that is just a single leaf (e.g. the expression `2` or `y`) is zero.
3. Converting the expression into a string and analysing the string to compute the depth is not a good idea.

For example:

Test	Result

`from student_answer import depth`
 	
`expression = 12`
`print(depth(expression))`
	`0`

`from student_answer import depth`
 	
`expression = 'weight'`
`print(depth(expression))`
	`0`

`from student_answer import depth`
 	
`expression = ['add', 12, 'x']`
`print(depth(expression))`
	`1`

`from student_answer import depth`
 	
`expression = ['add', ['add', 22, 'y'], 'x']`
`print(depth(expression))`
	`2`

Answer: (penalty regime: 0, 15, ... %)

```python
def depth(expression):
    """takes an expression (that follows our definition of expression) and
    returns the depth of the expression tree.
    The depth of a tree is the depth of its deepest leaf.
    """
    if type(expression) is not list:
        return 0
    return 1 + max(depth(expression[1]), depth(expression[2]))
```

	Test	Expected	Got	

✓	`from student_answer import depth`
	
	`expression = 12`
	`print(depth(expression))`
		`0`	`0`	✓

	Test	Expected	Got	

✓	`from student_answer import depth`
	
	`expression = 'weight'`
	`print(depth(expression))`
		0	0	✓

✓	`from student_answer import depth`
	
	`expression = ['add', 12, 'x']`
	`print(depth(expression))`
		1	1	✓

✓	`from student_answer import depth`
	
	`expression = ['add', ['add', 22, 'y'], 'x']`
	`print(depth(expression))`
		2	2	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Information

Evaluation of expressions

All expressions in this assignment evaluate to integers. These are the rules of evaluation:

- A constant leaf (an integer) evaluates to itself.
- A variable leaf (a string) is looked up in a dictionary of *bindings* which maps leaf symbols (strings) to integers.
- For a list, the following steps are taken:
  - The value of the function symbol is looked up in a dictionary of bindings. The function symbol are mapped to a binary function that takes two integers as its arguments and returns an integer.
  - The two remaining elements are evaluated.
  - The function is applied to the value of the two arguments. The returned value is the value of the list.

Question 3

Correct

Mark 1.00 out of 1.00

Write a function `evaluate(expression, bindings)` that takes an expression and a dictionary of bindings and returns an integer that is the value of the expression. The parameters of the function are:

- `expression`: an expression according to our definition of expressions;
- `bindings`: a dictionary where all the keys are strings and are either a function symbol or a variable leaf. A function symbol is mapped to a function that takes two arguments. A leaf symbol is mapped to an integer.

Note: this function is recursive and can be written in less than 10 lines of code.

For example:

Test	Result

`from student_answer import evaluate`
 	
`bindings = {}`
`expression = 12`
`print(evaluate(expression, bindings))`
	`12`

`from student_answer import evaluate`
 	
`bindings = {'x':5, 'y':10, 'time':15}`
`expression = 'y'`
`print(evaluate(expression, bindings))`
	`10`

`from student_answer import evaluate`
 	
`bindings = {'x': 5, 'y': 10, 'time': 15, 'add': lambda x, y: x + y}`
`expression = ['add', 12, 'x']`
`print(evaluate(expression, bindings))`
	`17`

`from student_answer import evaluate`
 	
`import operator`
 	
`bindings = dict(x = 5, y = 10, blah = 15, add = operator.add)`
`expression = ['add', ['add', 22, 'y'], 'x']`
`print(evaluate(expression, bindings))`
	`37`

Answer: (penalty regime: 0, 15, ... %)

```python
def evaluate(expression, bindings):
    """takes an expression and a dictionary of bindings and
    returns an integer that is the value of the expression.
    """
    if type(expression) is not list:
        if type(expression) is str:
            return bindings[expression]
        return expression

    operator = bindings[expression[0]]
    return operator(evaluate(expression[1], bindings), evaluate(expression[2], bi
```

	Test	Expected	Got	

✓	`from student_answer import evaluate`
	
	`bindings = {}`
	`expression = 12`
	`print(evaluate(expression, bindings))`
		12	12	✓

✓	`from student_answer import evaluate`
	
	`bindings = {'x':5, 'y':10, 'time':15}`
	`expression = 'y'`
	`print(evaluate(expression, bindings))`
		10	10	✓

✓	`from student_answer import evaluate`
	
	`bindings = {'x': 5, 'y': 10, 'time': 15, 'add': lambda x, y: x + y}`
	`expression = ['add', 12, 'x']`
	`print(evaluate(expression, bindings))`
		17	17	✓

✓	`from student_answer import evaluate`
	
	`import operator`
	
	`bindings = dict(x = 5, y = 10, blah = 15, add = operator.add)`
	`expression = ['add', ['add', 22, 'y'], 'x']`
	`print(evaluate(expression, bindings))`
		37	37	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Question 4

Correct

Mark 1.00 out of 1.00

Write a function `random_expression(function_symbols, leaves, max_depth)` that randomly generates an expression. The function takes the following arguments:

- `function_symbols`: a list of function symbols (strings)
- `leaves`: a list of constant and variable leaves (integers and strings)
- `max_depth`: a non-negative integer that specifies the maximum depth allowed for the generated expression.

The function will be called 10,000 times to generate these many expressions. Then the following tests are performed on the generated expressions:

- All the generated expressions must be valid expressions constructed from the given function symbols and leaves.
- Out of the 10,000 generated expressions, at least 1000 must be syntactically distinct. The semantic of expressions is disregarded when testing for distinctness. For example `['+', 1, 2]` and `['+', 2, 1]` will be regarded as different expressions.
- Out of the 10,000 generated expressions, at least 100 must be of depth 0, at least 100 of depth 1, ..., and at least 100 must be of depth `max_depth` (which is set to 4 in the test cases).

Notes

- This function is recursive.
- I suggest an implementation along these lines: Toss a coin. If it's a head (or if some other condition that you have to determine is satisfied) return a leaf node, otherwise return a random expression tree (a 3-element list). If the latter, you also need to randomly generate its two arguments.
- Consider using functions provided in the `random` module.
- You can implement this function in about 15 lines of code
- In order to be able to replicate your results (for example for debugging purposes), consider using `random.seed(some_integer)` which will cause the generators to always produce the same sequence of random numbers and as a result your program will always produce the same output.
- It is recommended that you test your code locally. You should consider developing your own test code that checks the stated requirements.

For example:

Test	Result

```
from student_answer import random_expression

# All the generated expressions must be valid

function_symbols = ['f', 'g', 'h']
constant_leaves =  list(range(-2, 3))
variable_leaves = ['x', 'y', 'i']
leaves = constant_leaves + variable_leaves
max_depth = 4

for _ in range(10000):
    expression = random_expression(function_symbols, leaves, max_depth)
    if not _is_valid_expression(expression, function_symbols, leaves):
        print("The following expression is not valid:\n", expression)
        break
else:
    print("OK")
```
	OK

```
from student_answer import random_expression

function_symbols = ['f', 'g', 'h']
leaves = ['x', 'y', 'i'] + list(range(-2, 3))
max_depth = 4

expressions = [random_expression(function_symbols, leaves, max_depth)
               for _ in range(10000)]

# Out of 10000 expressions, at least 1000 must be distinct
_check_distinctness(expressions)
```
	OK

Test	Result

```
from student_answer import random_expression

function_symbols = ['f', 'g', 'h']
leaves = ['x', 'y', 'i'] + list(range(-2, 3))
max_depth = 4

expressions = [random_expression(function_symbols, leaves, max_depth)
               for _ in range(10000)]

# Out of 10000 expressions, there must be at least 100 expressions
# of depth 0, 100 of depth 1, ..., and 100 of depth 4.

_check_diversity(expressions, max_depth)
```
	OK

Answer: (penalty regime: 0, 15, ... %)

```python
import random
random.seed(0)

def random_expression(function_symbols, leaves, max_depth):
    """randomly generates an expression.
    function_symbols: a list of function symbols (strings).
    leaves: a list of constant and variable leaves (integers and strings).
    max_depth: a non-negative integer that specifies the maximum depth
    allowed for the generated expression.
    """
    if max_depth == 0:
        return leaves[random.randint(0, len(leaves)-1)]

    if random.randint(0, 1):
        return leaves[random.randint(0, len(leaves)-1)]
    else:
        return [function_symbols[random.randint(0, len(function_symbols)-1)],
                random_expression(function_symbols, leaves, max_depth-1),
                random_expression(function_symbols, leaves, max_depth-1)]
```

	Test	Expected	Got	

✓	
	```
	from student_answer import random_expression

	# All the generated expressions must be valid

	function_symbols = ['f', 'g', 'h']
	constant_leaves =  list(range(-2, 3))
	variable_leaves = ['x', 'y', 'i']
	leaves = constant_leaves + variable_leaves
	max_depth = 4

	for _ in range(10000):
	    expression = random_expression(function_symbols, leaves, max_depth)
	    if not _is_valid_expression(expression, function_symbols, leaves):
	        print("The following expression is not valid:\n", expression)
	        break
	else:
	    print("OK")
	```
		OK	OK	✓

✓	
	```
	from student_answer import random_expression

	function_symbols = ['f', 'g', 'h']
	leaves = ['x', 'y', 'i'] + list(range(-2, 3))
	max_depth = 4

	expressions = [random_expression(function_symbols, leaves, max_depth)
	               for _ in range(10000)]

	# Out of 10000 expressions, at least 1000 must be distinct
	_check_distinctness(expressions)
	```
		OK	OK	✓

	Test	Expected	Got	

✓	
	```
	from student_answer import random_expression

	function_symbols = ['f', 'g', 'h']
	leaves = ['x', 'y', 'i'] + list(range(-2, 3))
	max_depth = 4

	expressions = [random_expression(function_symbols, leaves, max_depth)
	               for _ in range(10000)]

	# Out of 10000 expressions, there must be at least 100 expressions
	# of depth 0, 100 of depth 1, ..., and 100 of depth 4.

	_check_diversity(expressions, max_depth)
	```
		OK	OK	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Information

Describing sequence patterns by expressions

Let's represent a sequence of integers with a[0], a[1], a[2], ..., a[i]. If a sequence is such that for i > 1 we have a[i] = f(x, y, i) where x=a[i-2], y=a[i-1], and f is a function (expression involving i, x, or y), then the function describes the pattern of the sequence. In other words, the elements of the sequence from the third element onwards can be generated by the expression.

Example 1: The sequence 0, 1, 2, 3, 4, 5, 6,... can be described by f(x, y, i) = i or f(x, y, i)=y+1 or f(x, y, i) = x + 2.

Example 2: The sequence 1, 4, 9, 16, 25,... can be described by f(x, y, i) = (i+1)*(1+i).

Example 3: The sequence 0, 1, 1, 2, 3, 5, 8, 13, ..., the Fibonacci sequence, can be described by f(x, y, i) = x + y.

Question 5

Correct

Mark 1.00 out of 1.00

Write a function `generate_rest(initial_sequence, expression, length)` that takes an initial sequence of numbers, an expression, and a specified length, and returns a list of integers with the specified length that is the continuation of the initial list according to the given expression. The parameters are:

- `initial_sequence`: an initial sequence (list) of integer numbers that has at least two numbers;
- `expression`: an expression constructed from function symbols `'+'`, `'-'`, and `'*'` which correspond to the three binary arithmetic functions, and the leaf nodes are integers and `'x'`, `'y'`, and `'i'` where the intended meaning of these three symbols is described above;
- `length`: a non-negative integer that specifies the length of the returned list.

Hint: The values must be generated in order, from left to right. For the first value, the expression must be evaluated with `'i'` set to the length of the initial sequence (because this would be the index of the first number in the generated sequence) and `'x'` and `'y'` set to the last two elements of the initial sequence. As new values are generated, the values of i, x, and y are updated. Recall that values of variable leaves are set via a dictionary of bindings.

Note: It is recommended that you use your implementation of the `evaluate` function. This would allow you to implement this function in about 11 lines of code.

For example:

Test	Result

```
from student_answer import generate_rest

initial_sequence = [0, 1, 2]
expression = 'i'
length_to_generate = 5
print(generate_rest(initial_sequence,
                    expression,
                    length_to_generate))
```
	`[3, 4, 5, 6, 7]`

```
from student_answer import generate_rest

# no particular pattern, just an example expression
initial_sequence = [-1, 1, 367]
expression = 'i'
length_to_generate = 4
print(generate_rest(initial_sequence,
                    expression,
                    length_to_generate))
```
	`[3, 4, 5, 6]`

```
from student_answer import generate_rest

initial_sequence = [4, 6, 8, 10]
expression = ['*', ['+', 'i', 2], 2]
length_to_generate = 5
print(generate_rest(initial_sequence,
                    expression,
                    length_to_generate))
```
	`[12, 14, 16, 18, 20]`

```
from student_answer import generate_rest

initial_sequence = [4, 6, 8, 10]
expression = ['+', 2, 'y']
length_to_generate = 5
print(generate_rest(initial_sequence,
                    expression,
                    length_to_generate))
```
	`[12, 14, 16, 18, 20]`

```
from student_answer import generate_rest

initial_sequence = [0, 1]
expression = 'x'
length_to_generate = 6
print(generate_rest(initial_sequence,
                    expression,
                    length_to_generate))
```
	`[0, 1, 0, 1, 0, 1]`

Test	Result

`from student_answer import generate_rest`
 	
`# Fibonacci sequence`
`initial_sequence = [0, 1]`
`expression = ['+', 'x', 'y']`
`length_to_generate = 5`
`print(generate_rest(initial_sequence,`
`                    expression,`
`                    length_to_generate))`
	`[1, 2, 3, 5, 8]`

`from student_answer import generate_rest`
 	
`initial_sequence = [367, 367, 367]`
`expression = 'y'`
`length_to_generate = 5`
`print(generate_rest(initial_sequence,`
`                    expression,`
`                    length_to_generate))`
	`[367, 367, 367, 367, 367]`

`from student_answer import generate_rest`
 	
`# no pattern, just a demo`
`initial_sequence = [0, 1, 2]`
`expression = -1`
`length_to_generate = 5`
`print(generate_rest(initial_sequence,`
`                    expression,`
`                    length_to_generate))`
	`[-1, -1, -1, -1, -1]`

`from student_answer import generate_rest`
 	
`initial_sequence = [0, 1, 2]`
`expression = 'i'`
`length_to_generate = 0`
`print(generate_rest(initial_sequence,`
`                    expression,`
`                    length_to_generate))`
	`[]`

Answer: (penalty regime: 0, 15, ... %)

```python
def evaluate(expression, bindings):
    """takes an expression and a dictionary of bindings and
    returns an integer that is the value of the expression.
    """
    if type(expression) is not list:
        if type(expression) is str:
            return bindings[expression]
        return expression

    operator = bindings[expression[0]]
    return operator(evaluate(expression[1], bindings), evaluate(expression[2],

def generate_rest(initial_sequence, expression, length):
    """takes an initial sequence of numbers, an expression, and a specified
    length, and returns a list of integers with the specified length that
    is the continuation of the initial list according to the given expression.
    """
    whole_sequence = initial_sequence[:]
    for j in range(length):
        current_index = len(initial_sequence)+j
        bindings = {'x': whole_sequence[current_index-2],
                    'y': whole_sequence[current_index-1],
                    'i': current_index,
                    '+': lambda x, y: x+y,
                    '-': lambda x, y: x-y,
                    '*': lambda x, y: x*y,
                    }
        whole_sequence.append(evaluate(expression, bindings))
    return whole_sequence[len(initial_sequence):]
```

	Test	Expected	Got	

✓	from student_answer import generate_rest
	
	initial_sequence = [0, 1, 2]
	expression = 'i'
	length_to_generate = 5
	print(generate_rest(initial_sequence,
	                    expression,
	                    length_to_generate))
		[3, 4, 5, 6, 7]	[3, 4, 5, 6, 7]	✓

✓	from student_answer import generate_rest
	
	# no particular pattern, just an example expression
	initial_sequence = [-1, 1, 367]
	expression = 'i'
	length_to_generate = 4
	print(generate_rest(initial_sequence,
	                    expression,
	                    length_to_generate))
		[3, 4, 5, 6]	[3, 4, 5, 6]	✓

✓	from student_answer import generate_rest
	
	initial_sequence = [4, 6, 8, 10]
	expression = ['*', ['+', 'i', 2], 2]
	length_to_generate = 5
	print(generate_rest(initial_sequence,
	                    expression,
	                    length_to_generate))
		[12, 14, 16, 18, 20]	[12, 14, 16, 18, 20]	✓

✓	from student_answer import generate_rest
	
	initial_sequence = [4, 6, 8, 10]
	expression = ['+', 2, 'y']
	length_to_generate = 5
	print(generate_rest(initial_sequence,
	                    expression,
	                    length_to_generate))
		[12, 14, 16, 18, 20]	[12, 14, 16, 18, 20]	✓

✓	from student_answer import generate_rest
	
	initial_sequence = [0, 1]
	expression = 'x'
	length_to_generate = 6
	print(generate_rest(initial_sequence,
	                    expression,
	                    length_to_generate))
		[0, 1, 0, 1, 0, 1]	[0, 1, 0, 1, 0, 1]	✓

✓	from student_answer import generate_rest
	
	# Fibonacci sequence
	initial_sequence = [0, 1]
	expression = ['+', 'x', 'y']
	length_to_generate = 5
	print(generate_rest(initial_sequence,
	                    expression,
	                    length_to_generate))
		[1, 2, 3, 5, 8]	[1, 2, 3, 5, 8]	✓

✓	from student_answer import generate_rest
	
	initial_sequence = [367, 367, 367]
	expression = 'y'
	length_to_generate = 5
	print(generate_rest(initial_sequence,
	                    expression,
	                    length_to_generate))
		[367, 367, 367, 367, 367]	[367, 367, 367, 367, 367]	✓

	Test	Expected	Got	

✓	`from student_answer import generate_rest`
	
	`# no pattern, just a demo`
	`initial_sequence = [0, 1, 2]`
	`expression = -1`
	`length_to_generate = 5`
	`print(generate_rest(initial_sequence,`
	`                    expression,`
	`                    length_to_generate))`
		`[-1, -1, -1, -1, -1]`	`[-1, -1, -1, -1, -1]`	✓

✓	`from student_answer import generate_rest`
	
	`initial_sequence = [0, 1, 2]`
	`expression = 'i'`
	`length_to_generate = 0`
	`print(generate_rest(initial_sequence,`
	`                    expression,`
	`                    length_to_generate))`
		`[]`	`[]`	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Question 6

Correct

Mark 1.00 out of 1.00

Finally you are ready to put all the components together and solve the actual problem. You have to write a function `predict_rest(sequence)` that takes a sequence of integers of length at least 5, finds the pattern in the sequence, and "predicts" the rest by returning a list of the next five integers in the sequence.

This question is somewhat open-ended; it is up to you what algorithm you implement. The patterns in the test cases are easy enough that even a random search (i.e. generating random expressions until a match is found) is very likely to solve all the test cases. This demonstrates that a good representation can significantly simplify a problem. If you are interested in developing a more sophisticated solution, then you can look into implementing a proper evolutionary algorithm with operators for mutation and crossover. See the lecture notes and the related links.

Further assumptions

1. All the sequences in the test cases have patterns that can be expressed as a function of *x*, *y*, and *i* as described before.
2. All the patterns (functions) can be constructed by combining three binary functions: addition, subtraction, and multiplication.
3. Using integers between -2 and 2 (inclusive) as constant leafs should be enough to represent the patterns in the test cases.
4. All the patterns (functions) can be constructed by expression trees not deeper than 3.
5. Your algorithm must be able to solve all the problems given in the example test cases in less than 2 seconds (collectively). Following the guidelines given in the assignment should naturally achieve this.

Notes

1. To make sure your function does not overfit (e.g. memorise) the given example sequences, there are some hidden test cases (which are not more difficult than the example test cases).
2. Consider using `random.seed` so that your offline results match what will be generated on the server and you can replicate your results if needed.
3. Make sure the function does not modify its input.
4. In its simplest form, this function can be implemented in less than 10 lines of code.
5. It would be interesting to go through the test cases yourself and see if you can find the patterns.

For example:

Test	Result

`from student_answer import predict_rest`
 	
`sequence = [0, 1, 2, 3, 4, 5, 6, 7]`
`the_rest = predict_rest(sequence)`
`print(sequence)`
`print(the_rest)`
	`[0, 1, 2, 3, 4, 5, 6, 7]`
	`[8, 9, 10, 11, 12]`

`from student_answer import predict_rest`
 	
`sequence = [0, 2, 4, 6, 8, 10, 12, 14]`
`print(predict_rest(sequence))`
	`[16, 18, 20, 22, 24]`

`from student_answer import predict_rest`
 	
`sequence = [31, 29, 27, 25, 23, 21]`
`print(predict_rest(sequence))`
	`[19, 17, 15, 13, 11]`

`from student_answer import predict_rest`
 	
`sequence = [0, 1, 4, 9, 16, 25, 36, 49]`
`print(predict_rest(sequence))`
	`[64, 81, 100, 121, 144]`

`from student_answer import predict_rest`
 	
`sequence = [3, 2, 3, 6, 11, 18, 27, 38]`
`print(predict_rest(sequence))`
	`[51, 66, 83, 102, 123]`

`from student_answer import predict_rest`
 	
`sequence =  [0, 1, 1, 2, 3, 5, 8, 13]`
`print(predict_rest(sequence))`
	`[21, 34, 55, 89, 144]`

`from student_answer import predict_rest`
 	
`sequence = [0, -1, 1, 0, 1, -1, 2, -1]`
`print(predict_rest(sequence))`
	`[5, -4, 29, -13, 854]`

Test	Result

`from student_answer import predict_rest`
 	
`sequence = [1, 3, -5, 13, -31, 75, -181, 437]`
`print(predict_rest(sequence))`
	`[-1055, 2547, -6149, 14845, -35839]`

Answer: (penalty regime: 0, 15, ... %)

```python
import random
random.seed(0)

def random_expression(function_symbols, leaves, max_depth):
    """randomly generates an expression.
    function_symbols: a list of function symbols (strings).
    leaves: a list of constant and variable leaves (integers and strings).
    max_depth: a non-negative integer that specifies the maximum depth
    allowed for the generated expression.
    """
    if max_depth == 0:
        return leaves[random.randint(0, len(leaves)-1)]

    if random.randint(0, 1):
        return leaves[random.randint(0, len(leaves)-1)]
    else:
        return [function_symbols[random.randint(0, len(function_symbols)-1)],
                random_expression(function_symbols, leaves, max_depth-1),
                random_expression(function_symbols, leaves, max_depth-1)]

def evaluate(expression, bindings):
    """takes an expression and a dictionary of bindings and
    returns an integer that is the value of the expression.
    """
    if type(expression) is not list:
        if type(expression) is str:
            return bindings[expression]
        return expression

    operator = bindings[expression[0]]
    return operator(evaluate(expression[1], bindings), evaluate(expression[2],

def generate_rest(initial_sequence, expression, length):
    """takes an initial sequence of numbers, an expression, and a specified
    length, and returns a list of integers with the specified length that
    is the continuation of the initial list according to the given expression.
    """
    whole_sequence = initial_sequence[:]
    for j in range(length):
        current_index = len(initial_sequence)+j
        bindings = {'x': whole_sequence[current_index-2],
                    'y': whole_sequence[current_index-1],
                    'i': current_index,
                    '+': lambda x, y: x+y,
                    '-': lambda x, y: x-y,
                    '*': lambda x, y: x*y,
                    }
        whole_sequence.append(evaluate(expression, bindings))
    return whole_sequence[len(initial_sequence): ]

def predict_rest(sequence):

```

	Test	Expected	Got	

✓	`from student_answer import predict_rest`
	
	`sequence = [0, 1, 2, 3, 4, 5, 6, 7]`
	`the_rest = predict_rest(sequence)`
	`print(sequence)`
	`print(the_rest)`
		`[0, 1, 2, 3, 4, 5, 6, 7]`
		`[8, 9, 10, 11, 12]`
			`[0, 1, 2, 3, 4, 5, 6, 7]`
			`[8, 9, 10, 11, 12]`
				✓

	Test	Expected	Got	

✓	`from student_answer import predict_rest`
	
	`sequence = [0, 2, 4, 6, 8, 10, 12, 14]`
	`print(predict_rest(sequence))`
		`[16, 18, 20, 22, 24]`	`[16, 18, 20, 22, 24]`	✓

✓	`from student_answer import predict_rest`
	
	`sequence = [31, 29, 27, 25, 23, 21]`
	`print(predict_rest(sequence))`
		`[19, 17, 15, 13, 11]`	`[19, 17, 15, 13, 11]`	✓

✓	`from student_answer import predict_rest`
	
	`sequence = [0, 1, 4, 9, 16, 25, 36, 49]`
	`print(predict_rest(sequence))`
		`[64, 81, 100, 121, 144]`	`[64, 81, 100, 121, 144]`	✓

✓	`from student_answer import predict_rest`
	
	`sequence = [3, 2, 3, 6, 11, 18, 27, 38]`
	`print(predict_rest(sequence))`
		`[51, 66, 83, 102, 123]`	`[51, 66, 83, 102, 123]`	✓

✓	`from student_answer import predict_rest`
	
	`sequence =  [0, 1, 1, 2, 3, 5, 8, 13]`
	`print(predict_rest(sequence))`
		`[21, 34, 55, 89, 144]`	`[21, 34, 55, 89, 144]`	✓

✓	`from student_answer import predict_rest`
	
	`sequence = [0, -1, 1, 0, 1, -1, 2, -1]`
	`print(predict_rest(sequence))`
		`[5, -4, 29, -13, 854]`	`[5, -4, 29, -13, 854]`	✓

✓	`from student_answer import predict_rest`
	
	`sequence = [1, 3, -5, 13, -31, 75, -181, 437]`
	`print(predict_rest(sequence))`
		`[-1055, 2547, -6149, 14845, -35839]`	`[-1055, 2547, -6149, 14845, -35839]`	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Information

Final notes

This assignment demonstrates the importance of representation in AI and Machine Learning. We devised a functional form to represent the underlying pattens of sequences of numbers. The functions themselves where represented as trees. This allowed us to define operations such as generation (and crossover and mutation) to perform an (evolutionary) search. Python and the natural way in which it handles references, allowed us to represent trees as nested lists with elements pointing to objects such as functions, integers, and other lists.

If you have managed to answer the last question correctly, then you should be able to solve many similar problems available on the web. As an example take a look at http://www.fibonicci.com/math/number-sequences-test/ (commercial website) and see how many of the problems in the "hard" category you can solve (first yourself, then your program). Your program should be able to solve quite a number of them with ease - specially if you have implemented an evolutionary search.

Some problems (number sequences) require richer search spaces. For some problems you may need to allow for deeper expression trees. For some, you may need a different function set. Some problems may need a more elaborate representation for functions (not just *x*, *y*, and *i*). One may consider performing a *meta-search* (on top of a normal search) to find the right settings for a problem. For some sequences, however, like the sequence of prime numbers, there is no right settings in the current model; you need a whole new way of thinking about sequences and introduce more sophisticated models/machines. Unsurprisingly, as we allow for more sophisticated models, the search space becomes bigger (often exponentially), and the expected time to find a solution increases.