CS 5142
Scripting Languages

7/26/2012

Python

Outline

- Python

About Python

- First released 1991 by Guido van Rossum
  - Named after Monty Python comedy group; documentation uses examples from sketches
- General-purpose language
  - Not focused on just one domain only, such as application extension or web scripting
- Main focus: minimalist syntax, readability, comprehensive libraries (“batteries included”)
  - There should be only one obvious way to do it
- Used by YouTube and Google

How to Write + Run Code

- One-liner: **`python -c '`***command***`'`**
- Run script from file: **`python`** *file***`.py`**
- Debugger: **`python -m pdb`** *file***`.py`**
- Read-eval-print loop: **`python`**
- Run stand-alone: **`#!/usr/bin/env python`**

Example

```
#!/usr/bin/env python
import re, sys
cup2g = { 'flour': 110, 'sugar': 225, 'butter': 225 }
volume = { 'cup': 1, 'tbsp': 16, 'tsp': 48, 'ml': 236 }
weight = { 'lb': 1, 'oz': 16, 'g': 453 }
for line in sys.stdin.readlines():
   qty, unit, ing = re.search(r'([0-9.]+) (\w+) (\w+)', line).groups()
   if ing in cup2g and unit in volume:
      qty = float(qty) * cup2g[ing] / volume[unit]
      unit = 'g'
   elif unit in volume:
      qty = float(qty) * volume['ml'] / volume[unit]
      unit = 'ml'
   elif unit in weight:
      qty = float(qty) * weight['g'] / weight[unit]
      unit = 'g'
   sys.stdout.write('qty %d, unit %s, ing %s\n' % (int(qty),unit,ing))
```

Lexical Peculiarities

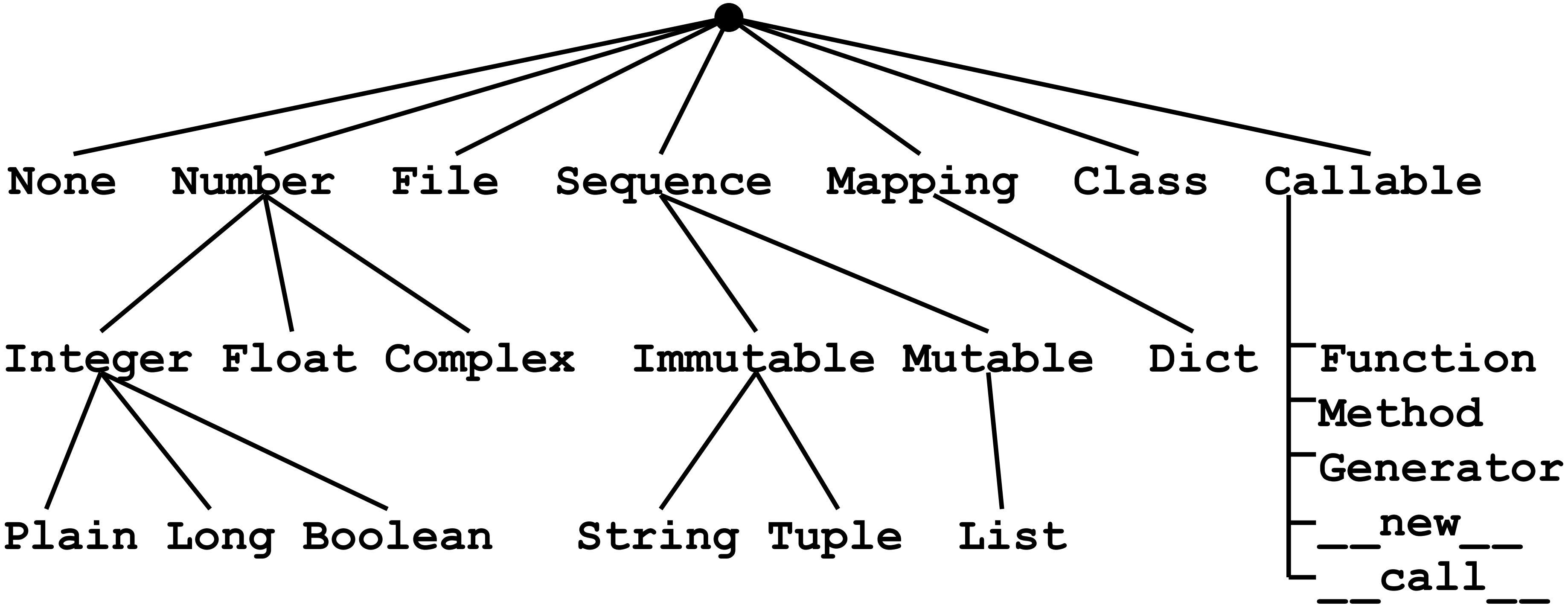
- Line break and indentation sensitive (!)

```
for line in sys.stdin.readlines():
   qty, unit, ing = re.search( \
      r'([0-9.]+) (\w+) (\w+)', line).groups()
   if ing in cup2g and unit in volume:
      qty = float(qty) * cup2g[ing] / volume[unit]
      unit = 'g'
   #  “suite” ends at dedent
```

- Single-line comments: #…
- Literals
  - **`True`**, **`False`**, **`None`**, **`123`**, **`12.34`**, **`'s'`**, **`"s"`**
  - No variable interpolation (use `%` operator instead)
  - Raw strings `r'…'` don’t even interpolate backslash
  - Triple-quoted strings `"""…"""` can contain newlines

Python

Types



None Number File Sequence Mapping Class Callable

Integer Float Complex Immutable Mutable Dict Function

Method

Generator

Plain Long Boolean String Tuple List __new__

__call__

Variable Declarations

- There are no explicit variable declarations
- Reading an undefined variable is an error
- "Local if written" rule:
  - If function contains assignment "**`x=y`**", then "**`x`**" is a local variable, even if there is another global "**`x`**"
  - If function only reads "**`x`**", it uses the global "**`x`**"
  - If function contains statement "**`global x`**", then assignment "**`x=y`**" writes the global "**`x`**" if any
- No implicit "this"
  - Field and method access need explicit receiver even from within same class

Python

Type Conversions

Type	Value	Boolean	Number	String
`bool`	`False`	Identity	`0`	Error
	`True`		`1`	
`int`	`0`	`False`	Identity	Error
	Other	`True`		
`str`	`""`	`False`	Error	Identity
	`"0"`	`True`		
	Other	`True`		
`None`		`False`	Error	Error
`list`	Empty	`False`	Error	Error
	Other	`True`		
`dict`	Empty	`False`	Error	Error
	Other	`True`		

Input and Output

- Output:
  - Statement: `print "hi"`
  - Statement without newline: `print "hi",`
  - Method call:
    ```
    import sys; sys.stdout.write("hi")
    ```
- Input:

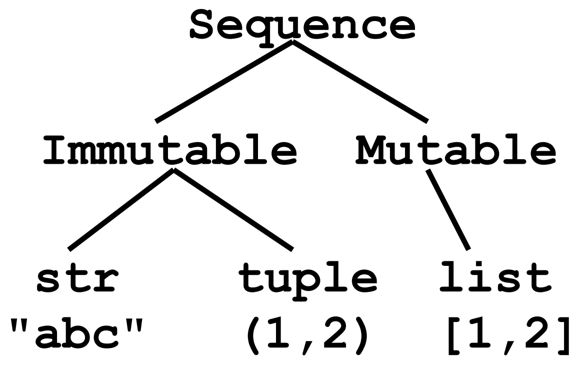
```
import sys;
line = sys.stdin.readline()
allLines = sys.stdin.readlines()
```

Python

Operators

`[]`		Array subscript
`.`		Attribute reference
`**`	2	Exponentiation
`~`	1	Bitwise negation
`+`, `-`	1	Positive, negative
`*`, `/`, `%`	2	Multiplicative
`+`, `-`	2	Additive
`<<`, `>>`	2	Shifts
`|`, `^`, `&`	2	Bitwise (not all same precedence)
`<`, `<=`, `>`, `>=`, `<>`, `!=`, `==`	2	Value comparison
`is`, `is not`	2	Identity comparison
`in`, `not in`	2	Membership test
`or`, `and`, `not`	2	Logical (not all same precedence)
`lambda`		Anonymous function

Arrays



Sequence

Immutable

Mutable

str

"abc"

tuple

(1,2)

list

[1,2]

- More general concept: sequence
- All sequences support
  - Membership test: `m in s`
  - Concatenation: `s1 + s2`
  - Indexing (0-based): `s[i]`
  - Slicing: `s[lb:ub]`, `s[lb:ub:step]`
  - Built-in functions: `len(s)`, `min(s)`, `max(s)`
- Constructing a list of numbers: `range(ub)`
- List mutation: `s[i]=m`, `s[lb:ub]=t`, `s.append(m)`, `s.pop()`, `s.sort()`, etc.

List Comprehensions

- Concise syntax for generating lists:

  listCompr ::= [*expr forClause comprClause**]
  forClause ::= **for** *id* **in** *expr*
  comprClause ::= *forClause* | *ifClause*
  ifClause ::= **if** *expr*

- Example:

```
l = [1,2,3,4]
t = 'a', 'b'
c1 = [x for x in l if x % 2 == 0]
c2 = [(x, y) for x in l if x < 3 for y in t]
print str(c1) # [2, 4]
print str(c2) # [(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

```
c2 = []
for x in l:
  if x < 3:
    for y in t:
      c2.append((x,y))
```

Hashes

- Python calls them “dictionary”, not “hash”
- Initialization:
  `d = {'lb': 1, 'oz': 16, 'g': 453}`
  or **`d = dict(lb=1, oz=16, g=453)`**
- Indexing: **`d['lb']`**
  - Can use any immutable type as key, including **`int`**, **`str`**, **`tuple`**
- Deleting: **`del weight['lb']`**
- Membership test: **`k in d`**
- Lots of other methods, e.g.,
  `d.keys()`, **`d.values()`**, **`d.has_key(k)`**

Control Statements

Conditional		**if** *expr***:** … [**elif** …] [**else:** …]
Loops	Fixed	**for** *target* **in** *expr***:** … [**else:** …]
	Indefinite	**while** *expr***:** … [**else:** …]

Unstructured control		**break**
		continue
		return [*expr*]
		yield [*expr*]

Exception handling		**try:** … (**except** [*Cls* [,*id*]]**:** …)* \ [**else:** …] [**finally:** …] **raise** [*expr*]

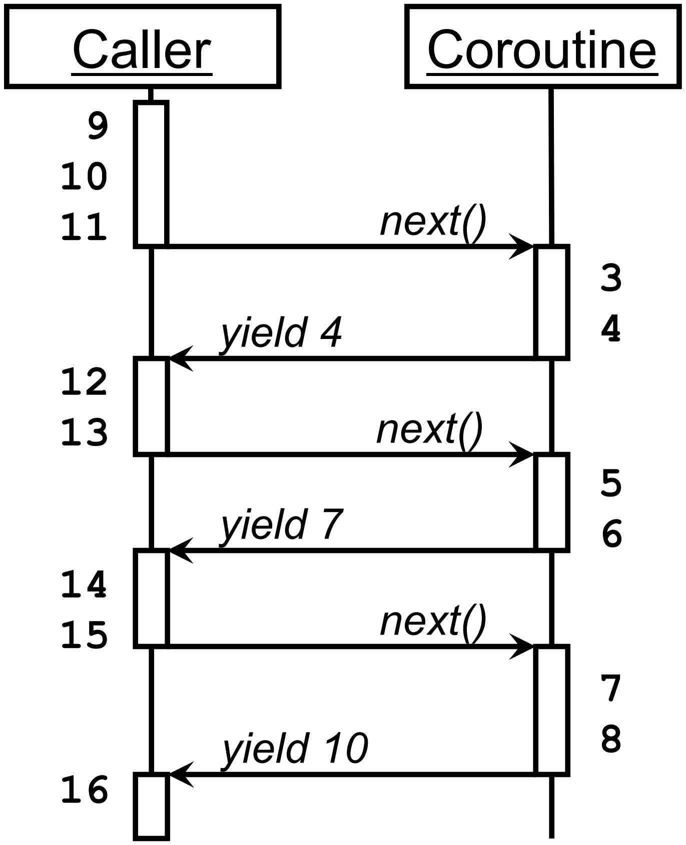
Writing Subroutines

- Declaration: `def` *id*`(`*arg*`*):` …
  - Can be nested
  - First-class: can store closure in variable
- Arguments: *arg* ::= *id* | *id* `=` *expr* | `*`*id* | `**`*id*
  - Mandatory argument: *id*
  - Optional argument: *id* `=` *expr*
  - Variable positional arguments: `*`*id*
  - Variable keyword arguments: `**`*id*
- Anonymous function: `lambda` *arg*`*:` …

Python≥2.5

Generators

```
#!/usr/bin/env python        # 1
def myGenerator(x):          # 2
   x = x + 3                 # 3
   yield x                   # 4
   x = x + 3                 # 5
   yield x                   # 6
   x = x + 3                 # 7
   yield x                   # 8
myCoroutine = myGenerator(1) # 9
print '1st call:'            #10
print myCoroutine.next()     #11
print '2nd call:'            #12
print myCoroutine.next()     #13
print '3rd call:'            #14
print myCoroutine.next()     #15
print 'after 3rd call'       #16
```



Python can also treat a generator result as an iterator:

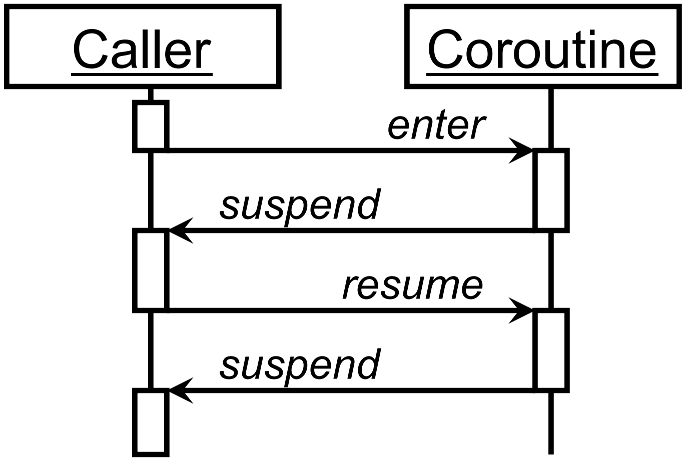
```
for y in myGenerator(1): print y
```

17

Concepts

Co-Routines

- Coroutine = subroutine that suspends execution, but remembers state for later resumption
  - Requires saving and restoring (partial) stack
  - Useful for separating logic of producer (e.g., tree walk) from consumer (e.g., adding values in tree)
- Python "generators" are a new implementation of the old concept coroutines



Python≥2.2

Using Objects

`a1 = Apple(150, 'green')`
`a2 = Apple(150, 'green')`
	Constructor calls

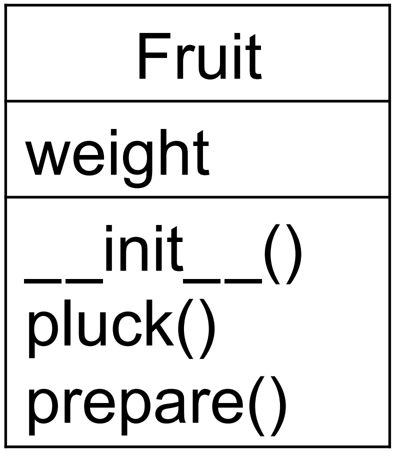
`a2.color = 'red'`	Field assignment

`print a1.prepare('slice')`
`print a2.prepare('squeeze')`
	Method calls

Python≥2.2

Defining Classes

```
class Fruit:
   def __init__(self, weight):
      self.weight = weight
   def pluck(self):
      return 'fruit(' + self.weight + 'g)'
   def prepare(self, how):
      return how + 'd ' + self.pluck()
```



Fruit

weight

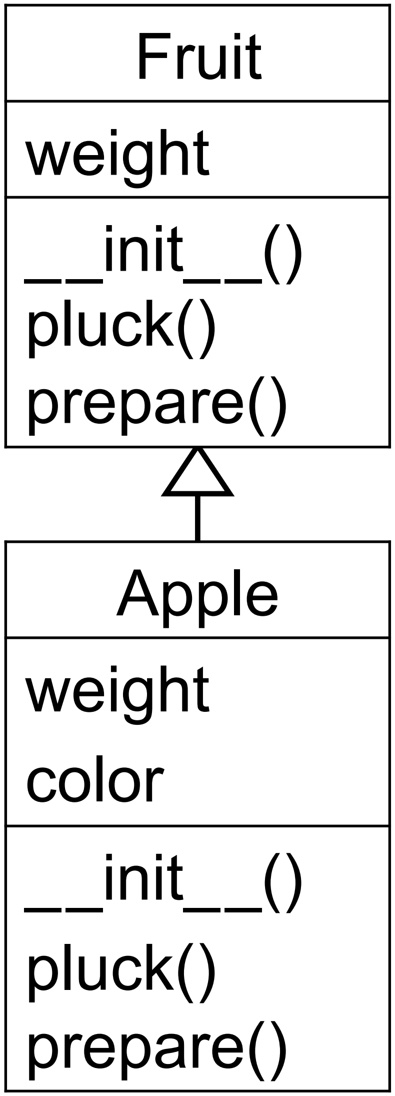
__init__()
pluck()
prepare()

Python≥2.2

Inheritance in Python

```
class Fruit:
   def __init__(self, weight):
      self.weight = weight
   def pluck(self):
      return 'fruit(' + self.weight + 'g)'
   def prepare(self, how):
      return how + 'd ' + self.pluck()
```

```
class Apple(Fruit):
   def __init__(self, weight, color):
      Fruit.__init__(self, weight)
      self.color = color
   def pluck(self):
      return self.color + ' apple'
```



Fruit

weight

__init__()
pluck()
prepare()

Apple

weight
color

__init__()
pluck()
prepare()

Reference

Python Documentation

- Tutorial:
  http://docs.python.org/tut/tut.html
- Library reference:
  http://docs.python.org/lib/lib.html
- Language reference:
  http://docs.python.org/ref/ref.html
- Book:
  Dive Into Python. Mark Pilgrim. Apress, 2004.
  Available for free online:
  http://diveintopython.net/

Soap-Box

Evaluating Python

Strengths

- Readability
- Libraries
- Simple yet powerful built-in data types
- Good for larger projects

Weaknesses

- Indentation-sensitive
- Unusual syntax
- RegExps less tightly integrated

Administrative

Last Slide

- Office hours moved to Wednesday.

- Today’s lecture
  - Python

- Next lecture
  - Context Free Grammars