Python

Department of Computer Science

University of El-oued

3-Mar-23

Introduction

- Most recent popular (scripting/extension) language
  - although origin ~1991
- heritage: teaching language (ABC)
- object-oriented

Python philosophy

- Coherence
    - not hard to read, write and maintain
- power

Python features

Lutz, *Programming Python*

no type declarations	simpler, shorter, more flexible
automatic memory management	garbage collection
high-level data types and operations	fast development
object-oriented programming	code structuring and reuse, C++
classes, modules, exceptions	"programming-in-the-large" support
dynamic loading of C modules	simplified extensions, smaller binaries

Python features

Lutz, *Programming Python*

interactive, dynamic nature	incremental development and testing
access to interpreter information	metaprogramming, introspective objects
compilation to portable byte-code	execution speed, protecting source code

Python

- elements from C++, Modula-3 (modules), ABC
- same family as Perl, Tcl, Scheme, REXX, BASIC dialects

Uses of Python

- shell tools
  - system admin tools, command line programs
- rapid prototyping and development
- graphical user interfaces
- database access
- distributed programming
- Internet scripting

Python structure

- modules: Python source files or C extensions
  - import, top-level via from, reload
- statements
  - control flow
  - create objects
  - indentation matters – instead of {}
- objects
  - everything is an object

Basic operations

- Assignment:
  - `size = 40`
  - `a = b  = c = 3`
- Numbers
  - integer, float
  - complex numbers: `1j+3, abs(z)`
- Strings
  - `'hello world', 'it\'s hot'`
  - `"bye world"`

String operations

- concatenate with + or neighbors
  - `word = 'Help' + x`
  - `word = 'Help' 'a'`
- subscripting of strings
  - `'Hello'[2]` → 'l'
  - slice: `'Hello'[1:2]` → 'el'
  - `word[-1]` → last character
  - `len(word)` → 5

Lists

- lists can be heterogeneous
  - `a = ['spam', 'eggs', 100, 1234, 2*2]`
- Lists can be indexed and sliced:
  - `a[0]` → spam
  - `a[:2]` → ['spam', 'eggs']
- Lists can be manipulated
  - `a[2] = a[2] + 23`
  - `a[0:2] = [1,12]`
  - `a[0:0] = []`
  - `len(a)` → 5

Basic programming

```
a,b = 0, 1
# non-zero = true
while b < 10:
  # formatted output, without \n
  print (b)
  # multiple assignment
  a,b = b, a+b
```

Control flow: if

```
x = int(raw_input("Please enter #:"))
if x < 0:
  x = 0
  print ('Negative changed to zero')
elif x == 0:
  print ('Zero')
elif x == 1:
  print ('Single')
else:
  print ('More')
```

- no case statement

Control flow: for

```
a = ['cat', 'window', 'defenestrate']
for x in a:
  print (x, len(x))
```

- no arithmetic progression, but
  - `range(10)` → [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

  -
    ```
    for i in range(len(a)):
     print (i, a[i])
    ```
- do not modify the sequence being iterated over

Loops: break, continue, else

- break and continue like C
- else after loop exhaustion

```
for n in range(2,10):
  for x in range(2,n):
    if n % x == 0:
      print (n, 'equals', x, '*', n/x)
      break
  else:
    # loop fell through without finding a factor
    print (n, 'is prime‘)
```

Defining functions

```
def fib(n):
  """Print a Fibonacci series up to n."""
  a, b = 0, 1
  while b < n:
    print (b)
    a, b = b, a+b

>>> fib(2000)
```

- First line is *docstring*
- first look for variables in local, then global
- need global to assign global variables

Lambda forms

- anonymous functions
- may not work in older versions

```
def make_incrementor(n):
  return lambda x: x + n

f = make_incrementor(42)
f(0)
f(1)
```

List methods

- `append(`*x*`)`
- `extend(`*L*`)`
  - append all items in list (like Tcl lappend)
- `insert(`*i*`,`*x*`)`
- `remove(`*x*`)`
- `pop([i]), pop()`
  - create stack (FIFO), or queue (LIFO) → pop(0)
- `index(`*x*`)`
  - return the index for value *x*

List methods

- `count(x)`
  - how many times x appears in list
- `sort()`
  - sort items in place
- `reverse()`
  - reverse list

Functional programming tools

- `filter(`*`function, sequence`*`)`

```
def f(x): return x%2 != 0 and x%3 == 0
filter(f, range(2,25))
```

- `map(`*`function, sequence)`*
  - call function for each item
  - return list of return values
- `reduce(`*`function, sequence`*`)`
  - return a single value
  - call binary function on the first two items
  - then on the result and next item
  - iterate

List comprehensions (2.0)

- Create lists without `map()`, `filter()`, `lambda`
- = expression followed by for clause + zero or more for or of clauses

```
>>> vec = [2,4,6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
```

List comprehensions

- cross products:

```
>>> vec1 = [2,4,6]
>>> vec2 = [4,3,-9]
>>> [x*y for x in vec1 for y in vec2]
[8,6,-18, 16,12,-36, 24,18,-54]
>>> [x+y for x in vec1 for y in vec2]
[6,5,-7,8,7,-5,10,9,-3]
>>> [vec1[i]*vec2[i] for i in
  range(len(vec1))]
[8,12,-54]
>>> [x * y for (x,y) in zip(vec1,vec2)]
```

List comprehensions

- can also use `if`:

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

del - removing list items

- remove by index, not value
- remove slices from list (rather than by assigning an empty list)

```
>>> a = [-1,1,66.6,333,333,1234.5]
>>> del a[0]
>>> a
[1,66.6,333,333,1234.5]
>>> del a[2:4]
>>> a
[1,66.6,1234.5]
```

Tuples and sequences

- lists, strings, **tuples**: examples of *sequence* type
- tuple = values separated by commas

```
>>> t = 123, 543, 'cat'
>>> t[0]
123
>>> t
(123, 543, 'cat')
```

Tuples

- Tuples may be nested

```
>>> u = t, (1,2)
>>> u
((123, 542, ‘cat'), (1,2))
```

- kind of like structs, but no element names:
  - (x,y) coordinates
  - database records
- like strings, immutable → can't assign to individual items

Tuples

- Empty tuples: ()

```
>>> empty = ()
>>> len(empty)
0
```

- one item → trailing comma

```
>>> singleton = 'foo',
```

Tuples

- sequence unpacking → distribute elements across variables

```
>>> t = 123, 543, 'cat'
>>> x, y, z = t
>>> x
123
```

- packing always creates tuple
- unpacking works for any sequence

Dictionaries

- like Tcl or awk associative arrays
- indexed by keys
- keys are any immutable type: e.g., tuples
- but not lists (mutable!)
- uses 'key: value' notation

```
>>> tel = {'hgs' : 7042, 'lennox': 7018}
>>> tel['cs'] = 7000
>>> tel
```

Dictionaries

- no particular order
- delete elements with del

```
>>> del tel['foo']
```

- keys() method → unsorted list of keys

```
>>> tel.keys()
['cs', 'lennox', 'hgs']
```

Conditions

- chained comparisons: a less than b AND b equals c:

  ```
  a < b == c
  ```

- and and or are short-circuit operators:
  - evaluated from left to right
  - stop evaluation as soon as outcome clear

Conditions

- Can assign comparison to variable:

```
>>> s1,s2,s3='', 'foo', 'cat'
>>> non_null = s1 or s2 or s3
>>> non_null
foo
```

- Unlike C, no assignment within expression

Comparing sequences

- unlike C, can compare sequences (lists, tuples, ...)
- lexicographical comparison:
  - compare first; if different → outcome
  - strings use ASCII comparison
  - can compare objects of different type, but by type name (list < string < tuple)

Comparing sequences

```
(1,2,3) < (1,2,4)
[1,2,3] < [1,2,4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1,2,3) == (1.0,2.0,3.0)
(1,2) < (1,2,-1)
```

Modules

- collection of functions and variables, typically in scripts
- definitions can be imported
- file name is module name + .py
- e.g., create module `fibo.py`

```
def fib(n): # write Fib. series up to n
   ...
def fib2(n): # return Fib. series up to n
```

Modules

- import module:

```
import fibo
```

- Use modules via "name space":

```
>>> fibo.fib(1000)
>>> fibo.__name__
'fibo'
```

- can give it a local name:

```
>>> fib = fibo.fib
>>> fib(500)
```

Modules

- function definition + executable statements
- executed only when module is imported
- modules have private symbol tables
- avoids name clash for global variables
- can import into name space:

```
>>> from fibo import fib, fib2
>>> fib(500)
```

- can import all names defined by module:

```
>>> from fibo import *
```

Module listing

- use `dir()` for each module

```
>>> dir(fibo)
['__name__', 'fib', 'fib2']
```

Exercice et Solution

- Implémentez une fonction *trier(classeur, valeur)* qui place une valeur dans un dictionnaire en fonction de son signe

-
  ```
  classeur = {'négatifs':[], 'positifs':[] }
  ```

-
  ```
  def trier(classeur, valeur):
      return classeur
  ```

SOLUTION

-
  ```
  def trier(classeur, valeur):
      if valeur >=0:
          classeur['positifs'].append(valeur)
      else:
          classeur['négatifs'].append(valeur)
      return classeur
  ```

-
  ```
  trier(classeur, 9)
  ```