Python

• developed ~1991 by Guido van Rossum
  - CWI, Amsterdam => ... => Google
• for scripting but very interactive

% python
>>> print "hello, world"
hello, world
>>> print 355.0/113
3.14159292035
>>> import math
>>> print math.pi
3.14159265359

• Disclaimer: I am NOT a Python expert
• see www.python.org

xkcd.com/353

I DUNNO...
DYNAMIC TYPING?
WHITESPACE?
COME JOIN US!
PROGRAMMING
IS FUN AGAIN!
IT'S A WHOLE
NEW WORLD
UP HERE!
BUT HOW ARE
YOU FLYING?

I JUST TYPED
import antigravity
THAT'S IT?

... I ALSO SAMMED
EVERYTHING IN THE
MEDICINE CABINET
FOR COMPARISON.

BUT I THINK THIS
IS THE PYTHON.
World's most boring example (yet again)

    for fahr in range(0, 300, 20):
        print "%3d %6.1f" % (fahr, 5.0/9*(fahr-32))

• grouping by indentation
• if elif else; while; for i in list
• constants: numbers, strings
  - \ escapes interpreted in '...' and "..." but not in r'...' or r"...
• variables hold strings or numbers as in Awk
  - interpretation determined by operators & context; have to be initialized
• operators:
  - arithmetic operators like C but no ++, --, ?: = is not an operator
  - string concatenation uses +
  - relational operators are the same for string and numeric comparisons
  - format with "fmt string" % (list of exprs)
• mostly uses class libraries for operations
  - many fewer operators than Perl
  - class libraries ("modules") instead, e.g., string, re, sys, os, math, ...

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
• echo command:

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

• tuples are like lists, but are constants
    soda = ( 'coke', 'pepsi' )
    soda.append('dr pepper') is an error
Dictionaries (== associative arrays)

- dictionaries are a separate type from arrays
  - subscripts are arbitrary strings
  - elements initialized with \texttt{dict = \{’pizza’:200, ’beer’:100\}}
  - accessed as \texttt{dict[str]}

- example: add up values from name-value input
  
  \begin{verbatim}
  pizza 200  
  beer 100  
  pizza 500  
  coke 50   
  \end{verbatim}

\begin{verbatim}
import sys, string, fileinput
val = {}  # empty dictionary
line = sys.stdin.readline()
while (line != ""): 
  (n, v) = line.strip().split()
  if val.has_key(n):
    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])
\end{verbatim}

\textbf{AWK version:}
\begin{verbatim}
  \{ val[$1] += $2 \}
  END {
    for (i in val)
      print i, val[i] }
\end{verbatim}

Regular expressions and substitution

- underlying mechanisms like Perl: libraries, not operators, less syntax
  \begin{verbatim}
  re.search(pat, str)  \ find first match
  re.match(pat, str)  \ test for anchored match
  re.split(pat, str)  \ split into list of matches
  re.findall(pat, str)  \ list of all matches
  re.sub(pat, repl, str)  \ replace all pat in str by repl
  \end{verbatim}

- shorthands in patterns
  \begin{verbatim}
  \d = digit, \D = non-digit
  \w = "word" character [a-zA-Z0-9\_], \W = non-word character
  \s = whitespace, \S = non-whitespace
  \b = word boundary, \B = non-boundary
  \end{verbatim}

- substrings
  - matched parts are saved for later use in \texttt{\1, \2, ...}
  \begin{verbatim}
  s = re.sub(r' (\S+)\s+(\S+) ', r'\2 \1', s)  \ \text{flips 1st 2 words of s}
  \end{verbatim}

- watch out
  - \texttt{re.match} is anchored (match must start at beginning)
  - patterns are not matched leftmost longest
Functions

def name(arg, arg, arg):
    statements of function

def div(a, b):
    ''' computes quotient & remainder. b had better be > 0'''
    q = a / b
    r = a % b
    return (q, r)  # returns a list

• functions are objects
  - can assign them, pass to functions, return from fcn

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

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?
Review: Formatter in AWK

```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 (version 1)

```python
import sys, string
line=""; space = ""
def main():
    buf = sys.stdin.read()
    for word in string.split(buf):
        addword(word)
        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()
```
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
- \% for string formatting
- global declaration to modify non-local variables in functions
- no uninitialized variables
  
  ```python
  if v != None:
      if arr.has_key():
  ```
- regular expressions not leftmost longest
  - re.match is anchored, re.sub replaces all
- function call needs parens
  - foo is not the same as foo()

What makes Python successful?

- comparatively small, simple but rich language
  - regular expressions, strings, tuples, assoc arrays
  - clean (though limited) object-oriented mechanism
  - reflection, etc.

- efficient enough
  - seems to be getting better
- large set of libraries
  - extensible by calling C or other languages
- embeddings of major libraries
  - e.g., TkInter for GUIs
- open source with large and active user community
- standard: there is only one Python
  - but watch out for Python 3000, which is not backwards compatible

- a reaction to the complexity and general ugliness of Perl?
Perl vs. Python

- most tradeoffs in Awk made to keep it small and simple
- most tradeoffs in Perl made to make it powerful and expressive
- most tradeoffs in Python made to make it small and interactive

- **domain of applicability**
  - Perl does system stuff well
  - Python is a lot simpler
  - Python is more extensible?

- **efficiency**
  - seem close to the same now

- **standardization**
  - there’s only one Perl but it evolves
  - there’s only one Python but it evolves

- **program size, installation, environmental assumptions**
  - both are big, use a big configuration script, take advantage of the environment
  - Python is somewhat smaller