Lesson 3 – Recursion

Definition

- Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get a small enough problem that can be solved trivially.
- Recursion involves a function calling itself
- Recursion is an elegant way to solve difficult problems

Example: Calculating the sum of a list of numbers

The normal way we have learned in the past

- iterative solution
- define a variable `theSum` and iteratively add the element into the `theSum`

```
def listSum(numList):
        """
        calculating the sum of a list
        """
        theSum = 0
        for i in numList:
                theSum = theSum + i
        return theSum
```

Analysis: How do you compute the sum of a list of numbers? You will start by recalling the addition function of 2 numbers, and iteratively calling addition to the next number. It looks like this:

(((((1 + 3) + 5) + 7) + 9)

this is the same as

(1 + (3 + (5 + (7 + 9))))

the inner part (7 + 9) is the problem that we can solve without a loop. And we can construct the problem by writing down like this

- total = (1 + (3 + (5 + (7 + 9))))
- total = (1 + (3 + (5 + 16)))
- total = (1 + (3 + 21))
- total = (1 + 24)
- total = 25

In another way, we can say that the sum of the list numList is the sum of the 1st element (`numList[0]`) and the rest of the element (`numList[1:]`).

```
listSum(numList) = numList[0] + listSum(numList[1:])
```

```
def listSum(numList):
        """
        use recursion to solve
        """
        if len(numList) == 1:
                return numList[0]
        else:
                return numList[0] + listSum(numList[1:])
```

Three laws of recursive algorithms:

- A recursive algorithm must have a **base case**
- A recursive algorithm must change its state and move to the base case
- A recursive algorithm must call itself **recursively**

1. A base case is an easy-to-solve case (`numList[0]`)
2. A change of state means some data is modified by the algorithm (`numList gets smaller`)
3. The function solves the problem by calling itself! It feels like a circle. The algorithm breaks down the big problem into smaller and easier problems.

Example: Let's change this question!

We have a list of lists of numbers. We want to get the sum of this nested list. We can have the below program

```
def listSum3(numList):
	theSum = 0
	for aList in numList:
		for num in aList:
			theSum = theSum + num
	return theSum
```

If we have a list of list of numbers. We'll change the algorithm as follows.

```
def listSum4(numList):
	theSum = 0
	for aListList in numList:
		for aList in aListList:
			for num in aList:
				theSum = theSum + num
	return theSum
```

We can simplify our `listSum3` by calling `listSum`, and simplify `listSum4` by calling `listSum3`

```
def listSum3(numList):
	theSum = 0
	for aList in numList:
		theSum = theSum + listSum(aList)
	return theSum

def listSum4(numList):
	theSum = 0
	for aListList in numList:
		theSum = theSum + listSum3(aListList)
	return theSum
```

However if we have a very irregular list….

```
[[1, [2]], [[[3]]], 4, [[5, 6], [[7], 8], [9, 10]]]
```

We need to use recursion to solve the nested list problem

```
def finalSum(numbers):
	"""
	final sum algorithm. Use recursion to solve the nested list problem
	"""
	if isinstance(numbers, int):
		return numbers
	else:
```

```
                theSum = 0
                for aList in numbers:
                        theSum = theSum + finalSum(aList)
        return theSum
```

Think about recursion. One recursive call will cause another recursive call. And that will call another recursive call ….
In order to feel confident about your recursive code, you need to

- Check that the base case is correct (Condition checking, parameters.. etc)
- Check that the recursive step is correct

Exercise:

```
def doDuplication(nestedList):
        """
        return a new nested list with all the numbers in the nestedList duplicated
        each integer should appear twice. The structure should be the same as the input, only
        with
        some new numbers added
        if nestedList is an int, return a list of 2 copies of it
        >>> doDuplication(1)
        [1, 1]
        >>> doDuplication([])
        []
        >>> doDuplication([1, 2])
        [1, 1, 2, 2]
        >>> doDuplication([1, [2, 3]])
        [1, 1, [2, 2, 3, 3]]
        """
        pass

def addOneToList(nestedList):
        """
        add 1 to every number in the nested list
        if nestedList is a int, do nothing
        if it is a list, add 1 to each number
        >>> addOneToList(1)
        1
        >>> addOneToList([])
        []
        >>> addOneToList([1, [2, 3], [[5]]])
        [2, [3, 4], [[6]]]
        """
        pass
```