PIC 16, Winter 2018

Lecture 2F: Functional Programming

Friday, January 19, 2018

Matt Haberland

Announcements

- Assignment 1F solution posted
- Remember to check out Preparation Notebooks

Activity

- Work on Assignment 2W,
- Start Assignment 2F, or

Write functions to reproduce the functionality of Python's built-in functional programming tools:

- `my_filter` – accepts a function and a list, return a list
- `my_map` – accept a function and a list, return a list
- `my_reduce` – accept a function and a list, return a single value
- `my_enumerate` – accept a list, return a list of
- `my_zip` – accept two lists, return a list of tuples
- `my_reversed` – accept a list, return the list in reverse
- `my_iteritems` – accept a dict, return a list of tuples

slightly different than corresponding built-in function

If you wish, write the functions using `for` loops initially, then try converting to list comprehensions.

Test your functions alongside the corresponding built-in functions. For example,

```
for i, el in enumerate(['a','b','c']):
    print i, el
for i, el in my_enumerate(['a','b','c']):
    print i, el
```

Questions

Intended Learning Outcomes

- By the end of today, I want you all to be able to:
  - choose between equality and identity comparison, and
  - use the `in` operator to check membership, and
  - determine whether `time.time` or `time.clock` has better precision on a given platform.

Identity vs Equality

- Evaluate:
  - `1 == 1` — `True`
  - `x = 1; y = x; x == y` — `True`
  - `x = 1; y = x; x is y` — `True`
  - `[1] == [1]` — `True`
  - `[1] + [2] == [1, 2]` — `True`
  - `[1] + [2] is [1, 2]` — `False`
  - `[1] is [1]` — `False`

Identity vs Equality

- Evaluate:
  - `1 is 1` — `True`
  - `10 * 10 is 100` — `True`
  - `chr(97) is 'a'` — `True`
  - `'a' + 'b' is 'ab'` — `True`
  - `x = '', y = 'a', x + y is 'a'` — `True`
  - `x = 1, y = 2, x + y is 3` — `True`
  - `x = 'a', y = 'b', x + y is 'ab'` — `False`
  - `10 * 100 is 1000` — `False`

These are the exception, not the rule.

Why do some of these work when they “shouldn’t”?
Python does some optimization under the hood when it can.
There’s no *harm* in re-using an existing immutable object.
Still, don’t rely on exceptions!
Use `==` to check whether two objects are *equivalent*.
Use `is` only to check whether two references refer to the *exact same object*.

Membership

- Considering:
  - `[1] + [2] is [1,2]` **`False`**
- What about:
  - `[1] + [2] in [[1,2], [3,4]]` **`True`**
- This implies that equality testing, not identity testing, is used to determine membership

- Considering:
  - a = range(10000000); b = set(a)
- Which is faster?
  - 5000000 in a
  - 5000000 in b

Try it! Use `%timeit`

`time.time` vs `time.clock`

- How can you determine which has better precision on your machine?

Questions?

Quiz 2F

- Please *do not* refer to the preparation document or other materials. Answers are to come from your brain only.

Activity

- Write a function `my_timer` that:
  - accepts a function `f` (which accepts no arguments) and
  - returns the time it takes to run `f`
- How can you use it to test a function that requires arguments?
- Make it execute `f` a prescribed number of times and return minimum, maximum, and average execution time
- Can you make it automatically determine the executions to try?
- Consider this: have `my_timer` accept a second argument `g`, a function that randomly generates a tuple of input arguments for `f`, and have `my_timer` pass a different set of arguments into `f` each execution

`a or b`

I think you should think about the `or` operator working differently on strings than it does on booleans. The strings are not treated as booleans; something different is going on.

You can think of `a or b` when both `a` and `b` are strings as:
If `a == ''` and `b == ''`, evaluate to `''`
If `a != ''` and `b == ''`, evaluate to `a`
If `a == ''` and `b != ''`, evaluate to `b`
If `a != ''` and `b != ''`, evaluate to `a`

In other words, evaluate to the first one that is not the empty string. If both are empty, evaluate to the empty string.

`a or b or c`

And apparently, whether `a` or `b` are strings or not:
`a or b or c` first does `a or b`, then `or`s the result with `c`.
That is, it does:
`(a or b) or c`
It does not do:
`(a or b) and (b or c)`
as you might think it would based on the rules of `a == b == c`.

a and b

Incidentally, `a and b` does the opposite of opposite of `a or b` when both are strings (unless `a == '' and b = ''`):

If `a == '' and b == ''`, evaluate to `''`

If `a != '' and b == ''`, evaluate to `b`

If `a == '' and b != ''`, evaluate to `a`

If `a != '' and b != ''`, evaluate to `b`