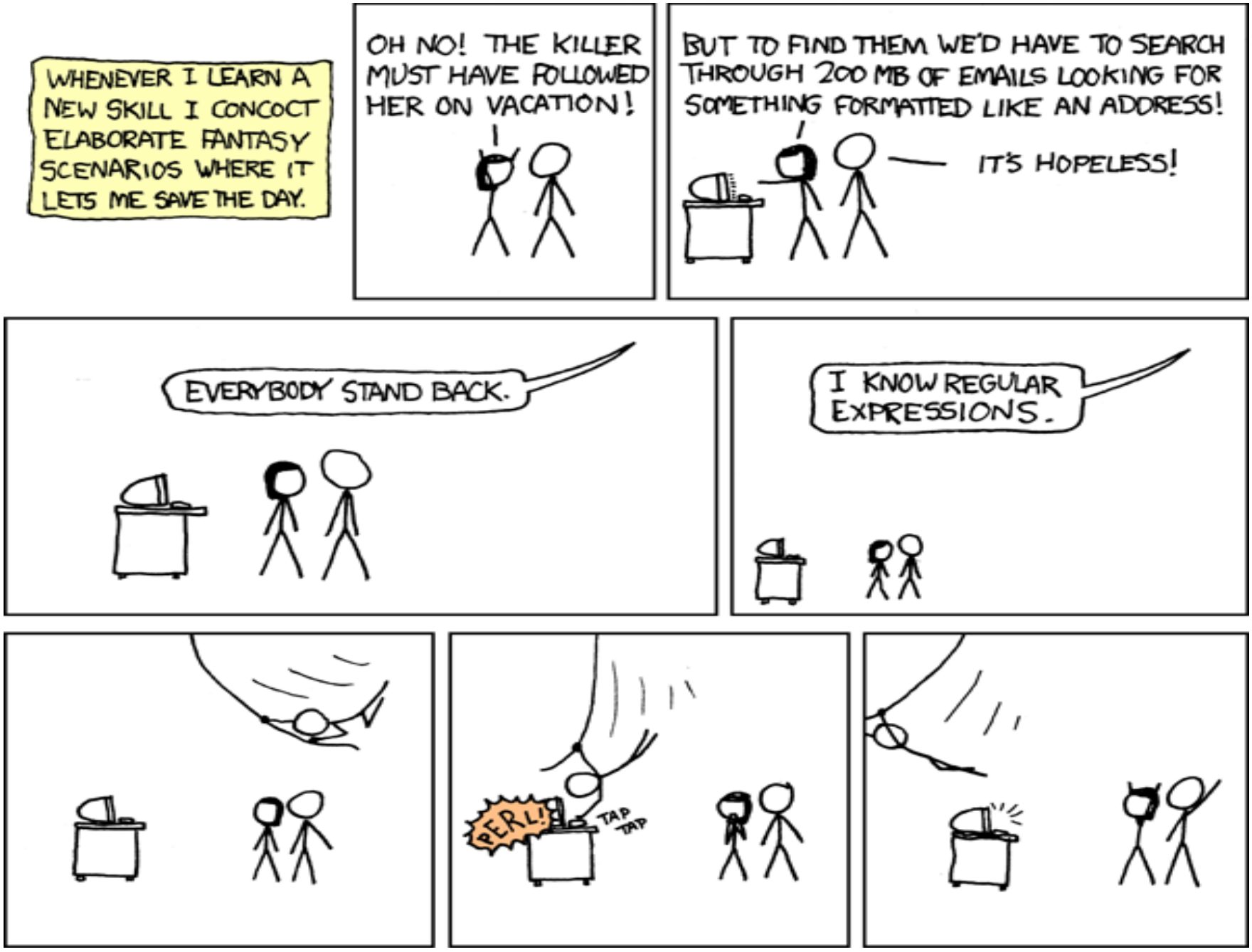
Lecture 5: Python

xkcd.com/208


WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Python

- developed ~1991 by Guido van Rossum
  - CWI, Amsterdam => ... => Google => Dropbox
- "I was looking for a 'hobby' programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus)."

  Guido von Rossum

Python constructs

- constants, variables, types
- operators and expressions
- statements, control flow
- aggregates
- functions, libraries
- classes, objects, modules
- etc.

Constants, variables, operators

- constants
  - integers, floats, `True/False`
  - `'string', "string", r'…', r"…", '''potentially multi-line string'''`

    no difference between single and double quotes

    `r'...'` is a raw string: doesn't interpret \ sequences within
- variables
  - hold strings or numbers, as in Awk

    no automatic coercions; interpretation determined by operators and context
  - no declarations (almost)
  - variables are either global or local to a function (or class)
- operators
  - mostly like C, but no ++, --, **`?:`**
  - relational operators are the same for numbers and strings
  - string concatenation uses +
  - format with `"fmt string" % (list of expresssions)`

Statements, control flow

- statements
  - assignment, control flow, function call, …
  - scope indicated by [consistent] indentation; no terminator or separator

- control flow

```
if condition:
    statements
elif condition:
    statements
else:
    statements
while condition:
    statements
for v in list:
    statements
```

[**break, continue** to exit early]

```
try:
    statements
except:
    statements
```

Exception example

```
import string
import sys

def cvt(s):
  while len(s) > 0:
    try:
      return string.atof(s)
    except:
      s = s[:-1]
  return 0

s = sys.stdin.readline()
while s != '':
  print '\t%g' % cvt(s)
  s = sys.stdin.readline()
```

Lists

- list, initialized to empty `food = []`
  - list, initialized with 3 elements:

```
food = [ 'beer', 'pizza', "coffee" ]
```

- elements accessed as `arr[index]`
  - indices from 0 to `len(arr)-1` inclusive
  - add new elements with list.append(value) : `food.append('coke')`
  - slicing: `list[start:end]` is elements `start..end-1`
- example: echo command:

```
for i in range(1, len(sys.argv)):
    if i < len(sys.argv):
        print sys.argv[i],    # , at end suppresses newline
    else:
        print sys.argv[i]
```

- tuples are like lists, but are constants

```
soda = ( 'coke', 'pepsi' )
soda.append('dr pepper')
```

is an error

List Comprehensions

```
>>> x = []
>>> for i in range(0,10): x.append(i)
...
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x = [i for i in range(10)]
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> pow2 = [2**i for i in range(10)]
>>> pow2
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Dictionaries (== associative arrays)

- dictionaries are a separate type from lists
  - subscripts are arbitrary strings
  - elements initialized with `dict = {'pizza':200, 'beer':100}`
  - accessed as `dict[str]`
- example: add up values from name-value input

pizza	200
beer	100
pizza	500
coke	50

```
import sys, string, fileinput
val = {}   # empty dictionary
line = sys.stdin.readline()
while line != "":
  (n, v) = line.strip().split()
  if val.has_key(n): # or n in val
    val[n] += string.atof(v)
  else:
    val[n] = string.atof(v)
  line = sys.stdin.readline()
for i in val:
  print "%s\t%g" % (i, val[i])
```

AWK version:

```
     { val[$1] += $2 }
END {
  for (i in val)
    print i, val[i] }
```

Dictionaries (== associative arrays)

- dictionaries are a separate type from lists
  - subscripts are arbitrary strings
  - elements initialized with `dict = {'pizza':200, 'beer':100}`
  - accessed as `dict[str]`
- example: add up values from name-value input

pizza	200
beer	100
pizza	500
coke	50

```
import sys, string, fileinput
val = {}   # empty dictionary
line = sys.stdin.readline()
while line != "":
  (n, v) = line.strip().split()
  val[n] = val.get(n, 0) + string.atof(v)
  line = sys.stdin.readline()
for i in val:
  print "%s\t%g" % (i, val[i])
```

AWK version:

```
     { val[$1] += $2 }
END {
  for (i in val)
    print i, val[i] }
```

Functions

```
def div(num, denom):
    ''' computes quotient & remainder.
        denom should be > 0.'''
    q = num / denom
    r = num % denom
    return (q, r)  # returns a tuple
```

- functions are objects
  - can assign them, pass them to functions, return them from fcns
- parameters are passed call by value
  - can have named arguments and default values and arrays of name-value pairs
- variables are local unless declared **`global`**
- EXCEPT if you only read a global, it's visible inside the function anyway!

```
x = 1; y = 2
def foo(): y = 3; print x, y
foo()
  1 3
print y
  2
```

Function arguments

- positional arguments

  ```
  def div(num, denom): …
  ```

- keyword arguments

  ```
  def div(num=1, denom=1):
  ```

  - must follow any positional arguments

- variable length argument lists *

  ```
  def foo(a, b=1, *varlist)
  ```

  - variable length argument must follow positional and keyword args

- additional keyword arguments **

  ```
  def foo(a, b=1, *varlist, **kwords)
  ```

  - all extra name=val arguments are put in dictionary kwords

Regular expressions

`re.search(re, str)`	find first match of re in str
`re.match(re, str)`	test for anchored match
`re.split(re, str)`	split str into a list of matches around re
`re.findall(re, str)`	list of all matches of re in str
`re.sub(re, rpl, str)`	replace all re in str with rpl
`\d \D \w \W \s \S`	digit non-digit word non-word space non-space

Warning: Patterns are not necessarily matched leftmost-longest

Replacements are global by default

```
>>> s = "inches and inches in india and indonesia
>>> re.sub('in|inch', "X", s)
Xches and Xches X Xdia and Xdonesia

>>> re.sub('inch|in', "X", s)
Xes and Xes X Xdia and Xdonesia
```

Classes and objects

```
class Stack:
   def __init__(self):  # constructor
      self.stack = []   # local variable
   def push(self, obj):
      self.stack.append(obj)
   def pop(self):
      return self.stack.pop()    # list.pop
   def len(self):
      return len(self.stack)

stk = Stack()
stk.push("foo")
if stk.len() != 1: print "error"
if stk.pop() != "foo": print "error"
del stk
```

- always have to use **`self`** in definitions
- special names like **`__init__`** (constructor)
- information hiding only by convention; not enforced

Modules

- a module is a library, often one class with lots of methods
- core examples:
  - sys
    argv, stdin, stdout
  - string
    find, replace, index, …
  - re
    match, sub, …
  - os
    open, close, read, write, getenviron, system, …
  - fileinput
    awk-like processing of input files
  - urllib, requests
    manipulating url's, accessing web sites

Review: Formatter in AWK

```
/./   { for (i = 1; i <= NF; i++)
            addword($i)
      }
/^$/  { printline(); print "" }
END   { printline() }

function addword(w) {
    if (length(line) + length(w) > 60)
        printline()
    line = line space w
    space = " "
}

function printline() {
    if (length(line) > 0)
        print line
    line = space = ""
}
```

Formatter in Python

```
import sys, string
line=""; space = ""
def main():
    buf = sys.stdin.readline()
    while buf != "":
        if len(buf) == 1:
            printline()
            print ""
        else:
            for word in string.split(buf):
                addword(word)
        buf = sys.stdin.readline()
    printline()

def addword(word):
    global line, space
    if len(line) + len(word) > 60:
        printline()
    line = line + space + word
    space = " "

def printline():
    global line, space
    if len(line) > 0:
         print line
    line = space = ""

main()
```

Python ecosystem

- installing Python
  - binary distributions
  - compile from source
- PyPI
  - repository for Python packages
- pip
  - installer for Python packages from PyPI
- virtualenv
  - keep different installations from interfering with each other

- Python 2 vs Python 3
  - print
  - integer arithmetic
  - Unicode

Surprises, gotchas, etc.

- indentation for grouping, ":" always needed
- no implicit conversions
  - often have to use class name (`string.atof(s)`)
- **`elif`**, not **`else if`**
- no **`++`**, **`--`**, **`?:`**
- assignment is not an expression
  - no equivalent of `while ((c = getchar()) != EOF) ...`
- % for string formatting
- **`global`** declaration to modify non-local variables in functions
- no uninitialized variables

```
if v != None:
if arr.has_key():
```

- regular expressions not leftmost longest
  - re.match is anchored, re.sub replaces all

Python practice, problem solving with code, etc.

www.pythonchallenge.com

NB: don't confuse with
www.pythonchallenge.org