G54CCS Labs, Exercise 2

In last week's exercise you created a Google App Engine (GAE) application that displayed a static web page. In this exercise you will extend this app to change the contents of that web page on each reload, and in passing, to introduce some more simple Python syntax.

Imports

First we see, as with the previous exercise, some *module imports*. These allow you to access extra functionality than is provided by the core Python language, which is quite simple and can only do relatively very basic operations.

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app

import logging, random
```

In this case, we import two GAE specific modules, and then we import the standard `logging` and `random` modules to enable simple logging of output and generation of random numbers.

Constant String

Next we have the HTML boilerplate for the page we will return:

```
HTML = """\
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Some Python</title>
    <meta charset='utf-8'>
  </head>

  <body>
    %s
  </body>
</html>"""
```

This is a `string`, that is, `HTML` is a *variable* — a label representing some contents which can change — that currently contains data which represents some text. Note the `%s` toward the end — this will allow us to insert extra contents into the HTML boilerplate at that point. String in Python can be enclosed in single (')

or double (") quotes, but not a mixture. Using triple single- or double-quotes is a verbatim string, i.e., one which includes all whitespace, newlines and so on until the appropriate closing triple.

Boilerplate

Next we have the definition of the class and its method used to handle incoming `GET` requests to the `/invoke` URL, followed by the simple logging statement as in the first exercise. Classes and methods are more advanced programming concepts which we will not discuss further here.

```
class Invoke(webapp.RequestHandler):
    def get(self):
        logging.info("request: %s" % (self.request,))
```

... followed by the meat of the programme, followed by the closing boilerplate:

```
        self.response.out.write(HTML % result)

urls = [
    (r'/invoke', Invoke),
    ]

application = webapp.WSGIApplication(urls, debug=True)
def main():
    run_wsgi_app(application)
if __name__ == "__main__":
    main()
```

We will now consider the meat of the programme between these two blocks.

Function, Integers, Floats, Strings, Arithmetic

```
        x = random.randint(0, 16)
        y = random.randint(16, 30)

        z = x/y
        f = float(x)/y

        result = "<p>x=%d y=%d z=%d f=%0.3f</p>" % (x, y, z, f)
```

Taking each line in turn, this does the following:

- Generates a random *integer* between 0 and 10, and assigns it to the variable `x`;
- Generates a random integer between 10 and 20, and assigns it to the variable `y`;
- Computes `x` divided by `y` and assigns the result to the variable `z`;
- Converts the integer `x` to a *floating point value* and divides it by `y` and assigns the result to the variable `f`; and finally
- Constructs a string, `result`, by a process which Python calls *string interpolation*, which contains the values of the *tuple* `x`, `y`, `z`, and `f` within an HTML paragraph (delimited by `<p>...</p>`).

Function calls in Python mean putting the name of the module from which the function comes if required (`random` in this case), followed by the separator (`.`) and then the name of the function (`randint`) followed by parentheses containing any parameters to the function (`(0, 16)` and `(16, 30)` in this case).

Look closely at the difference between `z` and `f`: `z` is always 0 because you cannot represent fractional (or irrational) numbers as a variable of integer type. By converting `x` to a floating point value first, we allow non-integer values to be represented in the computation `float(x)/y` and so `f` is also a floating point value.

The basic arithmetic operations are `+`, `-`, `*` (multiply) and `/` (divide). Since performing an operation on a variable and storing the result back in the same variable is such a common operation, Python provides shortcuts for this: `+=`, `+-`, etc. For example, `x += 1` is identical to `x = x+1`, i.e., it computes `x+1` and stores the result back into `x`. To test for equality, use `==` — `=` is used only to assign a value to a variable.

Also, look closely at the string interpolation: the substrings beginning `%` between the quotes "..." are *format specifiers* — placeholders which are filled in with strings representing the values given in the tuple. The particular format specifiers used here are `%d` which produces the string representing an integer as a decimal; and `%0.3f` which takes a floating point value and represents it as a decimal string to 3 decimal places.

Strings

```
s = "x"*10 + "y" + ":z"*5
ss = s.split(":")
result += "<p>%s -> '%s'</p>" % (s, ss)
```

Python has a wide range of string manipulation functions and methods. This demonstrates some simple ones:

- Multiplying a string by a number repeats the string the specified number of times;
- Adding two strings together concatenates them; and
- Invoking a string's `split` method creates a list of strings by dividing the original string where it contains any character in the parameter (in this case, splits the string `s` on every `:` character).

Lists

```
lst = [x, y]
lst.append(x)
result = "%s<p>lst='%s'</p>" % (result, lst)
result += "<p>elements:"
for e in lst: result += "%d," % e
result += "</p>"
```

You have already seen *tuples* — sequences of variables enclosed in parentheses `(...)` — which are *immutable*, i.e., they cannot be changed after being defined. The equivalent *mutable* type is the *list*, a sequence enclosed in square brackets `[...]`. Lists support a number of operations to sort, separate, append, extend and so on.

Note one common problem with tuples is the following: *how do you represent a tuple with just a single element*? The obvious way to do this — `(x)` — is unfortunately wrong: it is parsed as the arithmetic expression returning `x`, i.e., `(x) == x`. If you need to represent a tuple with a single element, simply insert a trailing comma, i.e., `(x,)`.

Dictionaries

```
d = { "0-10": x,
      "10-20": y,
      "zero": z,
      "string": s,
      }
result += "<p>%s</p>" % d
```

Dictionaries known as *maps* associate *values* with *keys*. That is, you insert a value associated with a key, and you can then subsequently get the value back out if you have the right key. (By analogy with a dictionary where, if you know the word, you can retrieve the associated definition.)

In Python, dictionaries are very flexible and widely used. A single dictionary can simultaneously contain keys and values of many different types. The main

restriction is that keys must be *hashable*, but all the basic types meet this restriction so we will not discuss it further here.

Iteration

```
for k in d:
    result += "<p>key='%s' value='%s'</p>" % (k, d[k])
```

Several types in Python are *iterable*, that is, you can *iterate* over them. Iteration refers to the process of consider each element of a variable in turn. So, for example, if you wish to consider every element of a list, you *iterate* through the list. The Python syntax for this is very simple:

`for <variable> in <iterable>: do-stuff-with-variable`.

Iteration across a list or tuple considers each element in order; iteration across a dictionary considers each key in an arbitrary order (in fact, in the order defined by the hash value of the key).

In the example above, we iterate through the dictionary `d` referring to each key as `k`. For each key, we concatenate a paragraph containing the key and the corresponding value to the result string.

Exercises

Ex.1: Try changing the `%d`s to `%x`s and see what happens. Try some format specifiers from `http://docs.python.org/release/2.6.7/library/stdtypes.html#string-formatting-operations`.

Ex.2: Investigate some of the other built-in string manipulation functions available in Python, documented at `http://docs.python.org/release/2.6.7/library/stdtypes.html#string-methods`.

Ex.3: Investigate the builtin list functions in Python. Modify your programme to print `lst` both unsorted and sorted.

Ex.4: Consider the following function definition:

```
def p(s):
    return "<p>%s</p>" % s
```

Add this to the application after the HTML boilerplate string, and then make use of it to simplify and reduce repetition in your code.

Ex.5: Explore the builtin methods available to both lists and dictionaries.