DSA5101 Introduction to Big Data for Industry

Lecture 3: Functions and Classes

LX Zhang
Department of Mathematics
National University of Singapore

Part II: Functions

- Using basic Python mechanisms, one can write codes for many problems
- Such a code can be long and messy for a complex problem
- Such a code can be hard to keep track of details and correct if wrong
- Such a code can be hard to reuse.

- Well-structured code rely on advanced abstraction and decomposition mechanisms: classes and functions (which are self-defined data types).

- A good piece of code consists of modules, which
  -- are self-contained
  -- can easily be reused
  -- keep code organized and coherent

Modular Structure of A Short Python Program

- Import libraries that are used in your program

```
import pandas as pd
import …
Import …
```

- Define classes for efficiently storing & declare global variable.

```
class Employee:
        …
…
class Salary:
        …
```

- Define functions for data analysis

```
def  function1(…):
        …
…
def  function5(…):
        …
```

- Starting point Computation

```
if __name__ == '__main__':
      function 5(…)
```

For readability, one also needs to annotate your classes and function besides the "readme" file.

Large program is usually divided into several files.

Part II: Functions

- Python empowered with many libraries that contain many functions.

  -- a library contains non-standard data types and functions
  for special computing tasks

- A function is an independent unit of code that performs a special task
- A function takes input data, process them and returns something. It has

  -- a name

  -- arguments or (0 or more) parameters to take input

  -- a body consisting of Python code statements.

  -- (optional) a specification (called a doc-string)

keyword name argument

```
def Fibonacci (k):

""""Input:  k, a positive integer
   output: the k-th Fibonacci number""""
```

docstring

```
        if k==0 or k==1:
              return 1
        else:
              return fibonacci (k-2)+ fibonacci(k-1)
```

body

Example: Count words

- Count the words in a song

```
def Count_Words_Freq(word_list):
        myDict ={}
        for word in word_list:
                if word in myDict:
                        myDict[word] +=1
                else:
                         myDict[word] = 1
         return myDict

Song=['you', 'and', 'me', 'from', 'one', 'world', 'we',  'are',
'family‘, ‘lah’, ‘lah, ‘lah’]

Words_Freq=Count_Words_Freq(Song)
```

- Function is not executed in a program until they are called
- Functions achieve abstraction with function specification

  -- don’t need to see details

  -- hide tedious coding details

  -- don’t need to know how it works to use it, like car.

In mathematics,

-- composition of two simply function

$$f(x) = x^3; g(x) = 2x + 1; (f \circ g)(2) = ?$$

-- use a set of polynomials to approximate a complicate function

In coding,

-- Divide-and-Conquer, dynamic programming

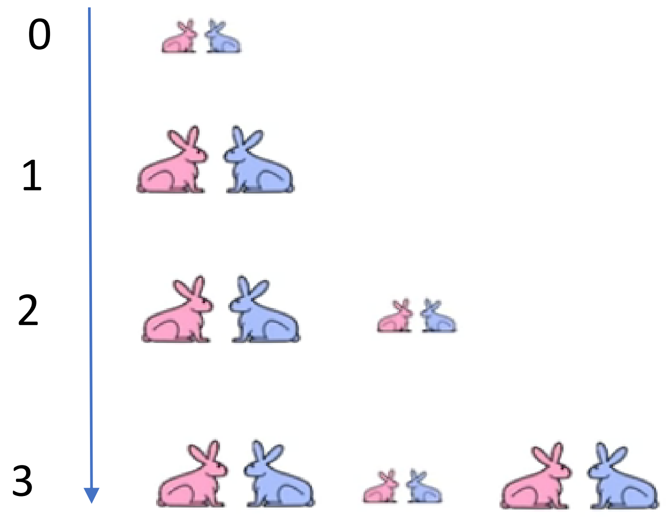
-- Use functions and data types for simplifying computation.

Recursive function and dynamic programming

Problem 1: Compute Fibonacci numbers

- Fibonacci numbers 1, 1, 2, 3, 5, 8, …
- Recursive formula $a[0]=1, a[1]=1, a[k]=a[k-1]+a[k-2], k>1;$

- Leonardo of Pisa (aka Fibonacci) modeled the following challenge
  - Newborn pair of rabbits (one female, one male) are put in a pen
  - Rabbits mate at age of one month
  - Rabbits have a one month gestation period
  - Assume rabbits never die, that female always produces one new pair (one male, one female) every month from its second month on.
  - How many female rabbits are there at the end of one year?

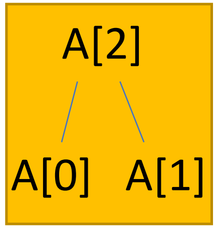


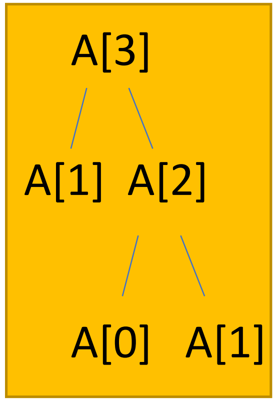
Naïve Implementation

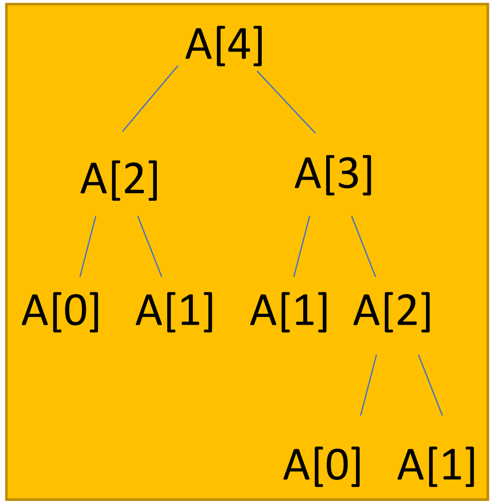
- Fibonacci numbers a[0]=1, a[1]=1, a[k]=a[k-1]+a[k-2], k>1;

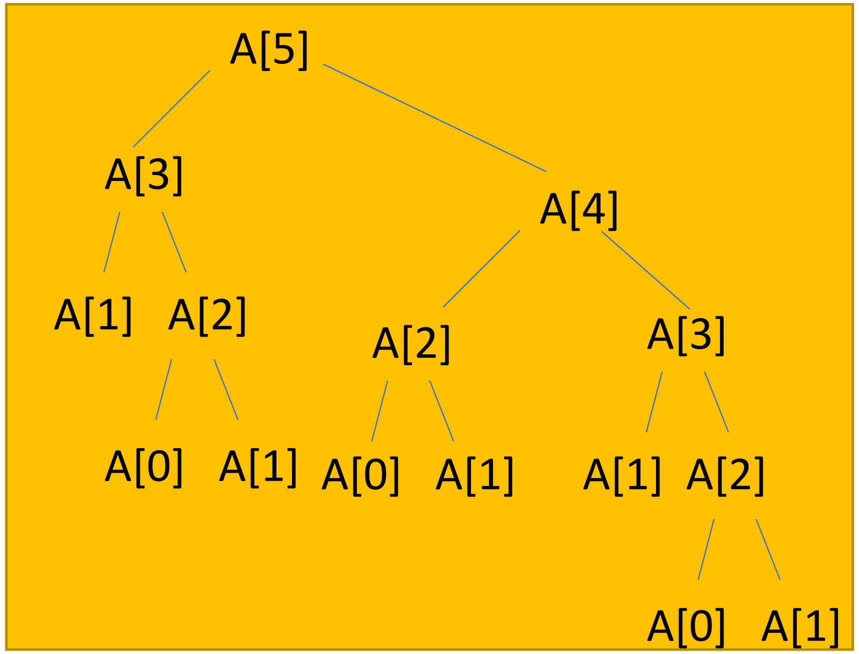
```
def   Fib  ( k ):

        if k==0 or k==1:
              return 1
        else:
              return Fib (k-2)+ Fib (k-1)
```





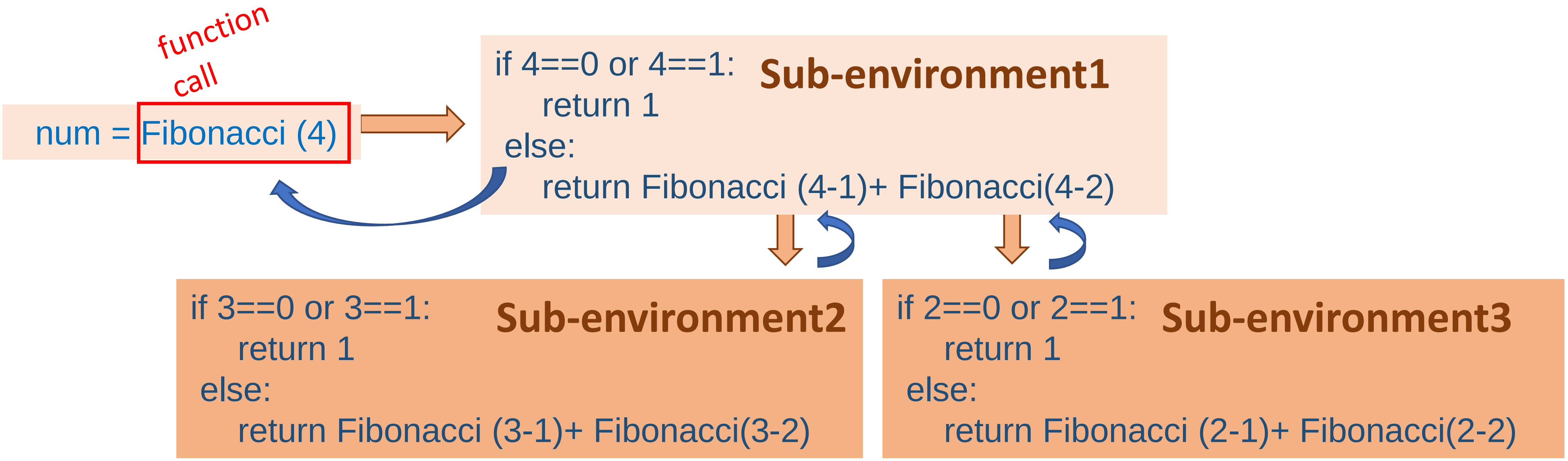




```
def   Fibonacci  ( k ):

      if k==0 or k==1:
            return 1
      else:
            return Fibonacci (k-1)+ Fibonacci(k-2)
```

- Frame/environment is created when a function is called.
- Scope is the mapping of names to objects



2nd Implementation: Memoization

```
#  efficiently compute Fibonacci numbers

def Fib2(k, dict):
        if  k in dict:
              return dict[k]
        else:
              ans= Fib2(k-1,  dict)+ Fib2(k-2, dict)
              dict[k]=ans
              return ans
 dict = {0:1, 1:1}
 Fib2(10, dict)
```

Lookup statement

Update the dictionary

- This tech is called Dynamic Programming, where we use memory to reduce redundant computing.
- Do a lookup first to use the stored values if calculated.
- Update the dictionary as progress through function calls
- Memoization != Memorization

3rd Implementation: Tabular computation

```
#  efficiently compute Fibonacci numbers

def fib_BottomUp(k):
  if k == 1 or k == 0:
      return 1
  table = [0]*(k+1)
  table[0] = 1
  table[1] = 1
  for j in range(2,k+1):
      table[j] = table[j-1] + table[j - 2]
  return table[k]
```

- This is a bottom-up implementation of the dynamic program

Time and space complexity analysis

```
#  efficiently compute Fibonacci numbers

def fib_BottomUp(k):
  if k == 1 or k == 0:
      return 1
  table = [0]*(k+1)
  table[0] = 1
  table[1] = 1
  for j in range(2,k+1):
      table[j] = table[j-1] + table[j - 2]
  return table[k]
```

```
def   Fib  ( k ):

      if k==0 or k==1:
          return 1
      else:
           return Fib (k-2)+ Fib (k-1)
```

- Calculate a value for each of the k cells.
- For each cell, one addition is used.
- In total,   2k (memory reading) + k (addition) operations

Problems that can be solved by DP

- The longest common subsequence problem
- The shortest common supersequence problem
- The string edit distance problem
- The Knapsack problem
- The shortest path problem
- The longest path problem
- …

The Shortest Common Supersequence (SCS) Problem

Input: two sequences P, Q;

Solution: the LCS of both P and Q.

Example: P="ABCDS";

Q="CBSDQ

SCS(P, Q)="ACBCSDSQ"

We will solve the SCS problem using two steps.

- Compute the length of the SCS.
- Infer a SCS from the computation in Step 1

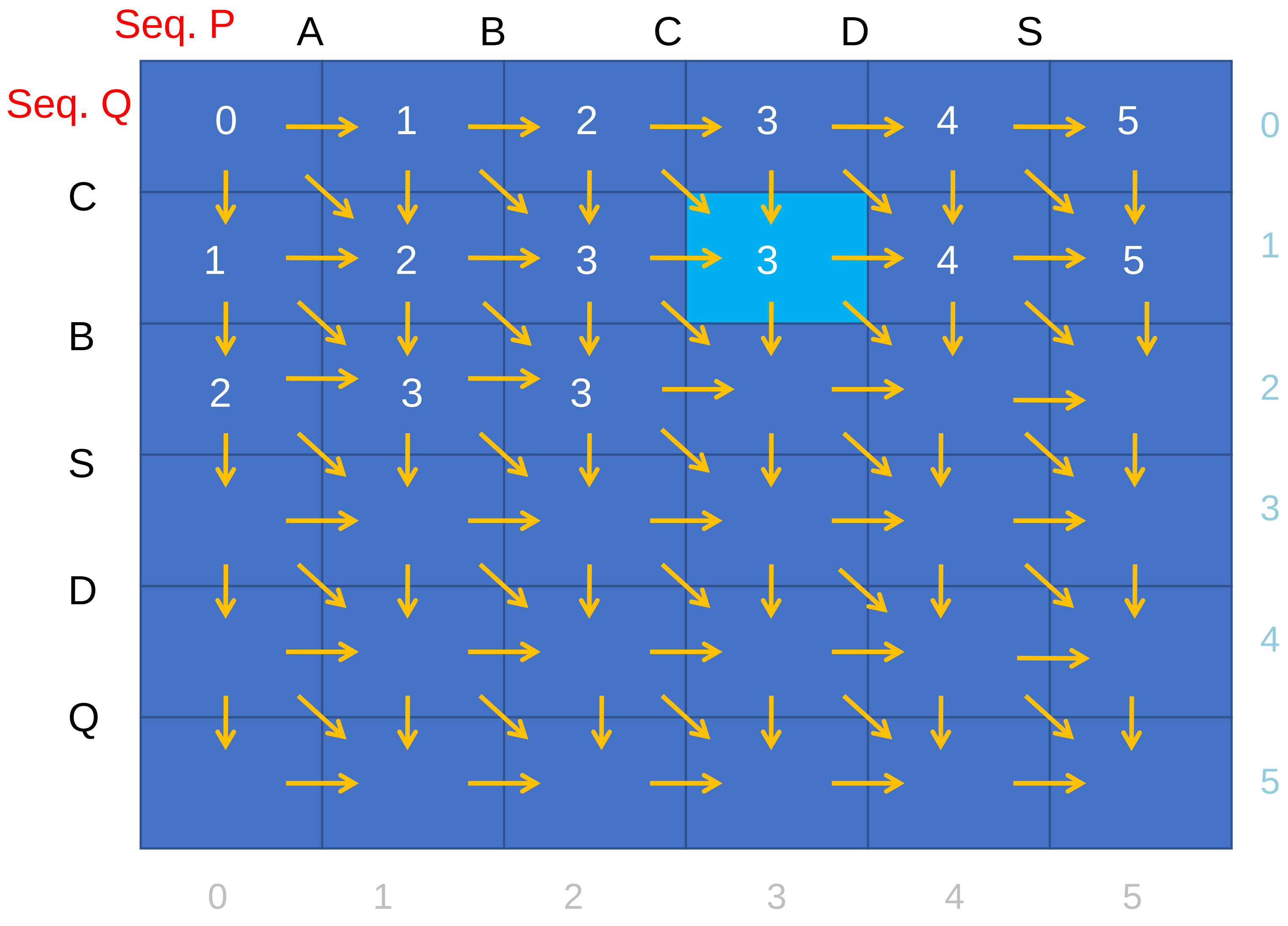
Let $P = x_1 x_2 \cdots x_m$ and $Q = y_1 y_2 \cdots y_n$

Let $S(i, j)$ denote the length of the SCS of the strings

$P_i = x_1 x_2 \cdots x_i$ $(0 \leq i \leq m)$ and $P_j = y_1 y_2 \cdots y_j$ $(0 \leq j \leq n)$

Then, we have the following recursive formula for $S(i, j)$:

$$S(i,j) = \begin{cases} j, & \text{if } i = 0 \\ i, & \text{if } j = 0 \\ S(i-1, j-1) + 1, & \text{if } i > 0, j > 0 \text{ and } x_i = y_j \\ \min\big(S(i-1, j), S(i, j-1)\big) + 1, & \text{if } i > 0, j > 0, \text{and } x_i \neq y_j \end{cases}$$

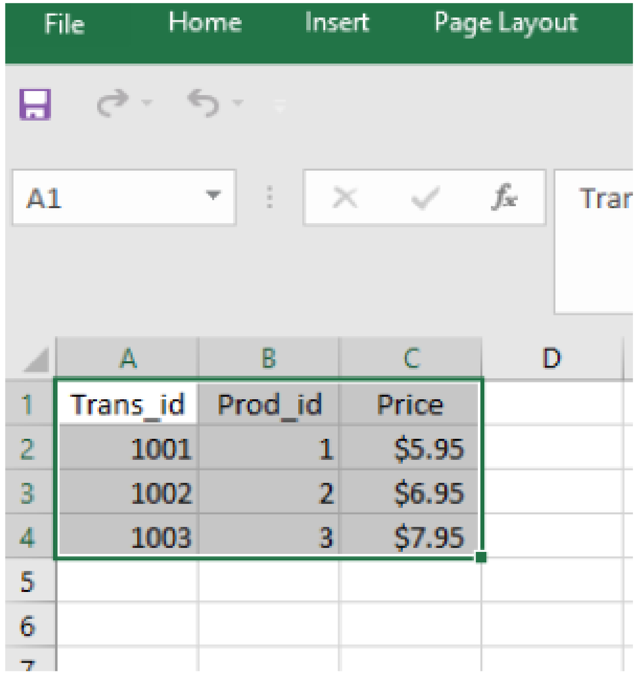


Homework: Code a Python Program on the basis of this tabular Computation.

Read from & write to xlsx Files

```
import openpyxl as xl
from openpyxl.chart import  BarChart, Reference
wb=xl.load_workbook('DSA5101_Exercise.xlsx')
sheet = wb['Sheet1']
```

Read a xlsx file nameted “DSA5101_Exercise.xlsx”



Homework: How to work on image and pdf files in Python?

YuTube Video: https://www.youtube.com/watch?v=7YS6YDQKFh0

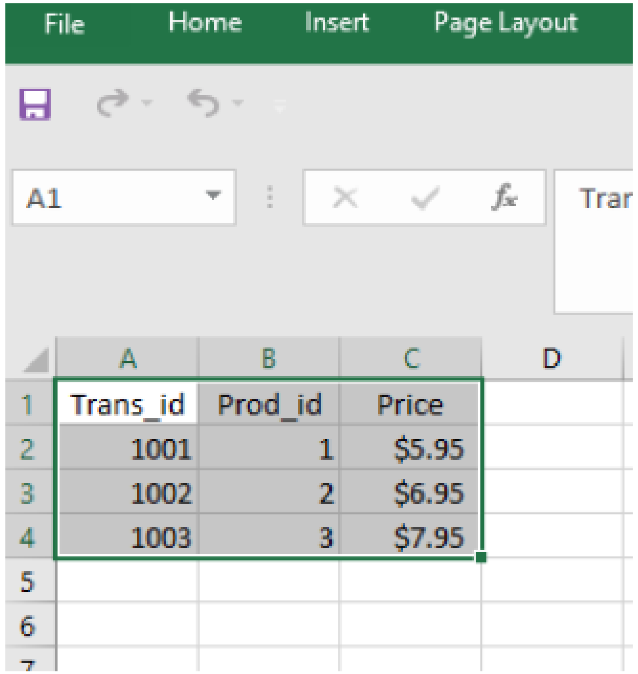
Read from & write to xlsx Files

```
import openpyxl as xl
from openpyxl.chart import  BarChart, Reference
wb=xl.load_workbook('DSA5101_Exercise.xlsx')
sheet = wb['Sheet1']
```

```
for row in range(2, sheet.max_row+1):
    cell=sheet.cell(row,3)
    corrected_price=cell.value * 0.9
    corrected_price_cell=sheet.cell(row, 5)
    corrected_price_cell.value = corrected_price
```

Access every cell in Column C

Write to every cell in Column E



-- sheet.max_row=4

-- the 3rd column is col C.

Read from & write to xlsx Files

```python
import openpyxl as xl
from openpyxl.chart import  BarChart, Reference
wb=xl.load_workbook('DSA5101_Exercise.xlsx')
sheet = wb['Sheet1']
```

```python
for row in range(2, sheet.max_row+1):
    cell=sheet.cell(row,3)
    corrected_price=cell.value * 0.9
    corrected_price_cell=sheet.cell(row, 5)
    corrected_price_cell.value = corrected_price
```

```python
values=Reference(sheet, min_row=2,
                        max_row=sheet.max_row,
                        min_col=5,
                        max_col=5)
```

Access every cell in Col. E

```python
chart = BarChart()
chart.add_data(values)
sheet.add_chart(chart, 'g2')
```

Draw the bar chart at the pos. “g2”

```python
wb.save('test.xlsx')
```