3110: Loose Ends

Wrapping up Functional Programming

Logistics

- First quiz graded – did very well
- First project out – due Thurs, 9:30am
- Homework out – due Wed 9:30am
- Following Spring semester, have a few loose ends

Quiz Solutions (1)

- Python is:
  - Lexical Scope
  - Eager Evaluator (for the most part, details later)
  - Application-order Evaluator

- True, a function with no side effects will not affect outside variables

Quiz Solutions (2)

- Can not change what an entry in a tuple points to. (produces an error)
- Can however change a list stored within a tuple.
- Board work on pointers

Quiz Solutions (3)

```
return (not(a_list[1:]) and a_list) or
        (a_list[0] < a_list[-1] and BoolMin(Join(a_list[1:-1], [a_list[0]]))) or
        (BoolMin(a_list[1:]))
```

Based on:

```
def min(a_list):
    if not(a_list[1:]):
         return a_list
    elif a_list[0] < lowest;
        return min(Join(a_list[1:-1], [a_list[0]])
    else:
        return min(a_list[1;])
```

Quiz Solutions (3)

- Elegant solution

```
return (not(a_list[1:]) and a_list) or
        (a_list[0] < a_list[-1] and BoolMin(a_list[:-1])) or
        (BoolMin(a_list[1:]))
```

Quiz Solutions (4)

- On board

Quiz Solutions (5)

```
sorted(points, key= lambda p: sqrt(p[0]**2 + p[1]**2))
```

Fill in the holes

- Keywords
- Generators
- Exceptions
- Tail Recursion
- Map-Reduce
- Type and Pattern Matching Type
- Review

Keywords

- Optional arguments:

fun(var_1, var_2, …, keyword_1=default_1, …)

- Don’t have to provide values for keywords, fun will just use the default values

Generators

Python's lazy tooth:

- [fun(a) for a in seq], generates the entire list

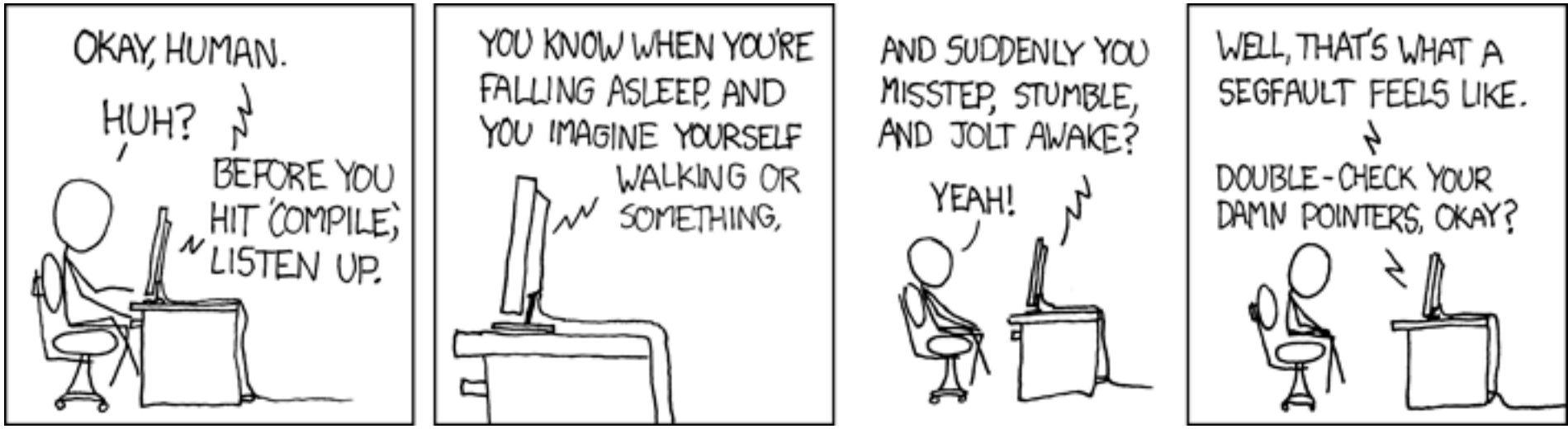
- (fun(a) for a in seq), produces a generator that acts lazy. Only computes fun(a) when asked .next(), and at that only computes fun(a) for the next element in the seq.
- Note, evaluation of fun(a) may or may not in return be lazy
- Allows for handling of possibly infinite sequences.

Exceptions

Format:

```
try:
    code …
except TypeError:
    code …
else:
    code …
```

- Can use instead of (if/else)
- In the case of adding Dictionary keys, slower than checking if key exists
- Most Common use is to check input/results
- But mostly:



Shouldn’t be part of your algorithm’s logic.

Tail Recursion

- when a recursive call ends with calling itself(without surrounding computations) the compiler can optimize by shortening the stack

```
def Fact_1(n):
    if n == 0:
        return 1
    else:
        return n*Fact(n-1)
```

```
def Fact_2(n):

    def Fact_TR(product, n):
        if n == 0:
            return product
        else:
            return Fact(n*product, n-1)

    return Fact_TR(1, n)
```

Board: draw stacks

Ah, but Python

- Python does not do Trail Recursion. Python has a limit of 1000 stack depth. Will cause error.
- Python users want stacks to be easy to read.

- can manually readjust, but doing so reportable causes nasty crashes for Python.

Tail Recursion in Java (works)

```
int factorial(int number)  {
     if(number == 0)  {
            return 1;
    }
    factorial_i(number, 1);
}

int factorial_i(int currentNumber, int sum)  {
    if(currentNumber == 1) {
            return sum;
    } else {
            return factorial_i(currentNumber - 1,
            sum*currentNumber);
    }
 }
```

Map-Reduce

Map:

- returns new list of function applied in parallel to each input element

Reduce:

```
reduce(function, seq)
```

- Has optional keyword init, see api, sets how reduce handles empty sequences

Map-Reduce

- We will come back to coding map-reduce's in concurrency and threading.

- Highlights:

  well-defined independent operations -> lends itself to scalable parallelism.

Types

- We have been ignoring types, going with instinct and letting Python catch us.
- May want to differentiate input based on type.
- Python has built in types, try:

  type(3) = <type ‘int’>

  type(‘hey’) = <type ‘str’>
- Built in types (float, list, tuple, dict, etc)

Checking Types

- type(obj) == type, type
  - Checks only for exact equivalence

- isinstance(obj, type), where type is a tuple of types we are checking obj against.
  - Automatically detects subclasses.

Creating Types

- Types tell you what methods can be applied
- Classes tell you what methods can be applied
- In Python 2.2 and later, checking Class type
  - isinstance(instance, class) works!

Pattern Matching Types

- Be nice if we could detect, whether or not an object can be matched to a more specific type. Ie (int, int) or (int, int, int)
- Easy to do in Haskell and Ocaml, bit more work in Python (see homework)

Nice Review/Higher Level FP

- Adam Byrtek