pylearn Documentation

Release

Jørgen Kvalsvik

Apr 04, 2018

Contents:

CHAPTER 1

Functions

One of the things people learning programming tends to struggle with is program structure, in especially as the program size increases. What starts out as a 10-line *let me see if this works* often develops into a 2000 line behemoth of fragile, deeply entngled blocks of code with lots of implicit dependencies.

Knowing when and how to break a program into smaller pieces is an art, but a few guidelines goes a long way.

There's lots of documents out there describing how to make functions, so this will primarily look into *why*, and certain subtleties that often come up.

1.1 Communication is key

To start this bit off, let's break down a simple function signature:

```
def fn(param1, param2):
```

In plain English it reads: whatever comes next is a new function named *fn*, parameterised over the two values *param1* and *param2*.

In terms of communication, two wonderful things happenes:

1. We're labeling a series of steps a new *name*
2. We're letting everyone know what this computation *depends on*

This might seem a bit fluffy, so we'll explore this a bit with some examples, but before we get into that, let's allow ourselves to be slightly dogmatic:

1. You can *always* break your program into smaller pieces
2. How something is computed is *never* interesting to your program
3. Most steps and sub computations are generalisable
4. **Never** depend on anything that isn't a parameter to your function
5. **Never** modify your input parameters

6. **Never** modify enclosing or global variables
7. Do *one thing* and **one thing only**

If you follow these guidelines, your functions are almost always *testable*. For more details on testing, see the *Testing*.

Let's look at a good function - this example is borrowed from http://composingprograms.com/pages/14-designing-functions.html

```
def pressure(v, t, n):
    """Compute the pressure in pascals of an ideal gas.

    Applies the ideal gas law: http://en.wikipedia.org/wiki/Ideal_gas_law

    v -- volume of gas, in cubic meters
    t -- absolute temperature in degrees kelvin
    n -- particles of gas
    """
    k = 1.38e-23  # Boltzmann's constant
    return n * k * t / v
```

There's a lot going on, so let's look at what this tells us:

- It computes the *pressure*
- **It depends on three parameters, and three parameters only**
  - volume
  - temperature
  - particles
- It clearly states the relationship between the input parameters (in terms of phyiscal units), through its *docstring*

When calculating pressure is a function, it's very easy to see at a glance what a line of code *is intended to do*, even though it hides *what it does*. This distinction is subtle, but important for high-quality programs.

```
df = pandas.read_csv( "gas.csv" )
for row in df.itertuples():
    gas = row['name']
    v = row['volume']
    t = row['temperature']
    n = row['particles']
    print("{} has pressure {}Pa", gas, pressure(v, t, n))
```

This reads a CSV file with the four columns *name, volume, temperature, particles*, and prints the pressure.

Why bother, it's just a simple function? Yes it is, but now we have a name for it. When understanding the *flow* of the program, `n * k * t / v` could be anything, but *pressure* computes pressure.

But I know the ideal gas law Good! But the ideal gas law isn't essential, the pressure is. Additionally, the function can hold more metadata and description, and do verification for you. Besides, someone else reading your program might not be as familiar, and `pressure(v, t, m)` communicates your intent, `n * k * t / v` doesn't.

Another cool thing is that you can quickly figure out where in your program pressure is computed, by looking for uses of this function, instead of having to read (and understand!) all the code. Imagine looking for both `n * k * t / v` and `(t*k/v)*n`

But I only need to compute pressure in this loop Most programs start out that way, until you need to compute pressure again. It's often fine to not make a function until you require this a second time, but then make sure you do. Even if it seems like more work now, it **always** saves time and work in the long run.

Besides, you might need pressure in a different *program* altogether, and now you know where to look, and you have a solid, tested implementation to use. Even better, put it in a library!

It's hard to come up with a good function name Yes it is, it's in fact *very* hard. It's not without its benefits though. Small functions with very limited scope (do one thing *only*) tend to be easier to name. Naming things also forces you to think very hard on what exactly something is, and what it tries to accomplish.

Often, when things are difficult to name, it's because it's either still not quite understood or its scope is too large, and becomes easier to name once it's split into approperiate chunks.

Furthermore, since a function is a natural barrier to the outside world, they force you to clearly state your *parameters* and *pre-conditions*, things that without good functions are completely implicit and usually very difficult to figure out. Even though some precondition seems obvious now, it usually isn't already in a week or two.

But this is just a throaway script It is until it isn't. It's not a lot of work to separate in cleaner chunks, and it makes it easier to verify things works as intended. Besides, that's no reason to be sloppy, we don't treat our other tools that way.

Note: Notice that most variables have 1-letter names, which is generally a bad idea - however, in this case the variables live in a 4-line block and are not visibe outside the loop, cleanly map to the parameters of the pressure function, and more importantly, have a describing name deriving from the CSV file column header in the `row[col]` lookup.

Let's review the example, because it turns out the CSV file didn't record *particles*, but rather the *weight*, in kilograms, because it turns out this was a shipment manifest, and they didn't care much for the amount of particles in the cargo hold. This changes our pressure function slightly.

```
def pressure(v, t, m, M):
    """Compute the pressure in pascals of an ideal gas.

    Applies the ideal gas law: http://en.wikipedia.org/wiki/Ideal_gas_law

    v -- volume of gas, in cubic meters
    t -- absolute temperature in degrees kelvin
    M -- molar mass of gas
    m -- mass of gas, in g
    """
    L = 6.022e22  # Avogrado's constant
    k = 1.38e-23  # Boltzmann's constant
    R = L * k
    return m * R * t / (M * v)
```

This actually introduces a new problem; we don't have molar mass in our database. Furthermore, it's clumsy to work with molar masses and kelvins.

```
def pressure(vol, temp, mass, gas):
    """Compute the pressure in pascals of an ideal gas.

    Applies the ideal gas law: http://en.wikipedia.org/wiki/Ideal_gas_law

    vol  -- volume of gas, in cubic meters
    temp -- absolute temperature in degrees celcius
    mass -- mass of gas, in kilograms
    gas  -- name of gas
    """

    molarmass = {
```

```
        'benzene': 78.114,
        'carbon monoxide': 28.010,
        ...
    }

    if gas not in molarmass:
        problem = 'Unknown molar weight for {}. '.format(gas)
        solution = 'Please add to the molar weight table'
        raise ValueError(problem + solution)

    m = molarmass[gas]
    t = temp - 273  # convert to kelvin
    M = mass * 1000 # convert to kg
    return ideal_gas(vol, t, m, M)
```

The function now does a lot of unit conversion for us, and even some simple error checking. Over time, these functions tend to become more sophisticated. In this function, our previous pressure function has been renamed to `ideal_gas`. Let's go back to our original loop:

```
df = pandas.read_csv( "gas.csv" )
for row in df.itertuples():
    gas = row['name']
    vol = row['volume']
    temp = row['temperature']
    mass = row['weight']
    P = pressure(vol, temp, mass, gas)
    print("{} has pressure {}Pa", gas, P)
```

Notice how similar the two blocks are. This demonstrates how effective the function was at removing the *uninteresting* details of computing pressure, and allows our program to be precise and elegant.

So what benefits to we draw from this?

Clarity The code is now clearer and communicates *intent*.

Maintainability Say you notice that `(m / M) * (t * R / v)` gives faster and more accurate results. If `pressure` is a function, you only need to fix the definition, and all callers get the benefit. While it might seem simple enough or irrelevant on a single-line statement, what if the function was 4 complicated steps? What about 10 steps?

Replacability The original function was quite easily replaced by a (better) alternative. This is a lot harder when there is no name you can look for.

Readability Smaller functions are easy to understand and easy to combine - it's easier to build larger programs with smaller functions.

1.2 The art of naming functions

TODO.

1.3 Is a function ever too small?

No.

1.4 It's only called once, should I still make it a function?

Yes.

1.5 Testing

TODO.

1.6 Glossary

Dependencies In the context of functions, *dependencies* usually refers to some *state* the program needs to be in for a computation or code block to make sense or be correct. This includes variables and their values, what directory you're in, the presence and path to certain files.

Docstring Python has a built-in engine for *self documentation* called docstrings, which a lot of python tools are aware of. They're the special triple-quoted strings that *immediately follow* a function definition, the top of a module, or a class definition. It's a good idea to always write docstrings describing the intent, behaviour and assumptions of a particular function. Often this stuff goes into comments, but comments are only visible when in the implementation file, and not available from the outside.

```
def function(args):
    """<short description>

    <long description>
    """
    <function body>
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

D