04 Exercises

September 13, 2020

0.1 Exercise 04.1 (simple function)

Write a function called `is_odd` which takes an integer as an argument and returns `True` if the argument is odd, and otherwise returns `False`. Test your function for several values.

[0]:
```
def is_odd(x):
    # YOUR CODE HERE
    if x%2 == 0:
      return False
    else:
      return True
    #raise NotImplementedError()
```

[0]:
```
assert is_odd(0) == False
assert is_odd(101) == True
assert is_odd(982) == False
assert is_odd(-5) == True
assert is_odd(-8) == False
```

0.2 Exercise 04.2 (functions and default arguments)

Write a single function named `magnitude` that takes each component of a vector of length 2 or 3 and returns the magnitude. Use default arguments to handle vectors of length 2 or 3 with the same code. Test your function for correctness against hand calculations for a selection of values.

[0]:
```
import math

# YOUR CODE HERE
def magnitude(a,b,c=0):
  y=math.sqrt(a**2+b**2+c**2)
  return y

#raise NotImplementedError()
```

[0]:
```
assert round(magnitude(3, 4) - 5.0, 10) == 0.0
assert round(magnitude(4, 3) - 5.0, 10) == 0.0
assert round(magnitude(4, 3, 0.0)- 5.0, 10) == 0.0
assert round(magnitude(4, 0.0, 3.0) - 5.0, 10) == 0.0
assert round(magnitude(3, 4, 4) - 6.403124237, 8) == 0.0
```

0.3 Exercise 04.3 (functions)

Given the coordinates of the vertices of a triangle, (x_0, y_0), (x_1, y_1) and (x_2, y_2), the area A of the triangle is given by:

$$A = \left| \frac{x_0(y_1 - y_2) + x_1(y_2 - y_0) + x_2(y_0 - y_1)}{2} \right|$$

Write a function named `area` that computes the area of a triangle given the coordinates of the vertices. The order of the function arguments must be `(x0, y0, x1, y1, x2, y2)`.

Test the output of your function against some known solutions.

[0]:
```
def area(x0, y0, x1, y1, x2, y2):
  A = abs((x0*(y1-y2))+(x1*(y2-y0))+(x2*(y0-y1)))/2
  return A
```

[0]:
```
x0, y0 = 0.0, 0.0
x1, y1 = 0.0, 2.0
x2, y2 = 3.0, 0.0
A = area(x0, y0, x1, y1, x2, y2)
assert round(A - 3.0, 10) == 0.0
```

0.4 Exercise 04.4 (recursion)

The factorial of a non-negative integer n is expressed recursively by:

$$n! = \begin{cases} 1 & n = 0 \\ (n-1)!\, n & n > 0 \end{cases}$$

Develop a function named `factorial` for computing the factorial using recursion. Test your function against the `math.factorial` function, e.g.

[0]:
```
import math
print("Reference factorial:", math.factorial(5))
```

```
Reference factorial: 120
```

[0]:
```
def factorial(n):
  if n==0:
    return 1
  elif n>0:
    return factorial(n-1)*n
#raise NotImplementedError()

print("Factorial of 5:", factorial(5))

import math
print("Reference value of factorial of 5:", math.factorial(5))
```

```
Factorial of 5: 120
Reference value of factorial of 5: 120
```

[0]:
```
assert factorial(0) == 1
assert factorial(1) == 1
assert factorial(2) == 2
assert factorial(5) == 120

import math
assert factorial(32) == math.factorial(32)
```

0.5 Exercise 04.5 (functions and passing functions as arguments)

Restructure your program from the bisection problem in Exercise 02 to

- Use a Python function to evaluate the mathematical function f that we want to find the root of;

and then

- Encapsulate the bisection algorithm inside a Python function, which takes as arguments:
  - the function we want to find the roots of
  - the points x_0 and x_1 between which we want to search for a root
  - the tolerance for exiting the bisection algorithm (exit when $|f(x)| < \text{tol}$)
  - maximum number of iterations (the algorithm should exit once this limit is reached)

For the first step, use a function for evaluating f, e.g.:

```
def f(x):
    # Put body of the function f(x) here, and return the function value
```

For the second step, encapsulate the bisection algorithm in a function:

```
def compute_root(f, x0, x1, tol, max_it):
    # Implement bisection algorithm here, and return when tolerance is satisfied or
    # number of iterations exceeds max_it

    # Return the approximate root, value of f(x) and the number of iterations
    return x, f, num_it

# Compute approximate root of the function f
x, f_x, num_it = compute_root(f, x0=3, x1=6, tol=1.0e-6, max_it=1000)
```

Try testing your program for a different function. A quadratic function, whose roots you can find analytically, would be a good test case.

0.5.1 Optional extension

Use recursion to write a `compute_root` function that *does not* require a `for` or `while` loop.

0.5.2 Solution

Define the function for computing $f(x)$:

[0]:
```
def my_f(x):
    "Evaluate polynomial function"
    return x**3 - 6*x**2 + 4*x + 12
```

Create the function that performs the bisection:

[0]:
```
def compute_root(f, x0, x1, tol, max_it):
    "Compute roots of a function using bisection"
    # Implement bisection algorithm here, and return when tolerance is␣
 ↪satisfied or
    # Initial end points
    error = tol + 1.0
    # Iterate until tolerance is met
    it = 0
    while error > tol:
        it += 1
        # Compute midpoint
        x_mid = (x0 + x1)/2
        f = my_f(x0)
        f_mid = my_f(x_mid)
        # Condition:
        if f*f_mid < 0:
            x1=x_mid
        else:
            x0=x_mid
        if abs(f_mid)< tol:
            break
        # number of iterations exceeds max_it
    if it > max_it:
        print("Oops, iteration count is very large. Breaking out of while loop.
 ↪")
    # Return the approximate root, value of f(x) and the number of iterations  ␣
 ↪
    return x_mid, f_mid, it

x, f_x, num_it = compute_root(my_f, x0=3, x1=6, tol=1.0e-6, max_it=1000)
print(x, f_x, num_it)
```

```
4.534070134162903 -7.047073751209609e-07 23
```

[0]:
```
x, f, num_it = compute_root(my_f, x0=3, x1=6, tol=1.0e-6, max_it=1000)
assert round(x - 4.534070134162903, 10) == 0.0
```

Optional extension: using recursion:

[7]:
```python
def compute_root(f, x0, x1, tol, max_it, it_count=0):
    "Compute roots of a function using bisection"
    it_count += 1
    # Compute midpoint
    x_mid = (x0 + x1)/2
    f = my_f(x0)
    f_mid = my_f(x_mid)
    if abs(f_mid) > tol:
        # Condition:
        if f*f_mid < 0:
            x1=x_mid
        else:
            x0=x_mid
        return compute_root(f, x0, x1, tol, max_it, it_count)
    elif it_count > max_it:
        print("Oops, iteration count is very large. Breaking out of while loop.
 ↪")
    else:
        return x_mid, f_mid, it_count
x, f, num_it = compute_root(my_f, x0=3, x1=6, tol=1.0e-6, max_it=1000)
print(x, f, num_it)
```

```
4.534070134162903 -7.047073751209609e-07 23
```

[0]:
```python
x, f, num_it = compute_root(my_f, x0=3, x1=6, tol=1.0e-6, max_it=1000)
assert round(x - 4.534070134162903, 10) == 0.0
```