Designing a Program

1 Introduction

This document provides you with suggestions on how to plan a program. It gives an example and an exercise for you to complete. This will help you to design the program that you use in your project.

The text draws heavily on Magnus Hetland's excellent programming tutorial "Instant Hacking"
http://hetland.org/writing/instant-hacking.html
and on Naomi Nishimura's discussion of "pseudo-code"
(http://www.cs.cornell.edu/Courses/cs482/2003su/handouts/pseudocode.pdf).

There are really 3 parts to the process of designing your program:

- Figure out the algorithm you will use. This is best done by writing "pseudo code".
- Decide which bits are best made into separate functions (or methods).
- Decide how to store the data and variables. These are called "data structures". They may be simple arrays, but they might be much more complicated.

The order in which you do these things is not fixed. You'd usually start by writing a pseudo-code description of what your program does. This will naturally suggest some helpful functions and lead you to think about the variables that will be required. As you describe these, you might realise that there are better ways to approach the algorithm etc.

2 Why plan a program?

Most beginners simply launch in to writing their Python program, by opening the editor and then typing a few commands. This is OK for simple things (such as the revision exercises), but is a bad idea for something complex like your computing project. Before you start creating any Python code, you need to sit back and think about the problem.

Some of the advantages:

- Separate thinking about what you want the computer to do from how to code it.
- Check that your algorithm really does what you want.
- Make sure your design can be extended easily.
- Make sure your design is structured and makes good use of functions.

Trust me - these things are really important. The last thing you want to do is to write 100s of lines of code only to discover that the algorithm has a fundamental misconception, or that you can't extend you program to do the next part of the problem because of the way the data is stored.

The process of planning your program should be something like this:

- First think about the problem you want to solve. Do *you* understand what you are going to ask the computer to do?
- Sketch out the algorithm that you're going to use. By writing "pseudo-code" you can write an English description of what your program will do.
- You don't have to worry about the syntax of Python or exact command names, so it is very quick. You can easily see what your design does, and write it down without being distracted by the details.
- You can now sit back and check that your algorithm really does what you want.
- You can identify which bits of code are self-contained - these should be functions.
- Does your design repeat similar calculations several times - write a function so that the core code for the calculation is only written and debugged once. *Believe me this saves a lot of time later!*
- What variables does your program need to store? Are these arrays, matrices or just lists? How many elements will they have? Are the variables real, complex or integer? Think up sensible names.
- Will your program write numbers to a file or draw a graph?
- What are the instabilities in your algorithm? What might cause it to crash? How can you avoid this happening?
- How can you check your code works. Now is a good time to think about a test problem (or two) where you know what the answer should be.

Once you've written your pseudo-code, its easy to translate this into Python. Start at the bottom, writing the low-level functions. As you write them, check that they work. Checking as you go along will also save you lots of time.

3 Writing pseudo-code

The idea of pseudo-code is to generate a compact and informal description of your algorithm. You lay it out using code-like constructs (eg., if...then; for...) but there's no strict syntax and you can describe the actions in English. You need to strike a balance between being compact and being sufficiently precise about how the algorithm works.

Here are some guidelines (from Naomi Nishimura). There's an example of some pseudo code below:

- Mimic good code and good English. Using aspects of both systems means adhering to the style rules of both to some degree. It is still important that variable names be mnemonic, comments be included where useful, and English phrases be comprehensible (but full sentences are usually not necessary).

- Ignore unnecessary details. If you are worrying about the placement of commas, you are using too much detail. It is a good idea to use some convention to group statements (begin/end, brackets, or whatever else is clear), but you shouldn't obsess about syntax.
- Don't belabour the obvious. In many cases, the type of a variable is clear from context; unless it is critical that it is specified to be an integer or real, it is often unnecessary to make it explicit.
- Take advantage of programming shorthands. Using if-then-else or looping structures is more concise than writing out the equivalent in English; general constructs that are not peculiar to a small number of languages are good candidates for use in pseudo-code.
- Consider the context. If you are writing an algorithm to sort data, the statement "use quicksort to sort the values" is hiding too much detail; if it's just a small part of the algorithm, the statement would be appropriate to use. You can define this function later.
- Separate out self-contained (or repeated) pieces of the algorithm as functions. Specifying the input/output as well as giving the function as name.
- Don't lose sight of the underlying algorithm. It should be possible to see through your pseudo-code to the algorithm below; if not (that is, you are not easily able to check the algorithm works), it is written at too high a level.
- Check for balance. If the pseudo-code is hard for a person to read or difficult to translate into working computer code (or worse yet, both!), then something is wrong with the level of detail you have chosen to use.

4 Some Pseudo-Code Examples

4.1 Fiesta Spam Salad

I've borrowed this example from Magnus Heltand's "Instant Hacking". Imagine you need to write a computer code to make Fiesta Spam Salad. Here's the recipe (from Hormel Foods Digital Recipe Book, http://www.hormel.com/):

```
Fiesta SPAM Salad

Ingredients:

Marinade:
1/4 cup lime juice
1/4 cup low-sodium soy sauce
1/4 cup water
```

```
1 tablespoon vegetable oil
3/4 teaspoon cumin
1/2 teaspoon oregano
1/4 teaspoon hot pepper sauce
2 cloves garlic, minced

Salad:
1 (12-ounce) can SPAM Less Sodium luncheon meat,
    cut into strips
1 onion, sliced
1 bell pepper, cut in strips
Lettuce
12 cherry tomatoes, halved

Instructions:

In jar with tight-fitting lid, combine all marinade ingredients;
shake well. Place SPAM strips in plastic bag. Pour marinade
over SPAM. Seal bag; marinate 30 minutes in refrigerator.
Remove SPAM from bag; reserve 2 tablespoons marinade. Heat
reserved marinade in large skillet. Add SPAM, onion, and
green pepper. Cook 3 to 4 minutes or until SPAM is heated.
Line 4 individual salad plates with lettuce. Spoon hot salad
mixture over lettuce. Garnish with tomato halves. Serves 4.
```

The idea of pseudo-code is to re-write this so that it looks more like a computer might understand it. However, its important that we don't lose sight of what's really going on. Our Psuedo-Code description might look something like this:

```
program make_Fiesta_SPAM_salad
   Marinade the SPAM     (function)
   Cook the SPAM         (function)
   serve the SPAM salad  (function)
finished!

function Maridade_the_SPAM
      input:  marinade ingredients, SPAM
      output: marinaded_SPAM, reserved_marinade
  collect all the ingredients for the marinade.
  put them in a bag
  shake well
  cut up the spam
  put the SPAM in the bag
  leave in the refrigerator for 30 min
  return:  SPAM,  two spoons of marinade

function Cook_the_SPAM
      input:  SPAM, marinade, onion, pepper
      output: hot salad mixture
   heat reserved marinade in pan
   add spam, onion, paper
   cook for 3 to 4 min
   return:  heated salad mixture

function serve_the_spam
      input: lettuce, hot salad mixture, salad plates, tomatoes
      output:  lunch
   line plates with lettuce
   spoon over salad mix
   add tomato halves
   return: lunch
```

Notice how we were able to split things up using functions so that the basic idea is very clear. However, the analogy is wearing a bit thin now. I'll leave you to think up some data structures. Let's get on with a more relevant example.

5 A proper example — projectile motion.

Let's tackle the problem of projectile motion, for example the motion of a cannon shell. We want to plot a graph of the position of the cannon shell as a function of time. If we ignore air resistance, this is a trivial problem, but we want our program to be capable of allowing for air resistance (however, we'll assume the density of air is constant).

We can solve this problem using "Euler's Method" (see "Computational Physics" by Giordiano & Nakanishi for example). In this method, we update the position at time t_i using the velocity at time t_i to get the position at time $t_{i+1} = t_i + \Delta t$.

$$\begin{aligned} x_{i+1} &= x_i + v_{x,i}\Delta t \\ y_{i+1} &= y_i + v_{y,i}\Delta t \end{aligned} \tag{1}$$

We also need to update the velocity allowing for the acceleration due to gravity and the effect of air resistance.

$$\begin{aligned} v_{x,i+1} &= v_{x,i} + \frac{F_{\text{drag}}\cos\theta}{m}\Delta t \\ v_{y,i+1} &= v_{y,i} + \frac{F_{\text{drag}}\sin\theta}{m}\Delta t - g\Delta t \end{aligned} \tag{2}$$

where the drag force is given by $F_{\text{drag}} = -A\rho_{\text{air}}v^2$ (where A is constant and ρ_{air} is the density of air). Since v_x, v_y are the x and y components of v, we have $v_x = v\cos\theta$.

[Those of you who studied the Computational Physics module last year will be familiar with improvements to this algorithm, but now we're concentrating on implementation, so we'll stick to Euler's scheme.]

We want to start with an initial velocity and angle, then step along in time plotting the position of the shell until it strikes the ground.

In pseudo-code our algorithm looks something like this:

```
program shell_trajectory
   get initial values                  (function)
   create arrays x,y,vx,vy,t
   initialize variables vx[0] and vy[0]. Set x[0],y[0],t[0] = 0
   choose timestep  delta_t
   loop over time steps:            (Python: "for i in range(max_steps)")
      compute t[i+1], x[i+1] and y[i+1]
      compute air resistance           (function)
      compute vx[i+1], vy[i+1]
      if y < 0 exit loop
   plot graph of y vs x                (function)

function get_initial_values
      input none
      output initial values of ...
    #  read from file or from keyboard?
    set initial x,y velocity,
    set density of air, constant A, mass of shell.
    return values

function air_resistance
      input: vx, vy, constant A, density of air
      output: Fx, Fy   (x and y components of drag force)
   compute speed        #   v = sqrt(vx**2 + vy**2)
   compute angle        #   theta = arctan(vy/vx)
   compute x and y components of drag force
   return drag force Fx, Fy

function plot_graph
       input: x, y
       output: none
   open graph window
   plot x, y
   add axis titles/units
   return

Data structures
```

```
A, rho_air, vx0, vy0:   initial values.  (what units?)
delta_t, max_steps:     parameter values.
x,y,t,vx,vy:   arrays of real numbers. Dimensions 0 .. max_steps-1
```

That will do for a start. Doing this has forced me to think about exactly how I implement the algorithm, and there are some issues that need a little more work. For example,

- Am I confident that using arctan will give the correct sign for theta if vy is negative? (I'll make a note to check this).
- How am I going to read in the initial values: from the keyboard or a file. Maybe I should have a subroutine to do this so it doesn't clutter up the main code. I should check that the values are sensible.
- What units am I going to use for the calculation?
- How do I decide on a value for delta_t and max_steps? Maybe I need a subroutine to guess these based on the time of flight for the case without air resistance... or I could just ask for these as part of the input data.
- There's a dangerous bug in the example: I need the loop to go i=0 ... max_steps-2 or I'll try to assign a value to a array element x[max_steps], which doesn't exist.[1]
- If I exit the loop because $y[i] < 0$, what values get assigned to the remaining elements of x, y, vx, vy etc? I need to be careful that these are set to zero when I create the array.
- I should also think about how I might extend the code in future. For example: would it be easy to change the program to allow the density of air to vary with height? Could I run the program with several values of the constant A to show how this affects the path?

I should now revise my pseudo-code to take these points into account (best) or add them as comments (second best). I need to be careful not to add too much detail, or I'll obscure what's really going on. Expanding things as functions is a good way to go.

6 Summary

The steps you should take when you write a program are:

- Separate the overall task into a 3 or 4 parts (functions).

[1]Think carefully about how Python uses array indices and "range" statements: using "for i in range(max_steps)" I'll get "i=0,1,2...max_steps-1", which normally goes with defining the array as dimension max_steps (eg "x=N.zeros(max_steps)"). The trouble is that in this case I assign to "x[i+1]".

- Now describe how each of these functions work. This might involve adding further functions.
- Decide how the data will be stored and what units it will have.
- Sit back and look at your design. Is it clear how it works? Are there any bugs in it?
- Revise your design to make it clearer. Fix any bugs.

Now you can convert your design into a working program.

- Identify the lowest-level functions (the ones that come last in your design). These are the building-blocks of your program.
- Implement the first of these in Python.
- Check that the function works correctly. For example, you can import your program as a module and then test your function on the command line.
- Move onto the next low-level function. Code it. Test it...
- Now combine the building blocks to make higher-level functions.
- Code it. Test it.
- Finally, code your main program using the high-level functions you've created.

Note the way you work from the top-level down when designing, but from the bottom-level up when coding. It is the difference between an architect and a builder.