HW 3 - Group 8 - Homework 3: AWK, Lua, First-Class Functions

Group Members –

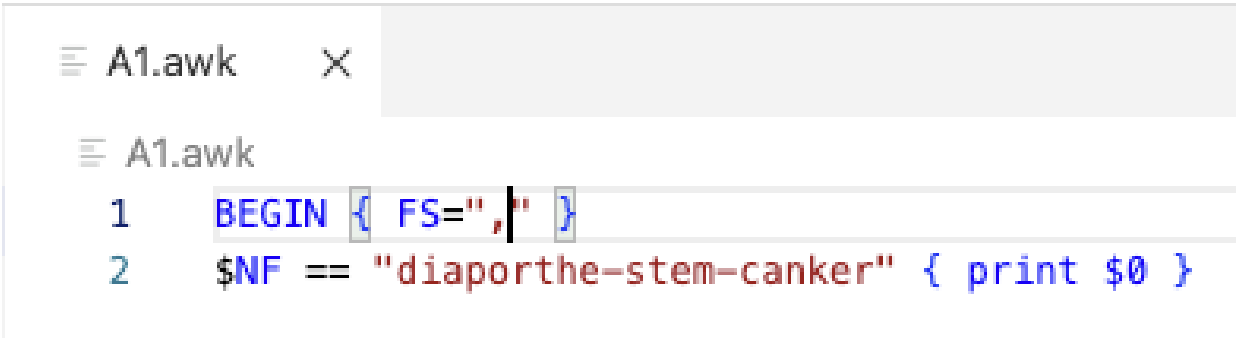
Aniruddha Shivananda (ashivan@ncsu.edu)

JonCarlo Migaly (jmigaly@ncsu.edu)

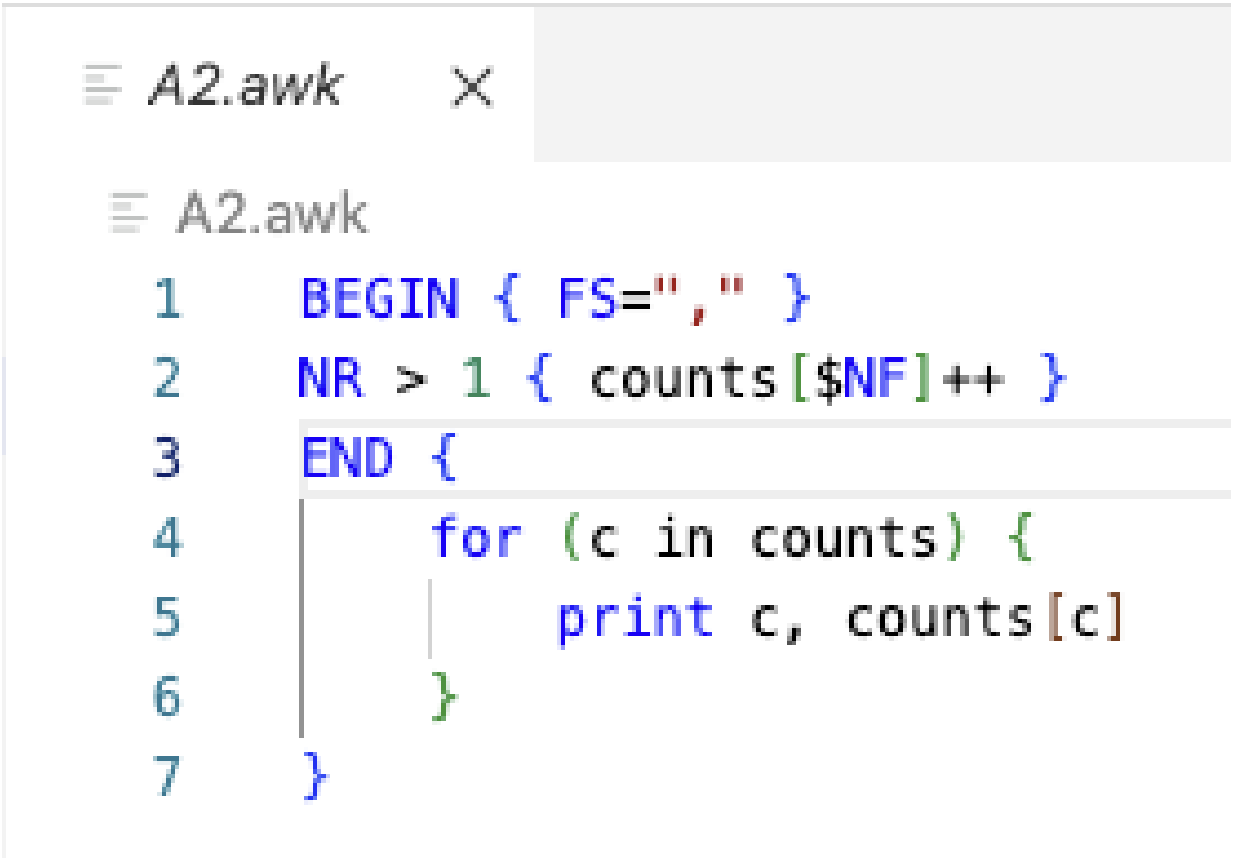
Patrick Parker (pparker2@ncsu.edu)

Github - https://github.com/AShivan26/How-to-be-a-SE-Guru

A1.awk


A1.awk
A1.awk
1 BEGIN { FS="," }
2 $NF == "diaporthe-stem-canker" { print $0 }

A2.awk


A2.awk
A2.awk
1 BEGIN { FS="," }
2 NR > 1 { counts[$NF]++ }
3 END {
4 for (c in counts) {
5 print c, counts[c]
6 }
7 }

A3.awk

A3.awk

```
BEGIN { FS="," }
NR > 1 { counts[$2]++ }
END {
    max_count = 0
    most_common = ""
    for (val in counts) {
        if (counts[val] > max_count) {
            max_count = counts[val]
            most_common = val
        }
    }
    print most_common, max_count
}
```

A4.awk

A4.awk

```
BEGIN { FS="," }
NR > 1 {
    matrix[$NF, $1]++
    classes[$NF] = 1
    col1_vals[$1] = 1
}
END {
    for (c in classes) {
        for (v in col1_vals) {
            if (matrix[c, v] > 0) {
                print c, v, matrix[c, v]
            }
        }
    }
}
```

A5.awk

A5.awk

```awk
BEGIN { FS="," }
NR <= 10 { print; next }
$3 != "?" { print }
```

A6.awk

A6.awk

```awk
BEGIN { FS="," }
NR > 1 { counts[$NF]++ }
END {
    entropy(counts, NR-1)
}

function entropy(arr, n,    i, p, e) {
    e = 0
    for (i in arr) {
        p = arr[i] / n
        e -= p * log(p)
    }
    print e
}
```

A7.awk

A7.awk

```awk
BEGIN { FS=","; srand() }
NR > 1 {
    if (NR - 1 <= 20) {
        reservoir[NR - 1] = $0
    } else {
        r = int(rand() * (NR - 1)) + 1
        if (r <= 20) {
            reservoir[r] = $0
        }
    }
}
END {
    for (i in reservoir) {
        print reservoir[i]
    }
}
```

A8.awk

A8.awk

```awk
BEGIN { FS=","; Total=0; Correct=0; Tested=0 }
NR==1 { next }
NR<=11 { train(); next }
{
  c=classify()
  if (c == $NF) Correct++
  Tested++
  train()
}
END {
    print "Accuracy: " (Correct/Tested) * 100 "%"
}

function train(    i,c) {
  Total++; c=$NF; Classes[c]++
  for(i=1; i<NF; i++) {
    if($i=="?") continue
    Freq[c,i,$i]++
    if(++Seen[i,$i]==1) Attr[i]++
  }
}

function classify(    i,c,t,best,bestc) {
  best=-1e30
  for(c in Classes) {
    t=log(Classes[c]/Total)
    for(i=1; i<NF; i++) {
      if($i=="?") continue
      t+=log((Freq[c,i,$i]+1)/(Classes[c]+Attr[i]))
    }
    if(t>best) { best=t; bestc=c }
  }
  return bestc
}
```

A9.awk

A9.awk

```awk
BEGIN {
    FS=","
    Total=0
    Correct=0
    Tested=0
    if (wait == "") wait=10
}
NR==1 { next }
NR <= wait + 1 { train(); next }
{
  c=classify()
  if (c == $NF) Correct++
  Tested++
  train()
}
END {
    print "Accuracy: " (Correct/Tested) * 100 "%"
}

function train(    i,c) {
  Total++; c=$NF; Classes[c]++
  for(i=1; i<NF; i++) {
    if($i=="?") continue
    Freq[c,i,$i]++
    if(++Seen[i,$i]==1) Attr[i]++
  }
}

function classify(    i,c,t,best,bestc) {
  best=-1e30
  for(c in Classes) {
    t=log(Classes[c]/Total)
    for(i=1; i<NF; i++) {
      if($i=="?") continue
      t+=log((Freq[c,i,$i]+1)/(Classes[c]+Attr[i]))
    }
    if(t>best) { best=t; bestc=c }
  }
  return bestc
}
```

A10.awk

A10.awk

```awk
BEGIN {
    FS=","
    Total=0
    Correct=0
    Tested=0
    NumClasses=0
    if (k == "") k=1
    if (m == "") m=2
}
NR==1 { next }
NR<=11 { train(); next }
{
  c=classify()
  if (c == $NF) Correct++
  Tested++
  train()
}
END {
    print "Accuracy: " (Correct/Tested) * 100 "%"
}

function train(    i,c) {
  Total++; c=$NF
  if (Classes[c]++ == 0) NumClasses++

  for(i=1; i<NF; i++) {
    if($i=="?") continue
    Freq[c,i,$i]++
    if(++Seen[i,$i]==1) Attr[i]++
  }
}

function classify(    i,c,t,best,bestc) {
  best=-1e30
  for(c in Classes) {
    t=log((Classes[c] + m)/(Total + m * NumClasses))

    for(i=1; i<NF; i++) {
      if($i=="?") continue
      t+=log((Freq[c,i,$i] + k)/(Classes[c] + k * Attr[i]))
    }
    if(t>best) { best=t; bestc=c }
  }
  return bestc
}
```

Part B: Lua via Python (8 exercises)

B1. In `nb.lua`, find the equivalent of Python's `class Obj(dict)`. Write 3-4 sentences explaining how `isa()` and `metatables` give you the same dot-notation access that Python's `__getattr__` provides.

A(B1). In Python, class `Obj(dict)` is a real class that overrides `__getattr__`. This is what lets us access dictionary keys as if they were attributes (e.g., typing `d.x` instead of `d['x']`).

Lua doesn't have classes in the same way. Instead, it uses `metatables` to fake it. By setting `mt.__index = mt`, Lua is told: "If you can't find a function (like add) inside the specific table `i`, go look for it in `mt` instead." This mimics the way Python looks up methods in a class definition if they aren't found in the instance itself.

B2. Look at this Lua function signature:

```
function NUM.add(i,v,     d)
```

The extra spaces before `d` are a Lua convention: everything after the gap is a local variable, not a parameter. So `i` and `v` are inputs; `d` is scratch space. Lua doesn't have a `local` keyword in function signatures, so this is how we signal intent.
Question: What are `i` and `v` in this context? Write the Python equivalent with type hints.

A(B2).

That variable d after the big gap is a local scratch variable. Lua doesn't have a simple way to declare locals inside a function header, so developers often put them at the end of the argument list as a hint to the reader that "these aren't parameters, they are just for internal use."

Python Equivalent

```
def add(i: Obj, v: float) -> None:
      # ... logic here ...
```

B3. Compare `like()` in both files. The Lua version uses `i.sd^2` but the Python version computes `sd` on the fly from `m2`. Why the difference? Which design would you prefer and why?

A(B3) The Lua version calculates standard deviation (`sd`) eagerly inside `add()`, updating it every time a number is added. The Python version calculates it lazily inside `like()`, computing it only when a probability is requested.

I prefer the lazy (Python) design . Calculating sqrt is computationally expensive. In an incremental Naive Bayes where we might train on thousands of rows before testing, updating `sd` on every insertion waste CPU cycles. It is more efficient to compute it only when needed for classification.

B4. At the bottom of `nb.lua`:

```
return {the=the, SYM=SYM, NUM=NUM, ...}
```

What's this doing? What's the Python equivalent? (Hint: think about `import`.)

A(B4) The `return { ... }` at the end of `nb.lua` constructs and returns a table containing the module's public API (functions and classes). When another file calls `require "nb"`, this table is assigned to the variable.

This is handled implicitly by the Python module system. When we run `import nb`, Python creates a module object containing all global definitions (classes and functions) from the file.

B5. In Python we write:

```
for n, row in enumerate(rows):
```

Find the Lua equivalent in `nb()`. How does Lua handle the "give me index and value" problem differently?

A(B5) Lua's `iter()` (as defined in `lib.lua`) typically returns only the value (the row). Lua handles the index manually by initializing a counter variable (`n`) before the loop and incrementing it inside the loop body.

Lua

```
n = 0
for row in iter(items) do
  -- ... logic ...
  n = n + 1
end
```

B6. (for grad students) Look at `Cols()` in both files. Python uses list comprehensions:

```
x=[c for c in all if not re.search(r"[!X]$", c.txt)]
```

What does Lua use instead? Find the equivalent line in `nb.lua`.

A(B6). Python uses list comprehensions for filtering. Lua uses a higher-order function `sel` (select) combined with an anonymous function.

Equivalent line in `nb.lua`

```
x = sel(all, function(c) return not c.txt:find"[!X]$" end)
```

B7. (for grad students) In `lib.lua`, find the `most()` function. It returns the key where `f(k,v)` is maximum. Write a Python equivalent in 3-4 lines.

A(B7). The `most` function finds the key `k` that maximizes `f(k,v)`.

```
def most(t, f):
   best, max_val = None, -1E32
   for k, v in t.items():
       if (tmp := f(k, v)) > max_val:
          best, max_val = k, tmp
   return best
```

B8. (for grad students) The Python code uses `match/case` for pretty-printing in `o()`. The Lua version uses `if/elseif`. But there's a deeper difference in how they detect types. What is it? (Hint: look at `math.type()` and `getmetatable()`.)

`A(B8)` Python uses Structural Pattern Matching (`match/case`) which checks the runtime class of the object (e.g., `dict`, `float`, `list`).

Lua uses `if/elseif` with helper functions like `math.type()` and `getmetatable()`. The deeper difference is that Lua only has 8 primitive types (tables, numbers, strings, etc.). It does not have a class system; "objects" are just tables. To distinguish a "class instance" from a plain list/dict, Lua has to check if the table has a metatable (`getmetatable(t)`), whereas Python natively distinguishes between types.

Part C: First-Class Functions (6 exercises)

```
-- fp.lua
-- Part C: First-Class Functions
```

```lua
-- Pretty-print a 1D array-like table: {1,4,9}
local function show(t)
  if t == nil then return "nil" end
  local parts = {}
  for i = 1, #t do
    parts[#parts + 1] = tostring(t[i])
  end
  return "{" .. table.concat(parts, ",") .. "}"
end

-- C1. collect(t, f) — map
local function collect(t, f)
  local out = {}
  for i = 1, #t do
    out[i] = f(t[i])
  end
  return out
end

-- C2. select(t, f) — filter (keep truthy)
local function select(t, f)
  local out = {}
  for i = 1, #t do
    local x = t[i]
    if f(x) then
      out[#out + 1] = x
    end
  end
  return out
end

-- C3. reject(t, f) — filter (keep falsy)
local function reject(t, f)
  local out = {}
  for i = 1, #t do
    local x = t[i]
    if not f(x) then
      out[#out + 1] = x
    end
  end
  return out
end

-- C4. inject(t, acc, f) — fold left
local function inject(t, acc, f)
  local a = acc
  for i = 1, #t do
    a = f(a, t[i])
  end
  return a
end

-- C5. detect(t, f) — first element matching predicate, or nil
```

```lua
local function detect(t, f)
  for i = 1, #t do
    local x = t[i]
    if f(x) then return x end
  end
  return nil
end

-- C6. Python-style range for Lua
-- Semantics:
--   range(stop)                -> 0, 1, ..., stop-1
--   range(start, stop)         -> start, start+1, ..., stop-1
--   range(start, stop, step)   -> start, start+step, ... up to but NOT including stop
-- Works with negative steps too.

local function range(a, b, c)
  local start, stop, step

  if b == nil then
    -- range(stop)
    start, stop, step = 0, a, 1
  else
    -- range(start, stop[, step])
    start, stop, step = a, b, c or 1
  end

  assert(step ~= 0, "range: step cannot be 0")

  -- Iterator state: current value to yield next
  local cur = start

  return function()
    -- Stop conditions (stop is exclusive, like Python)
    if step > 0 then
      if cur >= stop then return nil end
    else
      if cur <= stop then return nil end
    end

    local out = cur
    cur = cur + step
    return out
  end
end

-------------------------------------------------------------
-- Tests
-------------------------------------------------------------

local t1 = {1, 2, 3}
local squares = collect(t1, function(x) return x * x end)
print("C1 collect squares:", show(squares), " expected {1,4,9}")
```

```lua
local t2 = {1, 2, 3, 4, 5}
local evens = select(t2, function(x) return x % 2 == 0 end)
print("C2 select evens:", show(evens), " expected {2,4}")

local odds = reject(t2, function(x) return x % 2 == 0 end)
print("C3 reject evens (keep odds):", show(odds), " expected {1,3,5}")

local sum = inject({1, 2, 3, 4}, 0, function(a, x) return a + x end)
print("C4 inject sum:", sum, " expected 10")

local prod = inject({1, 2, 3, 4}, 1, function(a, x) return a * x end)
print("C4 inject product:", prod, " expected 24")

local first_gt2 = detect({1, 2, 3, 4}, function(x) return x > 2 end)
print("C5 detect first > 2:", first_gt2, " expected 3")

print("C6 range(5): expected 0 1 2 3 4")
for x in range(5) do io.write(x, " ") end
io.write("\n")

print("range(1, 5): expected 1 2 3 4")
for x in range(1, 5) do io.write(x, " ") end
io.write("\n")

print("range(1, 10, 2): expected 1 3 5 7 9")
for x in range(1, 10, 2) do io.write(x, " ") end
io.write("\n")

print("range(10, 1, -3): expected 10 7 4")
for x in range(10, 1, -3) do io.write(x, " ") end
io.write("\n")

print("range(-2, -8, -2): expected -2 -4 -6")
for x in range(-2, -8, -2) do io.write(x, " ") end
io.write("\n")
```