02 Exercises

September 13, 2020

0.1 Exercise 02.1 (if-else)

Consider the following assessment criteria which map a score out of 100 to an assessment grade:

Grade	Raw score (/100)
Excellent	≥ 82
Very good	≥ 76.5 and < 82
Good	≥ 66 and < 76.5
Need improvement	≥ 45 and < 66
Did you try?	< 45

Write a program that, given an a score, prints the appropriate grade. Print an error message if the input score is greater than 100 or less than zero.

[0]:
```
x=54.99
if x>=82 and x<=100:
  print('Excellent')
elif x>=76.5 and x<=82:
  print('Very Good')
elif x>=66 and x<=76.5:
  print('Good')
elif x>=45 and x<=66:
  print('Need improvement')
elif x>=0 and x<=45:
  print('Did you try?')
elif x>100 or x<0:
  print('the score out of the range')

#raise NotImplementedError()
```

```
Need improvement
```

0.2 Exercise 02.2 (bisection)

Bisection is an iterative method for finding approximate roots of a function. Say we know that the function $f(x)$ has one root between x_0 and x_1 ($x_0 < x_1$). We then:

- Evaluate f at the midpoint $x_{\text{mid}} = (x_0 + x_1)/2$, i.e. compute $f_{\text{mid}} = f(x_{\text{mid}})$

- Evaluate $f(x_0) \cdot f(x_{\text{mid}})$
  - If $f(x_0) \cdot f(x_{\text{mid}}) < 0$:

    f must change sign somewhere between x_0 and x_{mid}, hence the root must lie between x_0 and x_{mid}, so set $x_1 = x_{\text{mid}}$.
  - Else

    f must change sign somewhere between x_{mid} and x_1, so set $x_0 = x_{\text{mid}}$.

The above steps can be repeated a specified number of times, or until $|f_{\text{mid}}|$ is below a tolerance, with x_{mid} being the approximate root.

0.2.1 Task

The function

$$f(x) = x^3 - 6x^2 + 4x + 12$$

has one root somewhere between $x_0 = 3$ and $x_1 = 6$.

1. Use the bisection method to find an approximate root x_r using 15 iterations (use a `for` loop).
2. Use the bisection method to find an approximate root x_r such that $|f(x_r)| < 1 \times 10^{-6}$ and report the number of iterations required (use a `while` loop).

Store the approximate root using the variable `x_mid`, and store $f(x_{\text{mid}})$ using the variable `f`.

Hint: Use `abs` to compute the absolute value of a number, e.g. `y = abs(x)` assigns the absolute value of `x` to `y`.

(1) Using a `for` loop.

[0]:
```
# Initial end points
x0 = 3.0
x1 = 6.0

# Use 15 iterations
for n in range(15):
    # Compute midpoint
    x_mid = (x0 + x1)/2

    # Evaluate function at left end-point and at midpoint
    f0 = x0**3 - 6*x0**2 + 4*x0 + 12
    f = x_mid**3 - 6*x_mid**2 + 4*x_mid + 12

    # YOUR CODE HERE
    if f0*f<0:
      x1=x_mid
    else:
      x0=x_mid
    #raise NotImplementedError()
```

```python
    print(n, x_mid, f)
```

```
0 4.5 -0.375
1 5.25 12.328125
2 4.875 4.763671875
3 4.6875 1.910888671875
4 4.59375 0.699554443359375
5 4.546875 0.14548873901367188
6 4.5234375 -0.11891412734985352
7 4.53515625 0.01224285364151001
8 4.529296875 -0.053596146404743195
9 4.5322265625 -0.020741849206387997
10 4.53369140625 -0.0042658079182729125
11 4.534423828125 0.003984444148954935
12 4.5340576171875 -0.0001417014154867502
13 4.53424072265625 0.0019211164656098845
14 4.534149169921875 0.0008896438020826736
```

[0]:
```python
assert round(x_mid - 4.534149169921875, 10) == 0.0
assert abs(f) < 0.0009
```

(2) Using a `while` loop Use the variable `counter` for the iteration number.

Remember to guard against infinite loops.

[0]:
```python
# Initial end points
x0 = 3.0
x1 = 6.0

tol = 1.0e-6
error = tol + 1.0

# Iterate until tolerance is met
counter = 0
while error > tol:
    counter += 1
    # Compute midpoint
    x_mid = (x0 + x1)/2
    # Evaluate function at left end-point and at midpoint
    f0 = x0**3 - 6*x0**2 + 4*x0 + 12
    f = x_mid**3 - 6*x_mid**2 + 4*x_mid + 12
    # Condition:
    if f0*f<0:
      x1=x_mid
    else:
      x0=x_mid
    if abs(f)< 10**(-6):
```

```
      break
    #raise NotImplementedError()
    # Guard against an infinite loop
    if counter > 1000:
      print("Oops, iteration count is very large. Breaking out of while loop.")
      break
    print(counter, x_mid, error, f)
```

```
1 4.5 1.000001 -0.375
2 5.25 1.000001 12.328125
3 4.875 1.000001 4.763671875
4 4.6875 1.000001 1.910888671875
5 4.59375 1.000001 0.699554443359375
6 4.546875 1.000001 0.14548873901367188
7 4.5234375 1.000001 -0.11891412734985352
8 4.53515625 1.000001 0.01224285364151001
9 4.529296875 1.000001 -0.053596146404743195
10 4.5322265625 1.000001 -0.020741849206387997
11 4.53369140625 1.000001 -0.0042658079182729125
12 4.534423828125 1.000001 0.003984444148954935
13 4.5340576171875 1.000001 -0.0001417014154867502
14 4.53424072265625 1.000001 0.0019211164656098845
15 4.534149169921875 1.000001 0.0008896438020826736
16 4.5341033935546875 1.000001 0.0003739552628445608
17 4.534080505371094 1.000001 0.0001161229410939768
18 4.534069061279297 1.000001 -1.2790232830184323e-05
19 4.534074783325195 1.000001 5.166610522167048e-05
20 4.534071922302246 1.000001 1.9437873959304852e-05
21 4.5340704917907715 1.000001 3.3238050036743516e-06
22 4.534069776535034 1.000001 -4.7332178070291775e-06
```

[0]:
```
assert counter == 23
assert abs(f) < 1.0e-6
```

0.3 Exercise 02.3 (series expansion)

The power series expansion for the sine function is:

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

(See mathematics data book for a less compact version; this compact version is preferred here as it is simpler to program.)

1. Using a `for` statement, approximate $\sin(3\pi/2)$ using 15 terms in the series expansion and report the absolute error.

2. Using a `while` statement, compute how many terms in the series are required to approximate $\sin(3\pi/2)$ to within 1×10^{-8}.

Store the absolute value of the error in the variable `error`.

Note: Calculators and computers use iterative or series expansions to compute trigonometric functions, similar to the one above (although they use more efficient formulations than the above series).

0.3.1 Hints

To compute the factorial and to get a good approximation of π, use the Python `math` module:

```
import math
nfact = math.factorial(10)
pi = math.pi
```

You only need '`import math`' once at the top of your program. Standard modules, like `math`, will be explained in a later. If you want to test for angles for which sine is not simple, you can use

```
a = 1.3
s = math.sin(a)
```

to get an accurate computation of sine to check the error.

(1) Using a `for` loop

[0]:
```
# Import the math module to access math.sin and math.factorial
import math
# Value at which to approximate sine
x = 1.5*math.pi
# Initialise approximation of sine
approx_sin = 0.0
for n in range(16):
  approx_sin +=((-1)**n)*(x**((2*n)+1))/math.factorial((2*n)+1)
  error = abs((math.sin(x)-approx_sin))
  print(error, approx_sin, math.sin(x))
#raise NotImplementedError()
```

```
5.71238898038469 4.71238898038469 -1.0
11.728641652283956 -12.728641652283956 -1.0
7.636666525534631 6.636666525534631 -1.0
2.602329768407337 -3.602329768407337 -1.0
0.555634071762265 -0.444365928237735 -1.0
0.08189021082585013 -1.0818902108258501 -1.0
0.008861411268856534 -0.9911385887311435 -1.0
0.0007351881114854297 -1.0007351881114854 -1.0
4.829695774011267e-05 -0.9999517030422599 -1.0
2.5759875719177927e-06 -1.000002575987572 -1.0
1.1381159747969605e-07 -0.9999998861884025 -1.0
4.234491202126378e-09 -1.0000000042344912 -1.0
1.345145106412815e-10 -0.9999999998654855 -1.0
3.6917136014835705e-12 -1.0000000000036917 -1.0
```

```
8.79296635503124e-14 -0.9999999999999121 -1.0
2.220446049250313e-15 -1.0000000000000022 -1.0
```

[0]:

```
assert error < 1.0e-12
```

(2) Using a while loop *Remember to guard against infinite loops.*

[0]:

```
# Import the math module to access math.sin and math.factorial
import math

# Value at which to approximate sine
x = 1.5*math.pi

# Tolerance and initial error (this just needs to be larger than tol)
tol = 1.0e-8
error = tol + 1.0
# Intialise approximation of sine
approx_sin = 0.0
# Initialise counter
n = 0
# Loop until error satisfies tolerance, with a check to avoid
# an infinite loop
while error > tol and n < 1000:
  # compute how many terms in the series are required to approximate sin(3 /2)␣
↪to within 1×10−8
  approx_sin +=((-1)**n)*(x**((2*n)+1))/math.factorial((2*n)+1)
  error = abs((math.sin(x)-approx_sin))
  # raise NotImplementedError()
  # Increment counter
  n += 1
  if error < 1.0e-8:
    break
  print("The approx_sin is:", approx_sin)
  print("The error is:", error)
  print("Number of terms in series:", n)
```

```
The approx_sin is: 4.71238898038469
The error is: 5.71238898038469
Number of terms in series: 1
The approx_sin is: -12.728641652283956
The error is: 11.728641652283956
Number of terms in series: 2
The approx_sin is: 6.636666525534631
The error is: 7.636666525534631
Number of terms in series: 3
The approx_sin is: -3.602329768407337
The error is: 2.602329768407337
```

```
Number of terms in series: 4
The approx_sin is: -0.444365928237735
The error is: 0.555634071762265
Number of terms in series: 5
The approx_sin is: -1.0818902108258501
The error is: 0.08189021082585013
Number of terms in series: 6
The approx_sin is: -0.9911385887311435
The error is: 0.008861411268856534
Number of terms in series: 7
The approx_sin is: -1.0007351881114854
The error is: 0.0007351881114854297
Number of terms in series: 8
The approx_sin is: -0.9999517030422599
The error is: 4.829695774011267e-05
Number of terms in series: 9
The approx_sin is: -1.000002575987572
The error is: 2.5759875719177927e-06
Number of terms in series: 10
The approx_sin is: -0.9999998861884025
The error is: 1.1381159747969605e-07
Number of terms in series: 11
```

[0]:
```
assert error <= 1.0e-8
```