Errata List for the book
"A Primer on Scientific Programming with Python
2nd edition
by H. P. Langtangen

Simple typos are not reported in the list below – only more serious errors that may lead to confusion.

1. Chapter 2.3.2: "Suppose we want to create `Cdegrees` as $-10, -7.5, -15, \ldots, 40$." The -15 after -7.5 should be -5.

2. Chapter 2.4.3, page 70: "Observe that `table[4:6]` makes a list ... with three elements" is wrong, as this makes a list of two elements. The sublist construction should instead read `table[4:7]` both in the interactive session and the running text.

3. Chapter 2.6.2: The data goes from 1929 up to and including 2009. The formula `n = 2010 - 1929 + 1` must therefore be replaced by `n = 2009 - 1929 + 1` in the file `sun_data.py` and the code snippet from the file found in the book.

4. Chapter 3.3.2, equation (3.6): the f in front of the first sum should be replaced by the number 4.

5. Exercise 5.13, page 232: The j in the formula right below (5.16) should be replaced by i, i.e., the denominotor should read $x_k - x_i$.

6. Exercise 6.11: The file path `src/basic/lnsum.py` is wrong. The right location is `src/funcif/lnsum.py`.

7. Page 440: The syntax `super(Line, self).methodname(arg1, arg2, ...)` is wrong. The correct syntax is `super(Parabola, self).methodname(arg1, arg2, ...)` (`super` takes the subclass name as first argument). Also, for `super` to work, the class must be new-style class, i.e., derived from `object`. One then has to define class `Line` as

   ```
   class Line(object):
       ...
   ```

8. Page 626: The code snippet must compare `ForwardEuler` and `RungeKutta4`:

   ```
   T = 3
   dt = 1
   n = int(round(T/dt))
   t_points = linspace(0, T, n+1)
   figure()
   for method_class in ODESolver.ForwardEuler, ODESolver.RungeKutta4:
       method = method_class(f)
       method.set_initial_condition(1)
       u, t = method.solve(t_points)
       plot(t, u)
   ```

```
    legend('%s' % method_class.__name__)
    hold('on')

t = linspace(0, T, 41)  # finer resolution for exact solution
plot(t, u_exact)
legend('exact')
title("u'=u solved numerically")
```

9. Page 637, Exercise E.9: The `Problem` class should take only h, T_s, and $T(0)$ as attributes (t_1 and $T(t_1)$ can be used for estimating h). The `estimate_h` method should take t_1 and $T(t_1)$ as arguments, compute h, and assign it to `self.h`.

10. Page 647, Exercise E.32: The code examples for parsing command-line arguments in are typical when using the `getopt` module to parse command-line arguments, but the text in the exercise refers to Chapter 4.2.4, which (in the 2nd edition of the book) describes the module `argparse` for parsing command-line arguments. The text in this exercise becomes clearer if one simply skips reading the `if option == ...` lines in the code examples. Adapting the example in Chapter 4.2.4 to Exercise E.32 is not straightforward as we want to have `pi` and other mathematical symbols in the values on the command line. To this end, treat all command-line arguments in `argparse` as strings and perform explicit type conversion in the `get_input` function. Here is an example.

```
def get_input(T=4*pi,
              dt=4*pi/40,
              initial_u=1,
              initial_dudt=0,
              method=RungeKutta4,
              m=1.0,
              friction=lambda dudt: 0,
              spring=lambda u: u,
              external=lambda t: 0,
              u_exact=None):

    ...
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--dt', '--time_step', type=str,
                        default=str(dt), help='time step')
    ...
    parser.add_argument('--u_exact', type=str,
                        default='None', help='exact solution')
    ...
    args = parser.parse_args()
    # Modify/interpret string arguments
    dt = eval(args.dt)
    T = eval(args.T)
    ...
    if args.u_exact == 'None':
        u_exact = None
    else:
        u_exact = StringFunction(args.u_exact,
                                 independent_variable='t')
        u_exact.vectorize(globals())  # allow array argument t

    makeplot(T=T, ...)
```

One may also use the `getopt` module instead of `argparse`.

11. Page 650: `self.solver.set_initial_condition(ic, 0.0)` must be `self.solver.set_initial_co` since the initial t value is supposed to be given in the `time_points` array argument to `ODESolver.solve`.

12. Page 652, Exercise E.36: The call to `read_cml_func` requires SciTools version (at least) 0.8.3. The call must also look like

```
self.spring = read_cml_func('--spring', lambda u: u, iv='u',
                            globals_=globals())
# Equivalent:
self.spring = read_cml_func('--spring', 'u', iv='u',
                            globals_=globals())
```

Here, the second argument is the default expression used when there is no `--spring` argument on the command line, and `iv` denotes the name of the indpendent variable if a mathematical string expression is given on the command line. The collection of all global names in the calling program (`globals()`) must be passed on to `read_cml_func` in case one would like to specify constructions like `CubicSpring(1.8)` (otherwise `read_cml_func` cannot know about the name `CubicSpring`).

The alternative to using `read_cml_func` and specifying values on the command line is to set the values directly in the program, as outlined in the exercise.