Python

- high level, expressive, readable
- efficient
- weakly typed; no declarations for variables
- rich libraries
- escapes to other languages
- good documentation
- language is evolving

- the one scripting language to have if you're only having one?

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

Python source materials

- **Bob Dondero's Python summary from Spring 2011**
  - http://www.cs.princeton.edu/courses/archive/spring11/cos333/reading/pythonsummary.pdf
- **bwk's Python help file:**
  - http://…/cos333/python.help/
- **Official Python documentation:**
  - http://docs.python.org/tutorial/
  - http://docs.python.org/reference
  - http://docs.python.org/library
- **Idiomatic Python:**
  - http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html
- **Python challenge:**
  - http://www.pythonchallenge.com/

Python constructs

- constants, variables, types
- operators and expressions
- statements, control flow
- aggregates
- functions
- libraries
- classes
- 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
  - 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]

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 argv[i],    # , at end suppresses newline
    else:
        print argv[i]
```

- tuples are like lists, but are constants

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

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:**

```
print ' '.join(sys.argv)
```

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

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

- **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**

```
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 and substitution

- **underlying mechanisms like Perl: libraries, not operators, less syntax**

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

- **shorthands in patterns**

  \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

- **substrings**
  - matched parts are saved for later use in \1, \2, ...

  `s = re.sub(r'(\S+)\s+(\S+)', r'\2 \1', s)` flips 1st 2 words of s

- **watch out**
  - re.match is anchored (match must start at beginning)
  - patterns are not matched leftmost longest

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
    manipulating url's

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 (version 1)

```
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()
```

Formatter in Python (comparing input styles)

```
def main():
    buf = sys.stdin.read()
    for word in string.split(buf):
        addword(word)
    printline()

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()
```

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 3, which is not backwards compatible

- **a reaction to the complexity and irregularity of Perl?**

Perl vs. Python

- **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