Computational Methods and Modelling

Antonio Attili & Edward McCarthy

antonio.attili@ed.ac.uk
ed.mccarthy@ed.ac.uk

School of Engineering
University of Edinburgh
United Kingdom

Solution tutorial 2

Some basic python exercises. Approximation and errors.

Exercise 1: Sum of the squares of the first 20 odd natural numbers

sqsum_odd.py

```
# Function that returns the sum of
# square of first n natural numbers
def squaresum(n) :
    # Initialise the sum to 0
    sm = 0
    # Iterate i from 1
    # to n finding
    # the square of i and
    # add to sum.
    for i in range(1, n+1) :
        # Check if the current value
        # of i is odd
        if(not (i % 2) == 0):
            sm = sm + (i * i)
    return sm

# Main Program
# Specify n
n = 40
# Call the function squaresum
sum_numbers = squaresum(n)
# Print result on screen
print(sum_numbers)
```

Solution strategy:

- Add an if statement in the for loop to check if the current element of the sum is odd
- In the range 1 to n, there are only $n/2$ odd numbers; change the value of n
- Python files for the solution available on Learn

Exercise 1: Another solution

sqsum_odd_alternative.py

```
# Function that returns the sum of
# square of first n natural numbers
def squaresum_odd_alt(n) :
    # Initialise the sum to 0
    sm = 0
    # Initialise the number counters
    i = 0
    n_odd = 1
    # Iterate with while
    while n_odd <= n:
      i += 1
      if(not (i % 2) == 0):
          sm = sm + (i * i)
          n_odd += 1
    return sm

# Main Program
# Specify n
n = 20
# Call the function squaresum_odd_alt
sum_numbers = squaresum_odd_alt(n)
# Print result on screen
print(sum_numbers)
```

- The previous approach requires to adjust the value of n, which is an algorithmic complication. This is very easy in this case, but it might not be always the case.
- Another approach is to change the approach completely, to obtain a more straightforward and intuitive algorithm.
- In this case we used a while loop instead of a for loop to solve the problem.

Exercise 2: Working with arrays

array.py

```
# import libraries
import numpy as np
import random
# create array of zeros with n elements
n = 20
x = np.zeros(n)
# print array to check it is correct
print(x)
# populate array with random numbers
random_min = 0
random_max = 10
for i in range(0,n):
    x[i] = random.uniform(random_min,
                          random_max)
# print array to check it is correct
print(x)
# find array positions for which
# the value is between a and b
a = 5
b = 6
for i in range(0,n):
    if x[i] > a and x[i] < b:
       print(i)
```

Solution strategy:

- Arrays are created and manipulated with the package numpy
- A specific element in an array is accessed using an index. For example `x[i]` is the element in the `i` position in the array.
- Here, we also used a loop, which starts with the line `for i in range(0,n):` This is used to repeat one or more instructions multiple time.

Exercise 3: Plotting

plot.py

```
# import libraries
import numpy as np
import math
import matplotlib.pyplot as plt

# create array x of coordinate in 0:2pi
n = 20
x = np.linspace(0, 2.0*math.pi, num=n)

# print array to check it is correct
print(x)

# create array y of sine values
y = np.sin(x)

plt.plot(x,y,'.')
plt.xlabel('x')
plt.ylabel('y=sin(x)')

plt.show()
```

Solution strategy:

- We define the array `x` which contains `n`=20 equispaced coordinates in the desired range.
- Then we create the array `y`, which contains the sine of each element of `x`. Note that we used the numpy function `np.sin()` because we need a sin function that can operate on arrays. The function `math.sin()` does NOT work here.
- Finally we plot the two arrays `x` and `y` against each other and make the plot look more professional by adding labels.

Exercise 4: Error in a series approximation (exam-type question)

series.py

```
# importing modules
import numpy as np
import matplotlib.pyplot as plt
import math
N = 100
s = 0
pi_n = np.zeros(N)
nn = np.zeros(N)
error_true = np.zeros(N)
error_ext = np.zeros(N)
for i in range(1,N+1):
    pi_old = (s*6.0)**0.5
    s = s + 1.0/i**2.0
    pi_n[i-1] = (s*6.0)**0.5
    nn[i-1] = i
    error_true[i-1] =
    np.absolute(pi_n[i-1] - np.pi)
    error_ext[i-1]  =
    np.absolute(pi_n[i-1] - pi_old)
    print(i,pi_n[i-1],error_true[i-1],
    error_ext[i-1])
plt.figure()
plt.loglog(nn,error_true,'-b',
nn,error_ext,'.r')
plt.show()
```

Solution strategy:

- First we create some arrays to store the solution and the errors (`pi_n`, `error_true` etc) using the `numpy` function `np.zeros()`
- We create a `for` loop to add the terms of the series.
- The errors are computed at each iteration of the loop for convenience.
- In this case, the approximate error is computed as the absolute value of the difference of two consecutive iteration.
- Since we stored the errors in array, we can also plot them at the end.