Some Tips, Tricks, and Common Errors

These are small summaries of ideas, tips, and commonly seen errors that might be helpful to those beginning Python.

Functions

Functions help us with our mental chunking: they allow us to group together statements for a high-level purpose, e.g. a function to sort a list of items, a function to make the turtle draw a spiral, or a function to compute the mean and standard deviation of some measurements.

There are two kinds of functions: fruitful, or value-returning functions, which *calculate and return a value*, and we use them because we're primarily interested in the value they'll return. Void (non-fruitful) functions are used because they *perform actions* that we want done — e.g. make a turtle draw a rectangle, or print the first thousand prime numbers. They always return `None` — a special dummy value.

Tip: `None` is not a string

Values like `None`, `True` and `False` are not strings: they are special values in Python, and are in the list of keywords we gave in chapter 2 (Variables, expressions, and statements). Keywords are special in the language: they are part of the syntax. So we cannot create our own variable or function with a name `True` — we'll get a syntax error. (Built-in functions are not privileged like keywords: we can define our own variable or function called `len`, but we'd be silly to do so!)

Along with the fruitful/void families of functions, there are two flavors of the `return` statement in Python: one that returns a useful value, and the other that returns nothing, or `None`. And if we get to the end of any function and we have not explicitly executed any `return` statement, Python automatically returns the value `None`.

Tip: Understand what the function needs to return

Perhaps nothing — some functions exists purely to perform actions rather than to calculate and return a result. But if the function should return a value, make sure all execution paths do return the value.

To make functions more useful, they are given *parameters*. So a function to make a turtle draw a square might have two parameters — one for the turtle that needs to do the drawing, and another for the size of the square. See the first example in Chapter 4 (Functions) — that function can be used with any turtle, and for any size square. So it is much more general than a function that always uses a specific turtle, say `tess` to draw a square of a specific size, say 30.

Tip: Use parameters to generalize functions

Understand which parts of the function will be hard-coded and unchangeable, and which parts should become parameters so that they can be customized by the caller of the function.

Tip: Try to relate Python functions to ideas we already know

In math, we're familiar with functions like `f(x) = 3x + 5`. We already understand that when we call the function `f(3)` we make some association between the parameter x and the argument 3. Try to draw parallels to argument passing in Python.

Quiz: Is the function `f(z) = 3z + 5` the same as function `f` above?

Problems with logic and flow of control

We often want to know if some condition holds for any item in a list, e.g. "does the list have any odd numbers?" This is a common mistake:

```
1  def any_odd(xs):  # Buggy version
2      """ Return True if there is an odd number in xs, a list of integers. """
```

```
3       for v in xs:
4           if v % 2 == 1:
5               return True
6           else:
7               return False
```

Can we spot two problems here? As soon as we execute a `return`, we'll leave the function. So the logic of saying "If I find an odd number I can return `True`" is fine. However, we cannot return `False` after only looking at one item — we can only return `False` if we've been through all the items, and none of them are odd. So line 6 should not be there, and line 7 has to be outside the loop. To find the second problem above, consider what happens if you call this function with an argument that is an empty list. Here is a corrected version:

```
1  def any_odd(xs):
2      """ Return True if there is an odd number in xs, a list of integers. """
3      for v in xs:
4          if v % 2 == 1:
5              return True
6      return False
```

This "eureka", or "short-circuit" style of returning from a function as soon as we are certain what the outcome will be was first seen in Section 8.10, in the chapter on strings.

It is preferred over this one, which also works correctly:

```
1  def any_odd(xs):
2      """ Return True if there is an odd number in xs, a list of integers. """
3      count = 0
4      for v in xs:
5          if v % 2 == 1:
6              count += 1    # Count the odd numbers
7      if count > 0:
8          return True
9      else:
10         return False
```

The performance disadvantage of this one is that it traverses the whole list, even if it knows the outcome very early on.

Tip: Think about the return conditions of the function

Do I need to look at all elements in all cases? Can I shortcut and take an early exit? Under what conditions? When will I have to examine all the items in the list?

The code in lines 7–10 can also be tightened up. The expression `count > 0` evaluates to a Boolean value, either `True` or `False`. The value can be used directly in the `return` statement. So we could cut out that code and simply have the following:

```
1  def any_odd(xs):
2      """ Return True if there is an odd number in xs, a list of integers. """
3      count = 0
4      for v in xs:
5          if v % 2 == 1:
6              count += 1   # Count the odd numbers
7      return count > 0   # Aha! a programmer who understands that Boolean
8                         #   expressions are not just used in if statements!
```

Although this code is tighter, it is not as nice as the one that did the short-circuit return as soon as the first odd number was found.

Tip: Generalize your use of Booleans

Mature programmers won't write `if is_prime(n) == True:` when they could say instead `if is_prime(n):` Think more generally about Boolean values, not just in the context of `if` or `while` statements. Like arithmetic expressions, they have their own set of operators (`and`, `or`, `not`) and values (`True`, `False`) and can be assigned to variables, put into lists, etc. A good resource for improving your use of Booleans is http://en.wikibooks.org/wiki/Non-Programmer%27s_Tutorial_for_Python_3/Boolean_Expressions

Exercise time:

- How would we adapt this to make another function which returns `True` if *all* the numbers are odd? Can you still use a short-circuit style?
- How would we adapt it to return `True` if at least three of the numbers are odd? Short-circuit the traversal when the third odd number is found — don't traverse the whole list unless we have to.

Local variables

Functions are called, or activated, and while they're busy they create their own stack frame which holds local variables. A local variable is one that belongs to the current activation. As soon as the function returns (whether from an explicit return statement or because Python reached the last statement), the stack frame and its local variables are all destroyed. The important consequence of this is that a function cannot use its own variables to remember any kind of state between different activations. It cannot count how many times it has been called, or remember to switch colors between red and blue UNLESS it makes use of variables that are global. Global variables will survive even after our function has exited, so they are the correct way to maintain information between calls.

```
1  sz = 2
2  def h2():
3      """ Draw the next step of a spiral on each call. """
4      global sz
5      tess.turn(42)
6      tess.forward(sz)
7      sz += 1
```

This fragment assumes our turtle is `tess`. Each time we call `h2()` it turns, draws, and increases the global variable `sz`. Python always assumes that an assignment to a variable (as in line 7) means that we want a new local variable, unless we've provided a `global` declaration (on line 4). So leaving out the global declaration means this does not work.

Tip: Local variables do not survive when you exit the function

Use a Python visualizer like the one at http://netserv.ict.ru.ac.za/python3_viz to build a strong understanding of function calls, stack frames, local variables, and function returns.

Tip: Assignment in a function creates a local variable

Any assignment to a variable within a function means Python will make a local variable, unless we override with `global`.

Event handler functions

Our chapter on event handling showed three different kinds of events that we could handle. They each have their own subtle points that can trip us up.

- Event handlers are void functions — they don't return any values.
- They're automatically called by the Python interpreter in response to an event, so we don't get to see the code that calls them.
- A mouse-click event passes two coordinate arguments to its handler, so when we write this handler we have to provide for two parameters (usually named `x` and `y`). This is how the handler knows where the mouse click occurred.
- A keypress event handler has to be bound to the key it responds to. There is a messy extra step when using keypresses: we have to remember to issue a `wn.listen()` before our program will receive any keypresses. But if the user presses the key 10 times, the handler will be called ten times.

- Using a timer to create a future-dated event only causes one call to the handler. If we want repeated periodic handler activations, then from within the handler we call `wn.ontimer(....)` to set up the next event.

String handling

There are only four *really* important operations on strings, and we'll be able to do just about anything. There are many more nice-to-have methods (we'll call them sugar coating) that can make life easier, but if we can work with the basic four operations smoothly, we'll have a great grounding.

- len(str) finds the length of a string.
- str[i] the subscript operation extracts the i'th character of the string, as a new string.
- str[i:j] the slice operation extracts a substring out of a string.
- str.find(target) returns the index where target occurs within the string, or -1 if it is not found.

So if we need to know if "snake" occurs as a substring within `s`, we could write

```
1  if s.find("snake") >= 0:  ...
2  if "snake" in s: ...           # Also works, nice-to-know sugar coating!
```

It would be wrong to split the string into words unless we were asked whether the *word* "snake" occurred in the string.

Suppose we're asked to read some lines of data and find function definitions, e.g.: `def some_function_name(x, y):`, and we are further asked to isolate and work with the name of the function. (Let's say, print it.)

```
1  s = "..."                        # Get the next line from somewhere
2  def_pos = s.find("def ")         # Look for "def " in the line
3  if def_pos == 0:                 # If it occurs at the left margin
4    op_index = s.find("(")         # Find the index of the open parenthesis
5    fnname = s[4:op_index]         # Slice out the function name
6    print(fnname)                  # ... and work with it.
```

One can extend these ideas:

- What if the function def was indented, and didn't start at column 0? The code would need a bit of adjustment, and we'd probably want to be sure that all the characters in front of the `def_pos` position were spaces. We would not want to do the wrong thing on data like this: `# I def initely like Python!`
- We've assumed on line 3 that we will find an open parenthesis. It may need to be checked that we did!
- We have also assumed that there was exactly one space between the keyword `def` and the start of the function name. It will not work nicely for `def  f(x)`

As we've already mentioned, there are many more "sugar-coated" methods that let us work more easily with strings. There is an `rfind` method, like `find`, that searches from the end of the string backwards. It is useful if we want to find the last occurrence of something. The `lower` and `upper` methods can do case conversion. And the `split` method is great for breaking a string into a list of words, or into a list of lines. We've also made extensive use in this book of the `format` method. In fact, if we want to practice reading the Python documentation and learning some new methods on our own, the string methods are an excellent resource.

Exercises:

- Suppose any line of text can contain at most one url that starts with "http://" and ends at the next space in the line. Write a fragment of code to extract and print the full url if it is present. (Hint: read the documentation for `find`. It takes some extra arguments, so you can set a starting point from which it will search.)
- Suppose a string contains at most one substring "< ... >". Write a fragment of code to extract and print the portion of the string between the angle brackets.

Looping and lists

Computers are useful because they can repeat computation, accurately and fast. So loops are going to be a central feature of almost all programs you encounter.

Tip: Don't create unnecessary lists

Lists are useful if you need to keep data for later computation. But if you don't need lists, it is probably better not to generate them.

Here are two functions that both generate ten million random numbers, and return the sum of the numbers. They both work.

```
import random
joe = random.Random()

def sum1():
    """ Build a list of random numbers, then sum them """
    xs = []
    for i in range(10000000):
        num = joe.randrange(1000)  # Generate one random number
        xs.append(num)             # Save it in our list

    tot = sum(xs)
    return tot

def sum2():
    """ Sum the random numbers as we generate them """
    tot = 0
    for i in range(10000000):
        num = joe.randrange(1000)
        tot += num
    return tot

print(sum1())
print(sum2())
```

What reasons are there for preferring the second version here? (Hint: open a tool like the Performance Monitor on your computer, and watch the memory usage. How big can you make the list before you get a fatal memory error in `sum1`?)

In a similar way, when working with files, we often have an option to read the whole file contents into a single string, or we can read one line at a time and process each line as we read it. Line-at-a-time is the more traditional and perhaps safer way to do things — you'll be able to work comfortably no matter how large the file is. (And, of course, this mode of processing the files was essential in the old days when computer memories were much smaller.) But you may find whole-file-at-once is sometimes more convenient!