6.189 IAP 2011 –Common Python Mistakes

We’ve been seeing a bunch of mistakes or misconceptions so this handout should clear some things out for you and serve as a reference.

- Variables. Remember that a *variable* is a placeholder for a value that you (or someone calling your function) can assign, and it can be any type - string, list, int, float, dictionary, tuple. Variables are exactly like variables you learned about in calculus, although they can be more than numbers. So, `rock` is a variable, called `rock`, that we can assign any value - ie `rock = 8` or `rock = 'paper'`. However, `'rock'` is a *string* - we cannot assign it a value, it is already the string value of `'rock'`. If this distinction is still confusing to you, please visit office hours for clarification.

- The `in` keyword: Checks if some single item is in a larger collection. Returns True if the item is in the list, and False otherwise. Can be negated with the keyword `not`.

```python
>>> some_list = [1, 3, 6, 7]
>>> 3 in some_list
True
>>> 5 in some_list
False
>>> 5 not in some_list
True
>>> [3] in some_list
False
```

Take special note of this last example. While the single item 3 is in `some_list`, the list `[3]` is not.

- Boolean types. Remember that `True` and `False` are Boolean types, but `'True'`, `'True'`, `'False'` and `'False'` are all strings. Boolean types are really important because they enable us to do special things within if statements and while loops - so be sure to return a Boolean, not a string, when writing functions that ask you to return a Boolean.

- Print versus return. When you’re calling a function, you can print things wherever you want; the print statement functions as a handy debugging tool, even. However, you can only return one thing. This means that as soon as your code hits a return statement, the function will exit. Keep this in mind - you never want to return too early. Also, INDENTATION IS REALLY, REALLY IMPORTANT!

For example, can you find the error in the following code that should return True if a number is prime, and return False if it is not?

```python
def is_prime(number):
    for divisor in range(2, number):
        if (number % divisor) != 0:
            return True
    return False
```

In this case, there is a big error. It seems to work at first:

```python
>>> is_prime(7)
True
>>> is_prime(10)
False
```
But look at the test case

```python
>>> is_prime(9)
True
```

Hmmmm.... what’s wrong? Ultimately, the problem is that I return True in the wrong place. So, I try again.

```python
def is_prime(number):
    for divisor in range(2, number):
        if (number % divisor) == 0:
            return False
    return True
```

There is another big problem here - I return True too early (when? for what tests cases? why is that?). Finally, one last try:

```python
def is_prime(number):
    for divisor in range(2, number):
        if (number % divisor) == 0:
            return False
    return True
```

Ah, this time I’ve got it. See how I have returned True and False, but only when I know the number is or is not prime. I don’t want to return too early, because then I might get false positives or negatives.

- **is** versus **==**: == asks if two values are equal - ie, if they are interpreted the same way. However, is asks if two values are the **exact same object**, which often gives unexpected results. If you’re unsure, use == (the same applies to **not** is versus !==). Example:

```python
>>> a = [1,2]
>>> b = [1,2]
>>> c = a
>>> a is b
False
>>> a is c
True
>>> a == b
True
```

- Integer division. Remember that one of your arguments to division should be a float if you want to make a fractional quantity - do this by including a decimal or casting one of your arguments to a float.

```python
>>> x = 7
>>> print 1/x
0
>>> print 1./x
0.14285714285714285
>>> print 1/float(x)
0.14285714285714285
```

- **Debugging.** We’ve seen a lot of code that has too many return statements and students are very, very confused at the results they’re seeing. We suggest using Rubber Duck Debugging (http://en.wikipedia.org/wiki/Rubber_duck_debugging) - a silly concept that works. Explain your code
to a rubber duck, or a teddy bear, or your coffee cup - the idea is, that by explaining what your written code actually does (as opposed to what you want it to do) will help your spot your errors. Be sure to comment your code as you go; if, when you’re explaining your code, you reach a section that is particularly confusing, you’ve found a really great place for a comment!!

• Defining functions and reusing them. It is okay - in fact, it is completely necessary sometimes - to define multiple functions, and call them inside another function. Example:

```python
VOWELS = ['a', 'e', 'i', 'o', 'u']

def is_a_vowel(c):
    # check if c is a vowel
    lowercase_c = c.lower()
    if lowercase_c in VOWELS:
        # Return (BOOLEAN!) True if c is a vowel
        return True
    else:
        # c must not be a vowel; return (BOOLEAN!) False
        return False

def only_vowels(phrase):
    # Takes a phrase, and returns a string of all the vowels
    # Initialize an empty string to hold all of the vowels
    vowel_string = ''
    for letter in phrase:
        # check if each letter is a vowel
        if is_a_vowel(letter):
            # If it’s a vowel, we append the letter to the vowel string
            vowel_string += letter
        # if not a vowel, we don’t care about it- so do nothing!
    return vowel_string

# Code after a "return" doesn’t print
print "A line of code after the return!"
```

Note in this example how we

- Comment well, explaining what every line does!
- Return a Boolean type in the function `is_a_vowel` so that we can use that function within an if conditional in the `only_vowels` code.
- Return only when we know what the result is
- Show that any code after a return statement won’t be shown, because functions exit upon hitting a return.

See our solutions to Hangman if you didn’t really get this in the last project.