UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Examination in:	INF1100 — Introduction to programming with scientific applications
Day of examination:	Wednesday, December 18, 2013
Examination hours:	09.00 – 13.00.
This examination set consists of 12 pages.	
Appendices:	None.
Permitted aids:	None.

Make sure that your copy of the examination set is complete before you start solving the problems.

- Read through the complete exercise set before you start solving the individual exercises. If you miss information in an exercise, you can provide your own reasonable assumptions as long as you explain them in detail.
- Most of the exercises result in short code where there is little need for comments, unless you do something complicated or non-standard. In that case, comments should convey the idea behind the program constructions such that it becomes easy to evaluate the solution.
- Many exercises ask you to "write a function". A main program calling the function is then not required, unless it is explicitly stated. You may, in these types of exercises, also assume that necessary modules are already imported outside the function. On the other hand, if you are asked to write a complete program, explicit import of modules must be a part of the solution.
- The maximum possible score on this exam is 75 points. There are 11 exercises, and the number of points for each exercise is given in the heading.

(Continued on page 2.)

Exercise 1 (4 points)

A function $b(t)$ is defined by the formula

$$b(t) = \begin{cases} 0.01, & t < 6, \\ 0.002, & t \geq 6 \end{cases}$$

Write a Python function for computing this mathematical function $b(t)$. Exemplify how to call the Python function to compute $b(4)$.
Solution:

```
def b(t):
    if t < 6:
        return 0.01
    else:
        return 0.002

# Or shorter
# b = lambda t: 0.01 if t < 6 else 0.002

value = b(4)
```

Exercise 2 (6 points)

What is printed by the following program?

```
print 3/11

a = 2
b = 3
a = 4
print b

a = 'dog'
b = a
a = 'cat'
print b

a = [8, 9, 10]
b = a[1:-1]
a[1] = 10
print b[0]
```

(Continued on page 3.)

```
from numpy import linspace
a = linspace(8, 10, 3)
b = a[1:-1]
a[1] = 10
print int(b[0])

a = {'dog': 'woof', 'cat': 'meow', 'fish': 'blub'}
b = a
a['dog'] = 'toot'
print b['dog']
```

Solution:

```
0
3
dog
9
10
toot
```

Exercise 3 (5 points)

Write down all the text that appears in the terminal window when the following program is run.

```
animals = ['dog', 'cat', 'bird', 'mouse', 'cow', 'frog',
           'elephant', 'duck', 'fish', 'seal']
sounds = ['woof', 'meow', 'tweet', 'squeak', 'moo', 'croak', 'toot',
          'quack', 'blub', 'ow ow ow']
for animal, sound in zip(animals, sounds):
    if animal in ('elephant', 'seal'):
        prefix = ' and the'
    elif animal in ('mouse', 'fish'):
        prefix = ' and'
    else:
        prefix = ''
    goes = 'says' if animal == 'duck' else 'goes'
    print prefix, animal, goes, sound

phrase = raw_input('\nContinue as you like: ')
print phrase
```

Solution:

```
 dog goes woof
```

(Continued on page 4.)

```
 cat goes meow
 bird goes tweet
 and mouse goes squeak
 cow goes moo
 frog goes croak
 and the elephant goes toot
 duck says quack
 and fish goes blub
 and the seal goes ow ow ow

Continue as you like: the fox says wa-pa-pa-pa-pa-pa-pow!
the fox says wa-pa-pa-pa-pa-pa-pow!
```

(No points are lost because of missing spaces at the beginning of lines)

Exercise 4 (5 points)

What is printed by the following program?

```
def neq(a, b, tol=1E-14):
    """Perform a != b with tolerance."""
    return not (abs(a - b) < tol)

def myfunc(x):
    if x < 10:
        return 1
    elif 10 <= x < 100:
        return 2
    else:
        raise ValueError('x=%g >= 100 is illegal.' % x)

if neq(myfunc(2), 1):
    print 'Bug!'
if neq(myfunc(15), 1):
    print 'Ok!'
try:
    y = myfunc(1000)
    if neq(y, 2):
        print x, y
except ValueError:
    pass
```

Solution:

```
Ok!
```

(Continued on page 5.)

Exercise 5 (5 points)

What is printed by the following program?

```
from math import exp

class A:
    def __init__(self, x=0):
        self.x = x

    def __call__(self, t):
        x = self.x
        return exp(-x)*t

class B(A):
    def __init__(self, x=0, y=1):
        A.__init__(self, x)
        self.y = y

    def __call__(self, t):
        x, y = self.x, self.y
        return x + y + t

a = A()
b = B(1)
print a(2.0)
print b(2.0)
```

Solution:

```
2.0
4.0
```

Exercise 6 (5 points)

A polynomial

$$p(x) = \sum_{j=0}^{N} d_j x^j,$$

can be represented as a dictionary `d` such that `d[j]` equals d_j (i.e., `j` is an integer key in the dictionary and the corresponding value is d_j). For example, $2 - 5x^6$ is represented by

(Continued on page 6.)

```
d = {0: 2, 6: -5}
```

Somebody has written a Python function `poly_diff(d)` that computes and returns the derivative $p'(x) = \sum_{j=1}^{N} j d_j x^{j-1}$ of a polynomial:

```
def poly_diff(d):
    for j in d:
        r[j-1] = j*d[j]
    return r
```

When trying out this function, the first problem is an exception `NameError: global name 'r' is not defined`. After correcting this, you discover that the derivative of the test polynomial $p(x) = 2 - 5x^6$ has an unnecessary term. Write a correct function.

Solution:

```
def poly_diff(d):
    r = {}
    for j in d:
        if j >= 1:
            r[j-1] = j*d[j]
    return r
```

Exercise 7 (10 points)

A file contains lines with numbers separated by blanks. Write a Python function `sum_file(inputname, outputname)` that reads such a file (with name in `inputname`), computes the sum of the numbers on each line, and writes a new file (with name in `outputname`) where each line consists of the read numbers (on the line) followed by their sum. Format the numbers in the output such that they appear in nicely aligned columns. The number of numbers per line may vary from line to line. There are no blank lines in the input file.

For example, an input file may look like

```
1.2500    3.00  4.50
2.25      4       4.50
3.25    5.00    0.50   0.5
4.250    6.2     1
```

The output file may then look like

```
  1.25   3.00   4.50   8.75
  2.25   4.00   4.50  10.75
  3.25   5.00   0.50    0.5   9.25
  4.25   6.20   1.00  11.45
```

(Continued on page 7.)

Solution:

```
def sum_file(inputname, outputname):
    infile = open(inputname, 'r')
    outfile = open(outputname, 'w')
    for line in infile:
        numbers = [float(word) for word in line.split()]
        s = sum(numbers)
        for number in numbers:
            outfile.write('%7.2f' % number)
        outfile.write('%7.2f\n' % s)
    infile.close()
    outfile.close()
```

Exercise 8 (10 points)

You flip a coin n times. We want to compute the probability of getting at least m heads. Write a program that applies the Monte Carlo simulation method for computing the approximate probability. Read n and m from the command line. Print out an error if n and m are missing on the command line. Also print out an error message if n and m cannot be converted to integers.

Solution:

```
import random, sys
try:
    n = int(sys.argv[1])
    m = int(sys.argv[2])
except IndexError:
    print 'n and m must be given as command-line arguments!'
    sys.exit(1)
except ValueError:
    print 'n and m cannot be converted to integers!'
    sys.exit(1)

N = 10000   # no of experiments
M = 0       # no of successes
for e in range(N):
    # Let tail=1, head=2
    coins = [random.randint(1, 2) for i in range(n)]
    if coins.count(2) >= m:
        M += 1
print """
Approximate probability of at least %d heads among %d coints:
%.2f
""" % (m, n, float(M)/N)
```

(Continued on page 8.)

Exercise 9 (5 points)

A differential equation, or system of differential equations, written on the generic form

$$u'(t) = f(u(t), t), \quad u(0) = U_0,$$

can be solved by tools in a class hierarchy `ODESolver`. The complete Python code of the superclass and a subclass in this hierarchy is listed below. One numerical solution technique for $u' = f(u, t)$ is Heun's method:

$$u_* = u_k + \Delta t f(u_k, t_k),$$
$$u_{k+1} = u_k + \frac{1}{2}\Delta t f(u_k, t_k) + \frac{1}{2}\Delta t f(u_*, t_{k+1}),$$

where u_k is the numerical approximation to the exact solution $u(t)$ at time $t = t_k = k\Delta t$. Write a subclass of `ODESolver` to implement Heun's method. The subclass code should be in a file `Heun.py`, separate from `ODESolver.py` (i.e., you need to import `ODESolver`).

```
import numpy as np

class ODESolver:
    """
    Superclass for numerical methods solving scalar and vector ODEs

      du/dt = f(u, t)

    Attributes:
    t: array of time values
    u: array of solution values (at time points t)
    k: step number of the most recently computed solution
    f: callable object implementing f(u, t)
    """
    def __init__(self, f):
        self.f = lambda u, t: np.asarray(f(u, t), float)

    def set_initial_condition(self, U0):
        if isinstance(U0, (float,int)):  # scalar ODE
            self.neq = 1
            U0 = float(U0)
        else:                            # system of ODEs
            U0 = np.asarray(U0)          # (assume U0 is sequence)
            self.neq = U0.size
```

(Continued on page 9.)

```
        self.U0 = U0

    def solve(self, time_points):
        """
        Compute solution u for t values in the list/array
        time_points.
        """
        self.t = np.asarray(time_points)
        n = self.t.size
        if self.neq == 1:  # scalar ODEs
            self.u = np.zeros(n)
        else:              # systems of ODEs
            self.u = np.zeros((n,self.neq))

        # Assume that self.t[0] corresponds to self.U0
        self.u[0] = self.U0

        # Time loop
        for k in range(n-1):
            self.k = k
            self.u[k+1] = self.advance()
        return self.u, self.t

class ForwardEuler(ODESolver):
    def advance(self):
        u, f, k, t = self.u, self.f, self.k, self.t
        dt = t[k+1] - t[k]
        return u[k] + dt*f(u[k], t[k])
```

Solution:

```
from ODESolver import ODESolver

class Heun(ODESolver):
    def advance(self):
        u, f, k, t = self.u, self.f, self.k, self.t
        dt = t[k+1] - t[k]
        u_star = u[k] + dt*f(u[k], t[k])
        u_new = u[k] + 0.5*dt*f(u[k], t[k]) + \
                0.5*dt*f(u_star, t[k+1])
        return u_new
```

Exercise 10 (10 points)

This exercise is a continuation of the problem setting in Exercise 9. The task now is to implement the so-called Iterated Midpoint Method as a subclass in

(Continued on page 10.)

the `ODESolver` hierarchy. The method is defined mathematically as

$$
\begin{aligned}
v_q &= u_k + \frac{1}{2}\Delta t\left(f(v_{q-1}, t_{k+1}) + f(u_k, t_k)\right), \\
& \qquad q = 1, \ldots, N,\ v_0 = u_k \\
u_{k+1} &= v_N.
\end{aligned}
$$

We assume that the number of iterations, N, is known and supplied as argument to the constructor of the class. We also simplify to the case of a scalar ODE with only one unknown function u. Write the code of the subclass, called `IteratedMidpointMethod`, located in a separate file `midpoint.py`. Show how you can use class `IteratedMidpointMethod` to solve the ODE problem $y(x)y'(x) = 1$, $y(0) = 1$, $x \in [0, 5]$.
Solution:

```
import numpy as np
import ODESolver

class IteratedMidpointMethod(ODESolver.ODESolver):
    def __init__(self, f, N=2):
        # Must store N and allocate v as an array
        ODESolver.ODESolver.__init__(self, f)
        self.N = N
        self.v = np.zeros(N+1)

    def advance(self):
        u, f, k, t, v, N = self.u, self.f, self.k, self.t, self.v, self.N
        dt = t[k+1] - t[k]
        v[0] = u[k]
        for q in range(1, N+1):
            v[q] = u[k] + 0.5*dt*(f(v[q-1], t[k+1]) +
                                  f(u[k], t[k]))
        u_new = v[N]
        return u_new

# Example:
solver = IteratedMidpointMethod(lambda y, x: 1./y, N=3)
solver.set_initial_condition(1)
y, x = solver.solve(np.linspace(0, 5, 101))
```

Exercise 11 (10 points)

This exercise presents a model for the spreading of a flu. The population is divided into three groups: susceptibles (S) who can get the flu, infected (I) who have developed the flu and who can infect susceptibles, and recovered

(Continued on page 11.)

(R) who have recovered from the flu and become immune. Let $S(t)$, $I(t)$, and $R(t)$ be the number of people in category S, I, and R, respectively. The following differential equations describe how $S(t)$, $I(t)$, and $R(t)$ develop in a time interval $0 \leq t \leq T$:

$$S'(t) = -b(t)S(t)I(t), \tag{1}$$
$$I'(t) = b(t)S(t)I(t) - qI(t), \tag{2}$$
$$R'(t) = qI(t). \tag{3}$$

At $t = 0$ we have the initial conditions $S(0) = S_0$, $I(0) = I_0$, and $R(0) = 0$. The function $b(t)$ and the constant $q > 0$ must be known.

Write a Python function `flu(S0, I0, b, q, T)` that takes the initial values S_0 and I_0, the function $b(t)$, the parameter q, and the end time T for the simulation as arguments. The function can apply a subclass of class `ODESolver` (see Exercises 9 and 10) to solve the differential equations, or you can write your own code. Four arrays should be returned from the function `flu`:

- `t` containing the time points $t_k = k\Delta t$, where the numerical solution is computed, $k = 0, 1, \ldots, n$,
- `S` containing $S(t_0), S(t_1), \ldots, S(t_n)$,
- `I` containing $I(t_0), I(t_1), \ldots, I(t_n)$,
- `R` containing $R(t_0), R(t_1), \ldots, R(t_n)$.

We look at the spreading of the flu at a boarding school and reason as follows to set appropriate values of the parameters needed in the model. At $t = 0$ there are 100 susceptibles and 1 infected. The value of $1/q$ reflects the average length of the disease, here taken as 7 days, so $q = 1/7$ (time t is measured in days). The function $b(t)$ measures how easy an infected person can infect a susceptible. In the beginning we assume $b(t)$ to be a constant equal to 0.01. After 6 days, $t \geq 6$, people are aware of the flu and become more careful to protect themselves such that $b(t)$ drops from the value 0.01 to 0.002 (Exercise 1 asks you to implement such a function). Use five time steps per day such that the total number of time points in the simulation is $5T + 1$.

Make a call to the function `flu` with the mentioned parameters and $T = 40$. Also add code for plotting $S(t)$, $I(t)$, and $R(t)$ in the same figure with legends for each curve.

Solution:

```
def flu(S0, I0, b, q, T):
    def f(u, t):
        S, I, R = u
        return [-b(t)*S*I,
                 b(t)*S*I - q*I,
                q*I]
```

(Continued on page 12.)

```
    solver = Heun(f)
    solver.set_initial_condition([S0, I0, 0])
    time_points = np.linspace(0, T, 5*T+1)
    u, t = solver.solve(time_points)
    S = u[:,0]
    I = u[:,1]
    R = u[:,2]
    return t, S, I, R

def b(t):
    if t < 6:
        return 0.01
    else:
        return 0.002

# or
# b = lambda t: 0.01 if t < 6 else 0.002
t, S, I, R = flu(S0=100, I0=1, b=b, q=1./7, T=40)
from scitools.std import plot
plot(t, S, t, I, t, R, legend=['S', 'I', 'R'])
```

END