worksheet3-corrected

October 6, 2016

1 Course 3 - functions (corrected version)

Some comments about the assignment - This work is made for you: it is a revision of all previous worksheets. Moreover, the content of the following courses will be adapted depending on the results. For these reasons, the assignment must be personal. - Each exercise should be solved using the cells below it. If needed you can add more cells with the menu "Insert -> Insert Cell Above" and "Insert -> Insert Cell Below". - Your code should be clear and execute cleanly from top to bottom (you can check with the menu "Kernel -> Restart & Run All") - It is better to split your code into several cells rather than having a big cell with a lot of code - Do not modify the statements of the exercises. If you want to add comments to your answers, either use Python comment (with #) or create new cell of Markdown type. - Each function you program should come with examples.

2 Functions

In the previous worksheets you have already used a lot of Python functions like `print`, `len`, `math.cos`, `numpy.arange` or `matplotlib.pyplot.plot` (do you have others in mind?). Functions are useful because they allow you to write code that you can reuse at several places in a computation. In this worksheet we will learn how to write them.

We present the syntax by writing a function which takes in argument two numbers `x` and `y` and returns the value of `cos(x) + sin(y)`.

```
def f(x, y):
    from math import cos, sin
    s = cos(x)
    t = sin(y)
    return s + t
```

The keyword `return` specifies the value to be returned by the function. It can be any Python object.

Exercise: - Copy the function `f` above in a code cell and test it with various input values. - Program the function $g : z \mapsto exp(z^2 + z + 1)$. - What is the value of `f(g(1.0), 2.3)`?

```
In [1]: def f(x, y):
            from math import cos, sin
            s = cos(x)
            t = sin(y)
            return s + t
```

In [2]:
```
f(0, 1)
```

Out[2]:
```
1.8414709848078965
```

In [3]:
```
f(-0.3, 4.13)
```

Out[3]:
```
0.12018544334051251
```

In [ ]:

In [4]:
```
def g(z):
    from math import exp
    return exp(z*z + z + 1)
```

In [5]:
```
g(3)
```

Out[5]:
```
442413.3920089205
```

In [ ]:

In [6]:
```
f(g(1.0), 2.3)
```

Out[6]:
```
1.0742999676092524
```

In [ ]:

Exercise: Make a function `sign(x)` that returns the sign of a number `x` (i.e. `-1` if `x` is negative, `0` if `x` is zero and `1` if `x` is positive).

In [7]:
```
def sign(x):
    if x > 0:
        return 1
    elif x < 0:
        return -1
    else:
        return 0
```

In [8]:
```
[sign(a) for a in range(-5,5)]
```

Out[8]:
```
[-1, -1, -1, -1, -1, 0, 1, 1, 1, 1]
```

In [ ]:

Exercise: Let us consider the fibonacci sequence defined as

$$F_0 = F_1 = 1 \qquad \text{and} \qquad F_{n+2} = F_{n+1} + F_n$$

. - Write a function `fib_range(n)` that returns the list of the first n Fibonacci numbers. - Write a function `fib(n)` that returns the n-th Fibonacci number (this function must not use a list). - Check the following formulas up to $n = 100$:

$$F_{2n} = F_{n-1}^2 + F_n^2 \qquad F_{2n+1} = F_{n+1}^2 - F_{n-1}^2$$

```
In [153]: def fib_range(n):
              if n <= 0:
                  return []
              elif n == 1:
                  return [1]
              else:
                  l = [1,1]
                  for i in range(n-2):
                      l.append(l[-1] + l[-2])
                  return l
```

```
In [154]: fib_range(10)
```

```
Out[154]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

```
In [155]: [len(fib_range(i)) for i in range(10)]
```

```
Out[155]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [ ]:
```

```
In [19]: def fib(n):
             if n < 0:
                 raise ValueError("negative n not allowed")
             elif n < 2:
                 return 1
             else:
                 f0 = f1 = 1
                 for i in range(n-1):
                     z = f1
                     f1 = f0 + f1
                     f0 = z
                 return f1
```

```
In [23]: fib(0)
```

```
Out[23]: 1
```

```
In [24]: fib(5)
```

```
Out[24]: 8
```

```
In [22]: [fib(i) for i in range(10)] == fib_range(10)
```

```
Out[22]: True
```

```
In [ ]:
```

```
In [183]: F = fib_range(200)
```

```
In [189]: # checking the first formula
          all([F[2*n] == F[n-1]**2 + F[n]**2 for n in range(1,100)])
```

```
Out[189]: True
```

```
In [190]: # checking the second formula
          all([F[2*n+1] == F[n+1]**2 - F[n-1]**2 for n in range(1,100)])
```

```
Out[190]: True
```

```
In [ ]:
```

Exercise: - Write a function `mean(l)` that returns the mean value of a list `l` made of floating point numbers. - Write a function `var(l)` that returns the variance of `l`. - Compute the mean and variance of the following list of numbers

```
[1.34, 12.25, 5.5, 3.26, 7.22, 1.7, 11.12, 3.33, 10.23]
```

```
In [25]: def mean(l):
             return sum(l) / len(l)

         def var(l):
             m = mean(l)
             v = 0.0
             for x in l:
                 v = v + (x-m)**2
             return v / len(l)
```

```
In [28]: mean([1,1])
```

```
Out[28]: 1.0
```

```
In [29]: mean([1,2,3])
```

```
Out[29]: 2.0
```

```
In [ ]:
```

```
In [30]: var([1,1])
```

```
Out[30]: 0.0
```

```
In [31]: var([1,2,3])
```

```
Out[31]: 0.6666666666666666
```

```
In [ ]:
```

```
In [32]: mean([1.34, 12.25, 5.5, 3.26, 7.22, 1.7, 11.12, 3.33, 10.23])
```

```
Out[32]: 6.216666666666667
```

In [33]: var([1.34, 12.25, 5.5, 3.26, 7.22, 1.7, 11.12, 3.33, 10.23])

Out[33]: 15.480866666666666

In [ ]:

exercise: The Collatz function is the function defined on the positive integers as $collatz(n) = n/2$ if n is even and $collatz(n) = 3n+1$ otherwise. It is a conjecture that starting from any positive integer and repeatedly applying the collatz function we end up with the cycle $1 \mapsto 4 \mapsto 2 \mapsto 1$. For example

$$3 \mapsto 10 \mapsto 5 \mapsto 16 \mapsto 8 \mapsto 4 \mapsto 2 \mapsto 1.$$

- Program the function `collatz(n)`.
- Write a function `collatz_length(n)` that returns the number of steps needed to reach 1 by applying the Collatz function starting from n.
- Compute the lengths needed to attain 1 with the Collatz function for the first 50th integers.
- Find the largest of these lengths.

In [34]:
```
def collatz(n):
    if n%2 == 0:
        return n // 2
    else:
        return 3*n + 1
```

In [36]:
```
print(collatz(3), collatz(4), collatz(1))
```

```
10 2 4
```

In [ ]:

In [37]:
```
def collatz_length(n):
    count = 0
    while n != 1:
        n = collatz(n)
        count = count + 1
    return count
```

In [38]:
```
collatz_length(3)
```

Out[38]: 7

In [ ]:

In [40]:
```
print([collatz_length(n) for n in range(1,51)])
```

```
[0, 1, 7, 2, 5, 8, 16, 3, 19, 6, 14, 9, 9, 17, 17, 4, 12, 20, 20, 7, 7, 15, 15, 10,
```

In [ ]:

In [ ]:

Exercise:

- What does the following function do?

```
def function(n):
    d = []
    for i in range(1, n+1):
        if n % i == 0:
            d.append(i)
    return d
```

(*hint: you might want to test this function with small positive integers* `n` *as input*) - Can you find a more efficient way to program it?

```
In [41]: def function(n):
             "Returns the divisors of n"
             d = []
             for i in range(1, n+1):
                 if n % i == 0:
                     d.append(i)
             return d
```

```
In [42]: for n in range(1,20):
             print(n, function(n))
```

```
1 [1]
2 [1, 2]
3 [1, 3]
4 [1, 2, 4]
5 [1, 5]
6 [1, 2, 3, 6]
7 [1, 7]
8 [1, 2, 4, 8]
9 [1, 3, 9]
10 [1, 2, 5, 10]
11 [1, 11]
12 [1, 2, 3, 4, 6, 12]
13 [1, 13]
14 [1, 2, 7, 14]
15 [1, 3, 5, 15]
16 [1, 2, 4, 8, 16]
17 [1, 17]
18 [1, 2, 3, 6, 9, 18]
19 [1, 19]
```

In [ ]:

Exercise: Given a smooth function $f : \mathbb{R} \to \mathbb{R}$, the Newton method consists in starting from an initial value $x_0 \in \mathbb{R}$ and forming the sequence

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Under certain general conditions, one can show that x_n converges to a root of f (if you are curious you can have a look at the corresponding wikipedia article). - Write a function `nth_root(s, n, x0, k)` that returns the term x_k of the Newton sequence obtained for the function $f(z) = z^n - s$ and where `x0` is the initial condition. - Try this function with $s = 2$, $n = 2$, $x_0 = 1$, $k = 5$. Which equation does satisfy the output? - Try more examples and check whether the returned value satisfy the equation $f(z) = 0$.

In [43]:
```
def nth_roots(s, n, x0, k):
    x = x0
    for i in range(k):
        x = ((n-1)*x + s / x**(n-1)) / n
    return x
```

In [46]:
```
sqrt2 = nth_roots(2, 2, 1.0, 5)
print(sqrt2, sqrt2**2)
```

```
1.414213562373095 1.9999999999999996
```

In [ ]:

In [47]:
```
cbrt2 = nth_roots(2, 3, 1.0, 5)
print(cbrt2, cbrt2**3)
```

```
1.2599210498948732 2.0
```

In [ ]:

In [56]:
```
# studying the speed of convergence
for n in range(5,18):
    a = nth_roots(5, 12, 3.0, n)
    print(a, 5 - a**12)
```

```
1.9418495482740594 -2869.663007580449
1.780310213046638 -1008.7904574556771
1.6326827339964083 -353.77452663035245
1.4985219732018729 -123.22047634104769
1.3785147555466084 -42.09090420998932
1.275835810250663 -13.601049716410468
1.198095097809907 -3.74773250256351
1.1553207756459598 -0.6549790165403673
1.1441696752438786 -0.033675302637647064
1.1435318003683963 -0.00010306529686410215
```

```
1.1435298361014778 -9.736975670193715e-10
1.1435298360829202 3.552713678800501e-15
1.1435298360829205 -7.993605777301127e-15
```

```
In [ ]:
```

When calling a function you can also name the arguments. For example defining

```
def f(x, y):
    return x + 2*y
```

You can use any of the following equivalent form

```
f(1, 2)
f(x=1, y=2)
f(y=2, x=1)
```

Exercise: Copy, paste and execute the above code.

```
In [57]: def f(x, y):
             return x + 2*y

In [58]: f(1, 2)

Out[58]: 5

In [59]: f(x=1, y=2)

Out[59]: 5

In [60]: f(y=2, x=1)

Out[60]: 5

In [ ]:
```

It is also possible to set some of the arguments of a function to some default. For example

```
def newton_sqrt2(num_steps=5):
    x = 1.0
    for i in range(num_steps):
        x = (x + 2.0 / x) / 2.0
    return x
```

Defining the function as above, the argument `num_steps` is optional. You can hence use this function in any of the following form

```
newton_sqrt2()
newton_sqrt(12)
newton_sqrt(num_steps=12)
```

Exercise: - Find out what the above code is doing. - We have already seen some functions with optional arguments. Do you remember some?

```
In [66]: def newton_sqrt2(num_steps=5):
             "Returns the square root of 2 using the Newton method"
             x = 1.0
             for i in range(num_steps):
                 x = (x + 2.0 / x) / 2.0
             return x
```

```
In [62]: newton_sqrt2()
```

```
Out[62]: 1.414213562373095
```

```
In [64]: newton_sqrt2(12)
```

```
Out[64]: 1.414213562373095
```

```
In [65]: newton_sqrt2(num_steps=12)
```

```
Out[65]: 1.414213562373095
```

```
In [ ]:
```

Exercise: - Copy the function `nth_root`, change its name to `nth_root_bis` where: - the argument `x0` is optional with default `1.0` - the argument `k` is optional with default `10`

```
In [71]: def nth_root_bis(s, n, x0=1.0, k=10):
             x = x0
             for i in range(k):
                 x = ((n-1)*x + s / x**(n-1)) / n
             return x
```

```
In [ ]:
```

```
In [72]: # the square root of 3
         nth_root_bis(3, 2)
```

```
Out[72]: 1.7320508075688772
```

```
In [ ]:
```

It is valid to write a Python function that does not contain `return`. For example

```
def welcome_message(name):
    s = "Hello " + name + "!"
    print(s)
```

or with no argument

```
def am_i_beautiful():
    return 'yes'
```

Exercise: - What do the above functions do? - Copy, paste and execute the function `welcome_message` in a code cell and call it with your name as argument. - What is the output value of `welcome_message`? (*hint*: use the `type` function that we used in the worksheet 1)

```
In [73]: def welcome_message(name):
             s = "Hello " + name + "!"
             print(s)
```

```
In [75]: def am_i_beautiful():
             return 'yes'
```

```
In [77]: welcome_message("Euler")
```

```
Hello Euler!
```

```
In [78]: out1 = welcome_message("Robert")
         out2 = am_i_beautiful()
```

```
Hello Robert!
```

```
In [79]: print(type(out1), type(out2))
```

```
<class 'NoneType'> <class 'str'>
```

```
In [ ]:
```

2.1 Recursion

We call a function recursive when it calls itself. The following is a recursive implementation of the factorial function

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

It can be viewed as the following mathematical definition of factorial

$$0! = 1 \qquad n! = n \times (n-1)!$$

Let us insist that in a recursive function you always need a special case for the initial step. Otherwise your code contains an infinite loop.

Exercise: - Copy paste the `factorial` function above. - Make a for loop in order to print the valuf of `factorial(n)` for `n` from `0` to `10` included. - Implement a recursive function `binomial_rec(n, k)` to compute the binomial number $\binom{n}{k}$. - Implement a recursive function `fibonacci_rec(n)` to compute the n-th Fibonacci number. - In each case could you compute how many function calls are done? - Write a recursive function with no initial condition. What kind of errors do you get when it is called?

In [139]:
```
def factorial_rec(n):
    if n == 0:
        return 1
    else:
        return n * factorial_rec(n-1)
```

In [140]:
```
for n in range(11):
    print(n, factorial_rec(n))
```

```
0 1
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
10 3628800
```

In [141]:
```
def binomial_rec(n, k):
    "Return the binomial number (n,k) (recursive implementation)"
    if k < 0 or k > n:
        return 0
    elif k == 0 or k == n:
        return 1
    else:
        return binomial_rec(n-1, k) + binomial_rec(n-1, k-1)
```

In [142]:
```
binomial_rec(4,2)
```

Out[142]: 6

In [143]:
```
binomial_rec(5,3) == factorial_rec(5) / factorial_rec(3) / factorial_rec
```

Out[143]: True

In [ ]:

In [144]:
```
def fibonacci_rec(n):
    "Return the n-th Fibonacci number (recursive implementation)"
    if n < 0:
        raise ValueError("n must be non-negative")
    elif n <= 1:
        return 1
    else:
        return fibonacci_rec(n-1) + fibonacci_rec(n-2)
```

```
In [91]: print(fibonacci_rec(0),fibonacci_rec(1),fibonacci_rec(2),fibonacci_rec(3))
```

```
1 1 2 3
```

```
In [92]: all(fibonacci_rec(k) == fib(k) for k in range(10))
```

```
Out[92]: True
```

```
In [ ]:
```

```
In [146]: # here is a possible way to count the number of calls using
          # global variables
          counter = 0
          def binomial_rec(n, k):
              global counter
              counter = counter + 1
              if k < 0 or k > n:
                  return 0
              elif k == 0 or k == n:
                  return 1
              else:
                  return binomial_rec(n-1, k) + binomial_rec(n-1, k-1)
```

```
In [147]: counter = 0
          binomial_rec(4,2)
          print(counter)
```

```
11
```

```
In [148]: counter = 0
          binomial_rec(6,3)
          print(counter)
```

```
39
```

```
In [149]: counter = 0
          binomial_rec(10,5)
          print(counter)
```

```
503
```

```
In [152]: # the number of function calls is ENORMOUS!
          for n in range(1,12):
              counter = 0
              binomial_rec(2*n, n)
              print(n, counter)
```

```
1 3
2 11
3 39
4 139
5 503
6 1847
7 6863
8 25739
9 97239
10 369511
11 1410863
```

In [ ]:

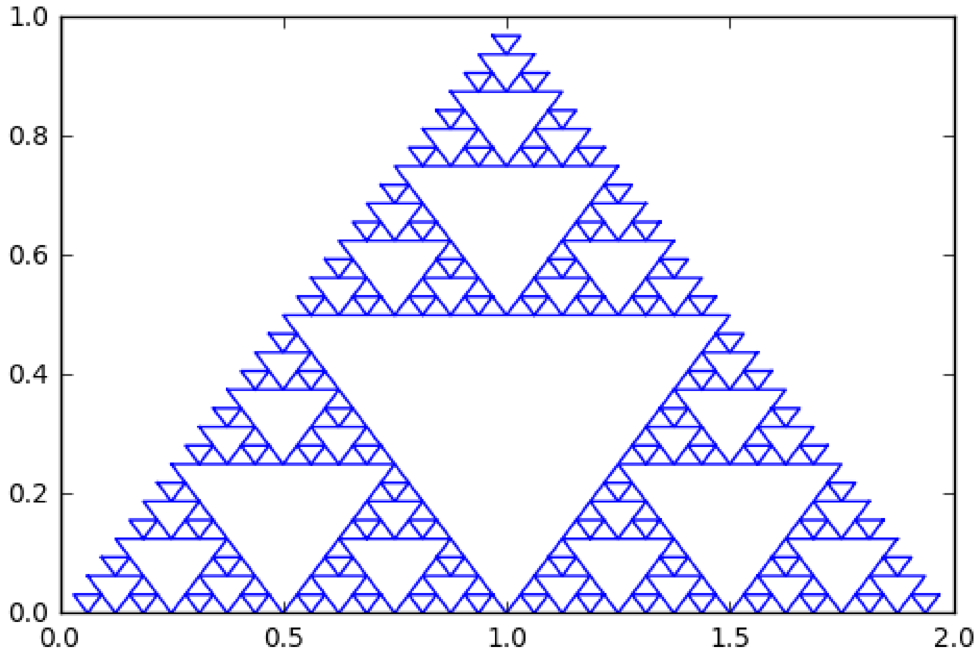
In [ ]:

Exercise: Use a recursive function to draw the following pictures

In [100]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [130]:
```
def recursive_figure(X, Y, depth):
    if depth != 0:
        Xmids = [(X[i]+X[i+1])/2 for i in range(3)]
        Ymids = [(Y[i]+Y[i+1])/2 for i in range(3)]
        plt.plot(Xmids + [Xmids[0]], Ymids + [Ymids[0]], 'b')
        recursive_figure([X[0],Xmids[2],Xmids[0],X[0]], [Y[0],Ymids[2],Ym
        recursive_figure([X[1],Xmids[0],Xmids[1],X[1]], [Y[1],Ymids[0],Ym
        recursive_figure([X[2],Xmids[1],Xmids[2],X[2]], [Y[2],Ymids[1],Ym
```

In [132]:
```
X = [0,2,1,0]
Y = [0,0,1,0]
recursive_figure(X, Y, 5)
plt.show()
```



In [ ]:

In [ ]:

In [ ]:

2.2 Extra exercises (optional, if you do all I buy you a beer)

Exercise (++): We have at our disposition three lists `l1`, `l2` and `l3`. At each step we are allowed to take the last element from a list and put it on the top of another list. But all lists should always be in increasing order.

The following example is a valid sequence of moves

```
l1         l2      l3

[0, 1, 2]  []      []
[0, 1]     []      [2]
[0]        [1]     [2]
[0]        [1, 2]  []
[]         [1, 2]  [0]
[2]        [1]     [0]
[2]        []      [0, 1]
[]         []      [0, 1, 2]
```

Write a (recursive) function that print all steps to go from

```
l1 = [0, 1, 2, 3, 4]
l2 = []
l3 = []
```

to

```
l1 = []
l2 = []
l3 = [0, 1, 2, 3, 4]
```

In [ ]:

In [ ]:

Exercise (++): Write a recursive function that solves the following problem: given a positive integer n and a positive rational number p/q find all solutions in positive integers $1 \leq a_1 \leq a_2 \leq \ldots \leq a_n$ that satisfies

$$\frac{1}{a_1} + \frac{1}{a_2} + \ldots + \frac{1}{a_n} = \frac{p}{q}.$$

(you could first prove that there are finitely many solutions)

In [ ]:

In [ ]:

Exercise (++): Write a function `plot_conic(a, b, c, d, e, f)` that plots the conic of equation

$$ax^2 + bxy + cy^2 + dx + ey + f = 0$$

.

In [ ]:

In [ ]:

Exercise (++): A square in a list is a sublist that is repeated twice. For example, the list `[0, 1, 2, 1, 2]` contains a repetition of the sublist `[1, 2]`. And `[0, 2, 1, 1, 2]` contains a repetition of `[1]`. A list that does not contain a square is called squarefree. For example, `[0, 1, 2, 0, 1]` is squarefree. - Write a function to check if a given list is squarefree. - How many lists containing only the numbers `0` and `1` are squarefree? - Find the smallest lexicographic squarefree list containing only the letters `0`, `1` and `2` and has length 100.

In [ ]:

In [ ]:

Exercise (++): For each recursive function you wrote make a non-recursive version.

In [ ]:

In [ ]: