RAK-19006 Various Topics of Civil Engineering: **Python 3 for scientific computing**
Spring 2018, TUT, Tampere

Published 22.5.2018

Solutions to exercises, *lectures 5–6, 9–11*

1. ★☆☆ *NumPy and SciPy.*

   a) This works essentially the same way as in MATLAB:

```
import numpy as np
xx = np.linspace(0, 1, 101)  # 100, + 1 for the fencepost (to get 100 intervals).
yy = xx**2
```

   b) Here is how to create and solve a random linear equation system using NumPy:

```
import numpy as np
n = 5
A = np.random.random((n, n))
b = np.random.random((n,))
x = np.linalg.solve(A, b)
```

   Recall that in NumPy, a vector is a rank-1 tensor; there are no row or column vectors in NumPy. (Contrast MATLAB, which uses the matrix formalism.)

   c) To force the datatype, use the *dtype* kwarg:

```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=np.float64)
```

   This makes NumPy treat your input data as if they were floats.

   Most of the commonly used functions in NumPy that create arrays support the *dtype* kwarg, but some don't – notably *np.random.random()*. If you need random complex numbers (for example), it is possible to use *A + 1j*B*, where *A* and *B* are arrays of matching (or in general, broadcast-compatible) shapes, containing random real numbers.

   Of course, the strategy depends on what you want. If you would like random numbers from inside a circle in the complex plane, then obviously *A * np.exp(1j * np.pi * B)* is more appropriate. The point here is that NumPy provides random real numbers as a building block.

   d) This was covered in the lectures (lec. 6, slide 22; lec. 6, slide 17): NumPy provides a “basic” version of the linear algebra routines, while SciPy provides an “advanced” one.

   Usually SciPy has more options (e.g. a separate solver for symmetric or Hermitian systems, instead of just a general solver; may give higher performance if applicable to your problem), or can solve some problem types NumPy itself cannot (e.g. the generalized linear eigenvalue problem vs. just the standard linear eigenvalue problem).

e) Items e) and f) were basically exercises in skimming the documentation for the linear algebra modules of NumPy (*np.linalg*) and SciPy (*scipy.linalg*).

To solve an eigenvalue problem in SciPy, see scipy.linalg.eig(). For a general matrix ***A***:

```
import scipy.linalg
lam, x = scipy.linalg.eig(A)
```

For hermitian or symmetric ***A***, the corresponding routine is *eigh()*.

The same routines support also generalized linear eigenvalue problems:

```
import scipy.linalg
lam, x = scipy.linalg.eig(A, B)
```

To get the eigenvalues only, use *eigvals()* (general) or *eigvalsh()* (symmetric/Hermitian) instead. This is faster, so it can be useful if you don't need the eigenvectors.

f) Singular value decomposition. In NumPy, *numpy.linalg.svd()*:

```
import numpy as np
U, sigma, Vh = np.linalg.svd(A)
```

In SciPy, *scipy.linalg.svd()*:

```
import scipy.linalg
U, sigma, Vh = scipy.linalg.svd(A)
```

To get the condition number of a matrix: *np.linalg.cond()*:

```
import numpy as np
c = np.linalg.cond(A)
```

> ※ *Why is this last question here?* The documentation states that *cond()* can determine the condition number with respect to various different matrix norms; the default is the 2-norm condition number, which is internally obtained by computing the SVD.

2. ★☆☆ *Basic plotting*.

a) Plotting x, x^2, x^3, x^4 and x^5 into the same picture, with x in [0, 1]:

```
import numpy as np
import matplotlib.pyplot as plt

xx = np.linspace(0, 1, 100001)  # see item f) for why so many points.

plt.figure(1)
plt.clf()  # for Spyder (no auto-clear between runs)
for k in range(1, 6):
    plt.plot(xx, xx**k)
```

b) Adding a grid, x-axis and y-axis labels, and a title.

```
plt.grid(b=True, which='both')
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title('Some monomials of x')
```

c) Labeling each curve and adding a legend.

To make sure that, in the source code, each label definitely stays together with the curve it is supposed to label, we use the *label* kwarg of *plt.plot()*. The complete program so far:

```
import numpy as np
import matplotlib.pyplot as plt

xx = np.linspace(0, 1, 100001)

plt.figure(1)
plt.clf()  # for Spyder
for k in range(1, 6):
    plt.plot(xx, xx**k, label=r'$x^{}$'.format(k))

plt.grid(b=True, which='both')
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title('Some monomials of x')
plt.legend(loc='best')
```

d) Saving the resulting figure. All we need is:

```
plt.savefig('monomials.svg')
```

This always saves the currently active figure; to switch, use *plt.figure(n)*.

※ Running without Spyder, add *plt.show()* to the end to *see* the figure; saving does not need it.

e) [optional] *How many samples is enough for plotting a curve?*

This depends on the use case and is a matter of opinion; the point was to think about the issue.

My personal opinion is that for line curves, such as here, if the evaluation of the function is not prohibitively expensive, a very large number of points is preferable, because that allows zooming while still keeping the curve smooth.

This is useful for interactive exploration, avoiding the need to adjust the plotting script just to look at different parts of the function. In data visualization, convenience is important, because the price of inconvenience is to risk missing a potential discovery.

(Even page layout matters; Edward Tufte has a nice example, on comparing geographical distributions, in his classic book The Visual Display of Quantitative Information, 2nd ed., Graphics Press, 2001. It's a light, funny and extremely informative read; highly recommended for anyone dealing with visualization of quantitative data.)

If the width of the plot area in the figure window is, say, 1000 pixels (≈ half the screen width at full HD, the typical laptop resolution), then using 1e5 points allows zooming in by a factor of 100 (i.e. twice by a factor of 10), and still has enough data for the result to look smooth.

If there is no need to zoom – if the output is for a fixed-size non-interactive picture only – then a significantly lower number of points should be enough. However, keep in mind that the resolution used in printing to paper is commonly 300 or even 600 DPI, whereas for computer displays the pixel density is usually near 100 DPI. (CRT monitors used to be near 70 DPI!) The mobile phone industry is currently pushing this frontier (4K resolution in the palm of your hand), but laptops are still usually limited to full HD.

So, as a rough guideline, if you intend the output for printing – such as for a scientific journal – consider how wide the final picture will be in inches, and multiply by 600 to get the horizontal size in pixels. Since the data takes up most of the figure area (or it at least should!), this is also a reasonable number of points to use on the *x* axis to take advantage of the available resolution.

In any case, be aware that for changing the size of a raster image – i.e. displaying it at any other number of pixels than the image has – high-quality resampling [1] [2] [3] is both essential and nontrivial to do right. In downsampling specifically, the main issue is to get rid of spatial frequencies higher than can be displayed at the target size, to eliminate aliasing artifacts.

The issues hardly stop there; even drawing a smooth-looking straight line, at an angle, is difficult on a raster device, and many programs do not even bother to try. The developers of Matplotlib, on the other hand, have realized this, and Matplotlib uses the Anti-Grain Geometry (AGG) graphics library. (Original historical homepage; current SourceForge project site.)

Finally, line curves have the advantage that for a resolution of n pixels (along the curve, say), they need only $O(n)$ function evaluations. Surface data, on the other hand, needs $O(n^2)$ evaluations, volume data needs $O(n^3)$ evaluations, …so with a given amount of computing resources, the available wall time and RAM become limiting factors much earlier than for line curves. (Unless we use some adaptive algorithms to decide how to distribute the points to where they matter the most, instead of using the classical uniform grid.)

3. ★★☆ *Intermediate plotting*.

a) Just some comments here: *linestyle* and *linewidth* are useful kwargs for *plt.plot()* when plotting multiple curves into the same picture. The basic styles are 'solid' and 'dashed', and *linewidth* simply takes a positive real number. It can also be smaller than 1; the antialiasing in the AGG library will take care of it.

The *color* option accepts (R,G,B) and (R,G,B,A) tuples, HTML RGB (and RGBA) colors as strings such as '#808080', some named colors, and some MATLAB classics ('b', 'r', 'g', 'k', etc.). Note that the default color has changed in Matplotlib 2.0, which revised all the defaults [PyData 2016 Carolinas talk by Thomas Caswell]; the default line is now light blue, different from 'b'.

The *marker* option activates markers (for the data being plotted in that plot command) and chooses the marker type. The *markersize* option scales the visual size of the markers.

The markers are placed ***at each data point***. When using markers, it may be useful to prepare two copies of your data; one with lots of points to get a smooth curve, and another with much fewer points, to use for placing the markers. For example:

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    x = np.atleast_1d(x)
    return np.sin(2 * np.pi * x)

xx1 = np.linspace(0, 1, 100001)  # 1e5 points plus fencepost...
xx2 = [0, 0.25, 0.5, 0.75, 1]        # just 5 points!
yy1 = f(xx1)
yy2 = f(xx2)

plt.figure(1)
plt.clf()
plt.plot(xx1, yy1, linestyle='solid', linewidth=1.0, color='k')
plt.plot(xx2, yy2, linestyle='None', marker='o', color='k', label=r'$\sin(2 \pi x)$')
plt.grid(b=True, which='both')
plt.legend(loc='best')
```

※ To switch line drawing off, *linestyle* is the string 'None', not the actual **None**!

b) Subplots with *plt.subplot()*.

Here's a complete program:

```
import numpy as np
import matplotlib.pyplot as plt

# configuration
nrows = 3
ncols = 2
fs = [lambda x: np.sin(2 * np.pi * x),
      lambda x: np.cos(2 * np.pi * x),
      lambda x: np.sinh(2 * np.pi * x),
      lambda x: np.cosh(2 * np.pi * x),
      lambda x: x**2,
      lambda x: x**0.5]
x1 = 0
x2 = 1
nx = 100001

# main program
xx = np.linspace(x1, x2, nx)

plt.figure(1)
plt.clf()

for row in range(nrows):
   for col in range(ncols):
      k = ncols*row + col  # subplots use “row-major order”…
      f = fs[k]
      plt.subplot(nrows, ncols, 1 + k)  # …and 1-based indexing
      plt.plot(xx, f(xx))
      plt.grid(b=True, which='both')
      plt.xlabel(r'$x$')
      plt.ylabel(r'$y$')

plt.suptitle(r'Some functions $y = f(x)$')
```

c) Placing the *y*-axis labels of the right column of subplots onto the right side. Let's update the solution from item b). Modified complete program:

```
import numpy as np
import matplotlib.pyplot as plt

# configuration
nrows = 3
ncols = 2
fs = [lambda x: np.sin(2 * np.pi * x),
      lambda x: np.cos(2 * np.pi * x),
      lambda x: np.sinh(2 * np.pi * x),
      lambda x: np.cosh(2 * np.pi * x),
      lambda x: x**2,
      lambda x: x**0.5]
x1 = 0
x2 = 1
nx = 100001

# main program
xx = np.linspace(x1, x2, nx)

plt.figure(1)
plt.clf()

for row in range(nrows):
   for col in range(ncols):
      k = ncols*row + col
      f = fs[k]
      ax = plt.subplot(nrows, ncols, 1 + k)  # grab a ref to the axis object, we'll need it…
      plt.plot(xx, f(xx))
      plt.grid(b=True, which='both')
      plt.xlabel(r'$x$')
      plt.ylabel(r'$y$')
      if col % 2 == 1:  # right column
         ax.yaxis.set_label_position('right')  # …to place the y-labels on the right side.

plt.suptitle(r'Some functions $y = f(x)$')
```

※ Can also use *plt.gca()* to grab a reference to the currently active axis object.

In some cases, we may need to omit adding the *xlabel* for the first two rows, but here we have used the fact that the plot areas have opaque white backgrounds by default, so if the xlabel won't fit, it will be automatically hidden under the subplot below it. But depending on the size of the figure window, the label may not be completely hidden, so let's fix that.

There are also other customizations that can be done; e.g. switch *x*-axis tick labels off for the first two rows, because (in this particular example) the *x* values are the same in all rows.

Another nice customization is to switch the *spines* off for the right and upper sides, as they contain no useful information, only adding visual noise. (***Maximize data ink.*** –Edward Tufte)

Performing these three changes, here's the final complete program:

```
import numpy as np
import matplotlib.pyplot as plt

# configuration
nrows = 3
ncols = 2
fs = [lambda x: np.sin(2 * np.pi * x),
      lambda x: np.cos(2 * np.pi * x),
      lambda x: np.sinh(2 * np.pi * x),
      lambda x: np.cosh(2 * np.pi * x),
      lambda x: x**2,
      lambda x: x**0.5]
x1 = 0
x2 = 1
nx = 100001

# main program
xx = np.linspace(x1, x2, nx)

plt.figure(1)
plt.clf()

for row in range(nrows):
   for col in range(ncols):
      k = ncols*row + col
      f = fs[k]
      ax = plt.subplot(nrows, ncols, 1 + k)
      plt.plot(xx, f(xx))
      plt.grid(b=True, which='both')

      # see https://matplotlib.org/examples/ticks_and_spines/spines_demo.html
      ax.spines['right'].set_visible(False)
      ax.spines['top'].set_visible(False)
      ax.yaxis.set_ticks_position('left')
      ax.xaxis.set_ticks_position('bottom')
      plt.ylabel(r'$y$')
      if row == nrows - 1:
         plt.xlabel(r'$x$')
      else:
         ax.xaxis.set_major_formatter(plt.NullFormatter())  # see StackOverflow
      if col % 2 == 1:  # right column
         ax.yaxis.set_label_position('right')

plt.suptitle(r'Some functions $y = f(x)$')
```

d) Independent *y*-axes.

See the example in the gallery. Here's a complete program:

```
import numpy as np
import matplotlib.pyplot as plt

f = lambda x: -6 * x**5 + 15 * x**4 - 10 * x**3 + 1
g = lambda x: (1 / 2) * (-x**5 + 3 * x**4 - 3 * x**3 + x**2)

xx = np.linspace(0, 1, 100001)

plt.figure(1)
plt.clf()

ax1 = plt.gca()
lines1 = ax1.plot(xx, f(xx), color='b', label=r'$f$')
ax1.grid(b=True, which='both', color='#ccccff')

ax2 = ax1.twinx()
lines2 = ax2.plot(xx, g(xx), color='r', label=r'$g$')
ax2.grid(b=True, which='both', color='#ffcccc')

# see StackOverflow for how to use legend() with twinx()
all_lines = lines1 + lines2
all_labels = [line.get_label() for line in all_lines]
plt.legend(all_lines, all_labels, loc='best')
```

※ The *twinx()* method ***creates a new, independent axis object***, sets the visibility of its *patch* (the background) to **False**, so that the original axis shows through, and sets up its *y*-labels to appear on the right side.

The implications of this behavior occasionally trip up new users.

For example, the grid of ax2 shows on top of the data of ax1. This is because the two axis objects are drawn independently; their drawing cannot be interleaved, so the *zorder* kwarg cannot help. See issue #7984 and StackOverflow.

One possible workaround here is to use *four* axis objects; the bottommost two for the grids, and the top two for the actual data. To ensure that the ticks match, it may be useful to plot another copy of the data – invisible, no line, no markers – onto the axis that holds the grid, and then plot the same data again using the desired style onto the axis that will hold the data.

This is left as an (advanced) exercise to the reader. (*Hint*: search the internet for how to create axis objects manually. Combine with what you know about what *twinx()* actually does. You may need to hide the ticks, labels and spines for the duplicate axes.)

4. ★★☆ *3D plotting*.

a) Plotting the *wireframe* of $f(x, y) = \sin(\pi x) \cos(\pi y)$ for x and y in the unit square. Let's use the template from lecture 5. Here's a complete program:

```
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as axes3d

f = lambda x, y: np.sin(np.pi * x) * np.cos(np.pi * y)

xx = np.linspace(0, 1, 101)
yy = np.linspace(0, 1, 101)

X, Y = np.meshgrid(xx, yy)
Z = f(X, Y)

fig = plt.figure(1)
plt.clf()
fig.patch.set_color((1,1,1))  # fig. background, RGB
fig.patch.set_alpha(1.0)  # fig. background, opacity
left_bot_w_h = [0.02, 0.02, 0.96, 0.96]  # more space to edges
ax = axes3d.Axes3D(fig, rect=left_bot_w_h)

ax.plot_wireframe(X, Y, Z)
```

b) Adding axis labels and a title. Setting a nice camera angle.

```
ax.view_init(15, 130)  # elev, azim
ax.axis('tight')
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
ax.set_title(r'$z$')  # note! No “zlabel” due to Matplotlib's history as a 2D plotter.
```

c) Experimenting with the options of the wireframe command to get a nice-looking plot. Replace the plotting command above with this:

```
ax.plot_wireframe(X, Y, Z, rcount=5, ccount=5)  # use a grid of 4 × 4 rectangles
```

Matplotlib versions earlier than 2.0 used *rstride* and *cstride*, which set the step to use when reading the data array. These have now been deprecated in favor of *rcount* and *ccount*, making wireframe behave somewhat similarly to linspace. The user only specifies how many samples to use, and the wireframe command does the sampling automatically. **Note the fencepost**; e.g. to get a grid of 10 × 10 rectangles, one needs to use *rcount=11*, *ccount=11*.

Note that if we have enough data, we get smooth lines along the edges of the rectangles, even though we used very few rectangles – this is not just a linear interpolation of the vertex values.

d) Plotting the same function as a filled surface (instead of a wireframe). Replace the plotting command above with this:

```
ax.plot_surface(X, Y, Z)
```

See the tutorial for options.

e) Making a colorsheet (as seen from the top) and adding a colorbar.

This is a 2D plot – now we don't need *Axes3D*. Here's a complete program:

```
import numpy as np
import matplotlib.pyplot as plt

f = lambda x, y: np.sin(np.pi * x) * np.cos(np.pi * y)

xx = np.linspace(0, 1, 101)
yy = np.linspace(0, 1, 101)

X, Y = np.meshgrid(xx, yy)
Z = f(X, Y)

fig = plt.figure(1)
plt.clf()

plt.pcolormesh(X, Y, Z)
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'$z$')

plt.colorbar()
```

5. ★★☆ *Version control*. (Lecture 9)

a) This exercise was about walking through the step-by-step crash course on *git*, in the lecture notes, pp. 80–88. All the explanations (and links to further reading) are already there, so there is no need for a separate solution. See also lecture 9, slides 20–22.

But if you want some more further reading, see Joel Spolsky's essay on DVCS. Quoting:

> *I know, it's strange… since 1972 everyone was thinking that we were manipulating versions, but, it turned out, surprisingly, that thinking about the* ***changes themselves*** *as first class solved a very important problem: the problem of merging branched code.*

b) [optional] This part was about trying GitHub in practice. GitHub's Hello World guide already provides the explanations, so not much remains to explain here. See also lecture notes, p. 77, and lecture 9, slide 23 (licensing – to make it legal for others to contribute).

Used with a free account, GitHub is mainly useful for collaborating on open-source projects (such as NumPy, SciPy, …), and for publishing your own open-source codes, whether or not you intend to collaborate with anyone.

The advantage of using a DVCS site such as GitHub – beside the benefits of version control – is that your source code is stored in the cloud, so you (and indeed anyone) can access it from any web browser or any computer that has *git*. It's also very easy to synchronize between the cloud and local copies via *git push* and *git pull*.

It's not a replacement for backups, but still it already makes it much less likely for a hard disk failure or a mistyped command to suddenly destroy your project.

6. ★★★ ***Let** it be*. (Lectures 10–11)

   https://github.com/Technologicat/python-3-scicomp-intro/blob/master/examples/let.py

   How the code works: we have essentially put together some techniques we have seen, namely the bunch design pattern (also lecture notes, pp. 41–42), the context manager (lec. 3, slides 12–14; lecture notes, p. 40), redirection of attribute lookup (dynscope.py, no_rebind.py), and decorators (with and without parameters; lec. 3, slide 41; lec. 11, slide 33; lecture notes, p. 49).

   a) The outermost **def** is the decorator factory, allowing the user, in effect, to give arguments to the decorator. The arguments travel to the inner **def**s via the lexical closure property; their values are captured from the outermost **def** when the decorator factory is called.

   Here the arguments are essentially a dict, constructed from any arguments passed to the decorator factory by name, using dictionary packing (** in the **def**).

   The middle **def** is the decorator itself. It receives the function being decorated, and returns whatever object is intended to replace it (usually a decorated version of the function).

   The innermost **def** is the decorated function, that will replace the original function definition. The *bindings* travel here by lexical closure. The decorated function passes through any other arguments, and adds a named argument *env* containing the bindings packed into a bunch.

   Why the factory pattern? Consider how Python's decorators work behind the scenes. Here:

```
@deco_factory(args)
def f(x):
    ...
```

   the *deco_factory(args)* part runs first (without the @), and then **its return value** is taken as the decorator to use with the @. Let's call the return value *deco*. We now have:

```
@deco
def f(x):
    ...
```

   which is just syntactic sugar for

```
def f(x):
    ...
f = deco(f)
```

   Hence, the original is syntactic sugar for

```
def f(x):
    ...
f = (deco_factory(args))(f)
```

b) The combined decorator **let** is a decorator factory.

It first calls the function *let_over_def* (which is another decorator factory), passing along the given bindings. This produces a decorator, with the bindings "frozen in" by lexical closure, as explained in item a).

Then, **let** defines the final decorator. It is simply the function composition of the above decorator (applying that first), and *immediate* (which is a decorator).

Finally, **let** returns this final decorator. This return value is the decorator function Python applies to the decorated function at the use site.

This is the kind of code you need to write if you want to increase user convenience by creating combined decorators for common use cases. Contrast the syntax you would use, if you were to use decorator composition (cf. function composition) at the use site. The combined decorator we have created here,

```
@let(x = 5)
def _(env):
    ...
```

is equivalent with this composition:

```
@immediate
@let_over_def(x = 5)  # Python applies the innermost decorator first
def _(env):
    ...
```

The main point here is that when implementing decorators, one must think in terms of how things work behind the scenes; the syntactic sugar only applies to use sites. Understanding how it desugars makes decorator definitions fairly easy to read, even if they contain nested **def**s.

c) The function **letexpr** works by calling the given *body*, which is expected to be a one-argument function (duck typing!), passing in the given bindings as an *env* object (which itself is an implementation of the bunch pattern).

This is where having an *env* class pays off; the approach taken in **letify** – a simple abuse of syntax – does not lend itself to use with **lambda**s. Here we have made a reusable component (the *env* class) that can be used in the implementation of both versions of **let** – a decorator for **def**s, and a function that can be used with **lambda**s.

The *body* parameter is singled out (from among the kwargs in **bindings) by a standard Python trick: we simply declare *body* as a named formal parameter in the function definition. If the caller gives us either a single positional argument, or a named argument with name *body*. that value is bound to this name *body*. Only *any additional named arguments* go into kwargs.

d) The simpler **letify** decorator has essentially the same effect as the more complex **let** decorator, so for this particular use, the simpler one is sufficient.

Further, because

```
@letify
def _(x = 5):
    ...
```

is just syntactic sugar for

```
def dummy(x = 5):
    ...
dummy = dummy()
```

we don't really need **letify**, either, if our only goal is to write **let**-like bindings where the names and corresponding values appear together at the start of the code block. The **def** block already does that, although we must remember to call it manually; the **letify** decorator automates that.

e) [optional] Some possible viewpoints:

- S*imple is better than complex*. The **letify** implementation is two lines of code.
- But the more complex **let** implementation buys us extra features (**letexpr**).
- In **let**, we need to access the bindings as *env.foo*, which adds visual noise.
- On the other hand, this **let** is a compromise between call stack inspection and “don't do that", similar to *dynscope.py*.
- *Explicit is better than implicit.* How much magic is acceptable? Do we need a let construct?

f) [optional] Some possible viewpoints:

- If one insists on using **let** and “let over lambda” in Python, then obviously yes, these are useful.
- Slightly magic, and not very pythonic. Maybe out of place in Python?
- *Let over lambda* is like an object instance, and Python already has syntax for that. Koan on objects and closures.
- On the other hand, lexical closures make it possible to have truly private data. (At least until someone **import**s *inspect*.)
- On the yet other hand, *we are all responsible users here*. In the Python community, there is no need for language-enforced private data; a name beginning with an underscore should be enough to mark as private.

In my personal opinion, **letexpr** is perhaps the most useful part here. The trivial option of abusing another **lambda** (and immediately calling it) makes code unreadable, and local bindings for a **lambda** are a feature that is occasionally useful (consider the list uniqifier we saw earlier on the course, that was also used as an example in *let.py*). The **begin** function can also be useful, if one needs to perform a side effect (e.g. a debug print) in a **lambda**.

g) [optional] *What, if anything, do decorator factories and macros have in common?*

Decorator factories give rise to parametric decorators, i.e. one can give arguments to them.

Both decorators (with factories) and macros are, in essence, parametrizable source filters.

Macros are much more powerful; they can perform practically any local edit to the AST, whereas decorators are limited to a particular, very specific kind of edit.

However, it happens that this particular source transformation, that is performed by Python decorators, lends itself to many practical uses. Among these are things like the **let** constructs we saw here, and the continuation magic in the **tco** library. Numba uses a decorator to insert a JIT compiler. Some profilers, notably *line_profiler* and *memory_profiler*, use a decorator to insert the magic needed for tracing. Pre- and postconditions of contract programming can be implemented as decorators. Debug logging, e.g. to capture and show function arguments (and its output) whenever the function is called, can also be inserted as a decorator.

In general, (almost?) any procedure that needs to read and/or change the input or the output of a function, or possibly inspect the function object itself, can be implemented as a Python decorator. Decorators customize what it means to call (or even define!) a function, on a per-function basis.

(Often it is the call that is customized, but the *immediate* decorator we defined changes the act of *defining* a function into "immediately run this block of code in a new lexical scope".)

Finally, a side note on syntax, Python and Lisps: the symbol datatype in Lisps comboes well with syntactic macros.

Essentially, a macro says "*Here's a new AST node type for you, and here's the code to run when you see one…*", where the new AST node type is named by "*this arbitrary symbol I just picked*". In Lisps, *the user* can create new AST node types, which behave just as if they were part of the core language!

The uniformity of Lisp's syntax (lec. 10, slide 25) makes it trivial for the parser to recognize any custom AST node types – because the first item in any list that contains runnable code always is the operator. (This property does not necessarily hold in the sublists in a macro invocation; in general, the *input* to a macro is data, not necessarily code.)

In contrast, in most other languages, available AST node types are fixed, so e.g. in Python we can't add new ones if the Python interpreter is to be able to parse the code (without requiring a custom preprocessor, which would first expand the macros into something Python's parser recognizes).

Hence, even MacroPy is somewhat limited in the syntax it can offer. There is no "match" syntactic element, but a function-call-ish thing and a bunch of **if**s, which are already recognized by Python's parser.

7. ★★★(★+) *Monads.* ***In Python.***

a) From a software engineering viewpoint, a **monad** *generalizes function composition*, performing a specific kind of custom operation between the actual function invocations.

This is a *useful refactoring* in a wide variety of situations. Extracting this “between-steps” behavior into a central location can eliminate significant amounts of boilerplate code.

Importantly, this *makes operations chainable*, when they otherwise would not be.

This being a Python course, that would be enough for an answer. Now, let's go a bit further!

As an illustrative example, consider multivalued (mathematical) functions. Let *f* be a function of type $a \rightarrow [a]$, i.e. taking one parameter of type *a*, and returning a list of these *a*s. One example of such an *f* is the multivalued square root, which was considered in the example code.

So, if we take an *x* of type *a*, and compute *f(x)*, we get a list of *a*s as the output. In Python notation, let *A = f(x)*. Say we would then like to apply another function *g*, of the same type as *f*, to the list *A*. We cannot just ”*g(A)*”, because *A* is a list of *a*s, and *g* expects a single *a* as input. Hence, to chain the multivalued functions *f* and *g*, we need some glue code that walks over *A*, applies *g* to each item, and then combines the result lists into one list.

In Lisps, this combination of mapping and then flattening the result by one level is known as *flatmap*. In Python notation, what we actually need to do is thus *flatmap(g, f(x))*. A rough sketch in Python:

```
def flatmap(proc, lst):
    lol = []  # list of lists
    for x in lst:
        lol.append(proc(x))
    flattened = []
    for sublist in lol:
        for item in sublist:
            flattened.append(item)
    return tuple(flattened)  # immutable output more FP
```

Of course, that's a silly version to illustrate the steps; it can be streamlined somewhat:

```
def flatmap(proc, lst):
    out = []
    for x in lst:
        out.extend(proc(x))
    return tuple(out)
```

Or even:

```
flatmap = lambda proc, lst: tuple(y for x in lst for y in proc(x))
```

(The leftmost **for** is the outer loop. Keep in mind *proc(x)* returns a list.)

Testing it:

```
def f(x):
    return tuple(chr(ord("a") + k) for k in range(x))   # ('a', 'b', 'c', …)
def g(x):
    return (x, 2*x, 3*x)
print(flatmap(g, f(4)))  # ('a', 'aa', 'aaa', 'b', 'bb', 'bbb', 'c', 'cc', 'ccc', 'd', 'dd', 'ddd')
```

This is the traditional, “lispy” FP solution: we have refactored the glue into a reusable higher-order function, *flatmap*. But we still have to remember that *flatmap*, specifically, is the tool we need whenever dealing with the chaining of – specifically – multivalued functions.

The idea of *making unchainable operations chainable* applies also to many other use cases, so we will have to remember the names of many specific tools, and which to use where. Is there another, yet more general scheme to refactor the glue, which would avoid this issue?

It turns out the answer is yes. In Haskell, the glue code is packed into the **List monad**. It's a specific type of monad, but from a programming viewpoint, **a** monad is a general API – any customized chaining of operations *always uses the same API*. This frees the programmer to think about what needs to be done, instead of the specifics of how:

```
do
  y <- f x
  g y
```

which is syntactic sugar (*do notation* [1] [2]) for:

```
f x >>= (\y -> g y)
```

Note the absence of a list-specific *flatmap*. It's not just the naming; the same chaining syntax is used with all monads. Yet only the List monad applies *flatmap*; the others do different things.

In Haskell, the backslash \ means λ; it looks somewhat similar, while being ASCII. The expression *Ma >>= f* means *bind(Ma, f)* (the latter in Python notation). To understand what *bind* does, it is helpful to look at its type signature: $M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$.

What does that even mean? If you feel your brain is starting to melt, recall slide 17 in lec. 10, where we talked about *currying*. The above is the same as $M\ a \rightarrow ((a \rightarrow M\ b) \rightarrow M\ b)$; it's essentially a curried two-argument function where the first argument is of type *M a* – i.e. an *a* packed into a monad *M* – and the second is of type $(a \rightarrow M\ b)$ – i.e. a function that takes an *a* to an *M b*, the result being packed into the same type of monad *M*.

So *bind* unpacks the data from the monad, applies the given function to that data, and then returns an “*M b*”. (Above, *a* and *b* have been the same type, hence we have had only “*a*”; but in general, they can be different.)

In the case of List, *bind* makes it so that the *y* in the λ we bind into becomes each item of the list, in turn. For each *y*, this λ returns a List. These Lists are then concatenated (in *bind*) into one result List – i.e. specifically for the List monad, *bind* essentially is *flatmap*.

In Python, in terms of our *monads.py*, the above line of Haskell becomes

```
f(x) >> (lambda y: g(y))
```

where we use >> to denote *bind* instead of >>= due to Python's syntactic limitations.

A complete example using *monads.py*:

```
def f(x):  # int → List str
   return List(chr(ord("a") + k) for k in range(x))   # ('a', 'b', 'c', …)
def g(x):  # str → List str
   return List(x, 2*x, 3*x)
print(f(4) >> (lambda y: g(y)))
```

Cf. the *flatmap* version above; we have only changed the return types and the invocation.

⚠ When reading about monads, you will come across **return**, which means something completely different from its regular meaning in programming. It is also called **unit**; it basically wraps a bare data value into a monad. The name is a pun: since the return value of the function we bind into must be an *M b*, we can use **return** to actually *return* (in the usual sense) a given *b*, while keeping the computation chainable.

In *monads.py*, for clarity, we have used the term **unit**. Also, we have placed this functionality either in the instance constructor, or in a class method named *unit*. In monads which have no other use for the constructor, we have made the constructor perform the job of **unit**; otherwise, we have defined a class method named *unit*.

(In monads that are essentially data containers, such as *Maybe*, *List* and *Writer*, the constructor is not needed for other purposes. But in monads that encapsulate computations, such as *Reader* and *State*, the constructor wraps a bare function into the monad, whereas **unit** wraps a bare *data value*, of the same type that the wrapped function returns.)

※ Monads that encapsulate computations become much easier to understand by thinking of them as containers for a *promise* to produce data of the given type, i.e. the type of the return value of the encapsulated computation. For example, in the case of *Reader*, this is the *a* in *(e → a)*.

The data “contained in the monad” is then essentially of type *a*; from the container viewpoint, the encapsulated function *(e → a)* is a minor technical detail. To actually produce a data value, the promise must be forced – i.e. the function must be called.

It then becomes immediately obvious that wrapping a computation (storing a user-given promise) and wrapping a given data value (by converting it into a promise; what **unit** does) must be different operations, as they have different argument types.

But don't take this idea too seriously – it's just something that may help if you find it easier to think of monads as containers rather than as computations. For *State* and *Reader*, the “minor technical details” – the functions – are what make them useful!

Some other use cases where we may think of making unchainable operations chainable:

● *Maybe*: a container for an optional value that may or may not be there. Composition is customized to skip the rest of the chain of composed functions when the value is missing.

Maybe implements the chaining of LBYL error handling (lec. 3, slide 10; contrast EAFP) in a pure FP way, with no need for if/elses at the call site – they have all been refactored into the *Maybe* monad!

● *Either*: a sum type of two mutually exclusive options; a generalization of *Maybe*.

● *Writer*: carry a debug log along.

● *Reader*: compose computations that may read from a shared environment.

● *State*: compose state-dependent computations.

So monads are containers, monads are computations – and, if you read about Haskell, you'll very quickly encounter also the IO monad – which encapsulates I/O actions.

Monads, among other uses, allow pure FP programs to do impure things, such as I/O – essentially because we can then hide these things *in between* the steps of the pure FP computation. [1, item 3]. But it is *not necessary* for I/O to be a monad, specifically; rather, that is merely an (or even possibly *the most*) obvious pure FP solution. [2] [3]

Also important, in the context of Haskell, is that *evaluating the definition of an action* (such as an I/O action) is different from actually *executing that action*. The definition takes place in the pure expression language, whereas the execution *does not need to* [1] [2]. The code needed to execute the action can be part of the runtime library [3].

See also:

> Why are side effects modeled as monads in Haskell? (StackOverflow)
> Haskell pre-monadic I/O (StackOverflow, on Haskell history)
> Haskell's IO explained in the same style as "You could have invented monads".

b) *Why should you make sure your own monads (if you define some) satisfy the monad laws?*

Two equally valid viewpoints:

> 1) *Software engineering*: Predictability, ability to reason about code.
> Precedence of function composition in a chain should not matter,
> i.e. it should be associative. In point-free style, $(f\ g)\ h = f\ (g\ h) = f\ g\ h$.
>
> 2) *Mathematics*: Strictly speaking, it would not be a **monad** if it didn't
> satisfy the **monad laws**.

As for what the laws are, see Stephan Boyer (2012): Monads, part 1: a design pattern.

c) Mostly this item was just about reading enough background material to get some level of understanding about monads – a useful, nontrivial refactoring technique that has its place in any programmer's toolkit, no matter the language (as long as high-level enough; Python is fine). But there was also this specific question:

> *On the other hand, the above is a very abstract way of thinking about a very common operation. As the simplest examples, consider the Maybe and List monads. What is actually being done when they bind? What would you have to do manually, if you did the same without monads?*

The *List* case was analyzed above – it's essentially *flatmap*.

In the case of *Maybe*, the *bind* function checks whether the data value is there, and based on this, it either applies the function we bind to, or if the value is missing, skips the computation and leaves the new value as *Nothing*. It's a centralized if-else check, which *occurs between the steps of the computation* – which is exactly what LBYL error handling needs. (But in impure computations, prefer EAFP; see Time of Check to Time of Use (TOCTTOU).)

Finally, some notes on monads and/or Haskell:

● In the context of Lisp, we talked about *code as data*: macros, and the homoiconicity of Lisp syntax. Haskell also blurs the line between code and data, but in a very different way: memoized pure functions can be thought of as lookup tables, and on the other hand, things that look like data structures may actually be implemented as functions (e.g. the naturals *[0,1..]* essentially are a generator). [source] Haskell being a pure-FP language with lazy evaluation by default makes it especially suited for this interpretation of the idea of *code as data*.

● If you are curious about what "*a monoid in the category of endofunctors*" is, see the online book by Bartosz Milewski (2014): Category Theory for Programmers.

For a quick taste, to see whether you want to read it through, try the section on Functors. This is related to monads, too; all monads are functors. (A functor is a more general, simpler object. While at it, you may also want to read about another related topic, *applicatives*.)

> ※ In Haskell, the term *functor* means something completely different from its C++ meaning. In Python terms, a C++ "functor" is any object that defines *__call__()*.
>
> Haskell takes its terminology from category theory, which predates C++; a Haskell *functor* is basically an object we can **map** over (in the sense of the higher-order function), with some restrictions to keep it mathematically nice – see the book for details. (In Haskell, the function that maps over a functor is called **fmap**.)

● For understanding the *Reader* monad, see Martin Oldfield: Monads in Haskell: Reader.

● If you want to play around with monads in Racket instead of in Haskell, there is a library: Monads in Racket, A Dynamically-Typed Language, which also has do-notation.

  Implementing polymorphic **unit** in a dynamically typed language is nontrivial. Explanation by library author: Tony Garnock-Jones (2015): Monads in Dynamically-Typed Languages.