Python
Henning Schulzrinne
Department of Computer Science
Columbia University
(based on tutorial by Guido van Rossum)

Introduction
• Most recent popular (scripting/extension) language
  • although origin ~1991
• heritage: teaching language (ABC)
  • Td: shell
• perl: string (regex) processing
• object-oriented
  • rather than add-on (OOTd)

Python philosophy
• Coherence
  • not hard to read, write and maintain
• power
• scope
  • rapid development + large systems
• objects
• integration
• hybrid systems

Python features
<table>
<thead>
<tr>
<th>universal &quot;first-class&quot; object model</th>
<th>Fewer restrictions and rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>run-time program construction</td>
<td>handles unforeseen needs, end-</td>
</tr>
<tr>
<td>interactive, dynamic nature</td>
<td>user coding</td>
</tr>
<tr>
<td>access to interpreter information</td>
<td>incremental development and</td>
</tr>
<tr>
<td>wide portability</td>
<td>testing</td>
</tr>
<tr>
<td>compilation to portable byte-code</td>
<td>cross-platform programming</td>
</tr>
<tr>
<td>built-in interfaces to external</td>
<td>without ports</td>
</tr>
</tbody>
</table>

Python
• elements from C++, Modula-3
  (modules), ABC, Icon (slicing)
• same family as Perl, Td, Scheme, REXX, BASIC dialects
Uses of Python

- shell tools
  - system admin tools, command line programs
- extension-language work
- rapid prototyping and development
- language-based modules
  - instead of special-purpose parsers
- graphical user interfaces
- database access
- distributed programming
- Internet scripting

What not to use Python (and kin) for

- most scripting languages share these
- not as efficient as C
  - but sometimes better built-in algorithms (e.g., hashing and sorting)
- delayed error notification
- lack of profiling tools

Using python

- /usr/local/bin/python
  - #! /usr/bin/env python
- interactive use

```
#!/usr/local/bin/python
# import systems module
import sys
marker = ':::::,
for name in sys.argv[1:]:
    input = open(name, 'r')
    print marker + name
    print input.read()
```

Python structure

- modules: Python source files or C extensions
  - import, top-level via from, re, os
- statements
  - control flow
  - create objects
  - indentation matters – instead of {};
- objects
  - everything is an object
  - automatically reclaimed when no longer needed

First example

```
#!/usr/local/bin/python
# import systems module
import sys
marker = ':::::,
for name in sys.argv[1:]:
    input = open(name, 'r')
    print marker + name
    print input.read()
```

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'
  - continuation w\a or use \n\n\n
```
**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
- immutable: cannot assign to subscript

---

**Lists**

- lists can be heterogeneous
  - `a = ['spam', 'eggs', 100, 1234, 2**42]`
- lists can be indexed and sliced:
  - `a[0]` → spam
  - `a[2:]` → ['spam', 'eggs']
- lists can be manipulated
  - `a[0:2] = [1, 12]`
  - `a[0:0] = []`
  - `len(a) → 5`

---

**Basic programming**

```python
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**

```python
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**

```python
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` and `continue` like C
- `else` after loop exhaustion
  ```python
  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'
  ```
Do nothing

- pass does nothing
- syntactic filler

while 1:
    pass

Defining functions

```python
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(1000)
```

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

Functions: default argument values

```python
def ask_ok(prompt, retries=4, complaint='Yes or no, please! '):
    while 1:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return 1
        if ok in ('n', 'no'): return 0
        retries = retries - 1
        if retries < 0: raise IOError,
            'refusnik error'
        print complaint
    >>> ask_ok('Really? ')
```

Keyword arguments

- last arguments can be given as keywords

```python
def parrot(voltage, state='a stiff', action='voice', type='Norwegian Blue '):
    print """-- This parrot wouldn't", action,
    print """if you put", voltage, "Volts through it."
    print """Lovely plumage, the ".", type
    print """-- It's", state, "!"
    parrot(1000)
    parrot(action='VOOM', voltage=100000)
```

Lambda forms

- anonymous functions
- may not work in older versions

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

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

List methods

- append(x)
- extend(i)
    - append all items in list (like Td lappend)
- insert(i, x)
- remove(x)
- pop([i]), pop()
    - create stack (FIFO), or queue (LIFO) \rightarrow 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

```python
>>> 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

- can also use `if`:
  ```python
  >>> [3*x for x in vec if x > 3]
  [12, 18]
  >>> [3*x for x in vec if x < 2]
  []
  ```

List comprehensions

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

Del - removing list items

- remove by index, not value
- remove slices from list (rather than by assigning an empty list)
  ```python
  >>> a = [-1,1,6.6,333,333,1234.5]
  >>> del a[0]
  >>> a
  [1, 6.6, 333, 333, 1234.5]
  >>> del a[2:4]
  >>> a
  [1, 6.6, 1234.5]
  ```
Tuples and sequences

- lists, strings, **tuples**: examples of **sequence** type
- tuple = values separated by commas
  >>> t = 123, 543, 'bar'
  >>> t[0]
  123
  >>> t
  (123, 543, 'bar')

Tuples

- Tuples may be nested
  >>> u = t, (1,2)
  >>> u
  ((123, 542, 'bar'), (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',

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']
- use has_key() to check for existence
  >>> tel.has_key('foo')
  0
**Conditions**

- can check for sequence membership with `is` and `is not`:
  ```python
  >>> if 4 in vec:
      ... print '4 is'
  ...  
  chained comparisons: a less than b AND b equals c:
  ```
- a < b == c
- and or are short-circuit operators:
  - evaluated from left to right
  - stp evaluation as soon as outcome clear

**Comparing sequences**

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

**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`

**Conditions**

- Can assign comparison to variable:
  ```python
  >>> s1, s2, s3 = '', 'foo', 'bar'
  >>> non_null = s1 or s2 or s3
  >>> non_null = foo
  ```
- Unlike C, no assignment within expression

**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

- function definition + executable statements
- executed only when module is imported
- modules have private symbol tables
- avoids name clash for global variables
- accessible as `module_globalname`
- can import into name space:
  ```python
  >>> from fibo import fib, fib2
  >>> fib(500)
  ```
- can import all names defined by module:
  ```python
  >>> from fibo import *
  ```

Module search path

- current directory
- list of directories specified in PYTHONPATH environment variable
- uses installation-default if not defined, e.g., `:/usr/local/lib/python`
- uses `sys.path`
  ```python
  >>> import sys
  >>> sys.path
  ['./', 'C:\Program Files\python2.2', 'C:\pyth2.2\lib\site-packages']
  ```

Compiled Python files

- include byte-compiled version of module if there exists `fibo.pyc` in same directory as `fibo.py`
- only if creation time of `fibo.pyc` matches `fibo.py`
- automatically write compiled file, if possible
- platform independent
- doesn’t run any faster, but `loads` faster
- can have only `.pyc` file → hide source

Standard modules

- system-dependent list
- always `sys` module
  ```python
  >>> import sys
  >>> sys.p1
  '...
  >>> sys.p2
  '...
  >>> sys.path.append('some/directory')
  ```

Module listing

- use `dir()` for each module
  ```python
  >>> dir(fibo)
  ['__doc__', '__name__', '__file__', 'fib', 'fib2']
  >>> dir(sys)
  ['__builtins__', '__doc__', '__import__', '__loader__', '__name__', '__package__', '__pyx_cache__,', '__pyx_capi__', '__pyx_code__', '__pyx_fused_manager>', '__pyx_function_manager', '__pyx_gensverständ', '__pyx_key_cache', '__pyx_key_cache_len', '__pyx_key_cache_version', '__pyx_all_implicit_namespaces__']
  ```

Classes

- mixture of C++ and Modula-3
- multiple base classes
- derived class can override any methods of its base class(es)
- method can call the method of a base class with the same name
- objects have private data
- C++ terms:
  - all class members are public
  - all member functions are virtual
  - no constructors or destructors (not needed)
**Classes**

- Classes (and data types) are objects
- Built-in types cannot be used as base classes by user
- Arithmetic operators, subscripting can be redefined for class instances (like C++, unlike Java)

**Class definitions**

Class `ClassName`:

```python
<statement-N>
...
<statement-N>
```

- Must be executed
- Can be executed conditionally (see Tcl)
- Creates new namespace

**Namespaces**

- Mapping from name to object:
  - Built-in names (abs())
  - Global names in module
  - Local names in function invocation
- Attributes = any following a dot
  - z.real, z.imag
- Attributes read-only or writable
  - Module attributes are writeable

**Namespaces**

- Scope = textual region of Python program where a namespace is directly accessible (without dot)
  - Innermost scope (first) = local names
  - Middle scope = current module's global names
  - Outermost scope (last) = built-in names
- Assignments always affect innermost scope
  - Don't copy, just create name bindings to objects
- Global indicates name is in global scope

**Class objects**

- `obj.name` references (plus module!):
  
  ```python
class MyClass:
    "A simple example class"
    i = 123
    def f(self):
        return 'hello world'

>>> MyClass.i
123
>>> MyClass.f

MyClass.f is method object
```

**Class objects**

- Class instantiation:
  ```python
  >>> x = MyClass()
  >>> x.f()
  'hello world'
  ```
- Creates new instance of class
  ```python
  >>> note x = MyClass vs x = MyClass()
  ```
  ```python
  >>> __init__special method for initialization of object
  ```
  ```python
def __init__(self, realpart, imagpart):
    self.r = realpart
    self.i = imagpart
  ```
**Instance objects**

- attribute references
- data attributes (C++/Java data members)
  - created dynamically
  - `x.counter = 1`
  - while `x.counter < 10`:
    - `x.counter = x.counter * 2`
    - `print x.counter`
    - `del x.counter`

**Method objects**

- Called immediately:
  - `x.f()`
- can be referenced:
  - `xf = x.f`
  - while `l`:
    - `print xf()`
- object is passed as first argument of function → 'self'
  - `x.f()` is equivalent to `MyClass.f(x)`

**Notes on classes**

- Data attributes override method attributes with the same name
- no real hiding → not usable to implement pure abstract data types
- clients (users) of an object can add data attributes
- first argument of method usually called `self`
  - 'self' has no special meaning (cf. Java)

**Another example**

- `bag.py`
  - class `Bag`:
    - `def _init_(self)`:
    - `self.data = []`
    - `def add(self, x)`:
      - `self.data.append(x)`
    - `def addOne(self, x)`:
      - `self.add(x)`
      - `self.add(x)`

**Another example, cont'd.**

- `invoke`:
  - `>>> from bag import *`
  - `>>> 1 = Bag()`
  - `>>> 1.add('first')`
  - `>>> 1.add('second')`
  - `>>> 1.data`
    - `['first', 'second']`

**Inheritance**

- `class derivedClassName(baseClassName)`
  - `<statement> ->`
  - `<statement> ->`
  - search class attribute, descending chain of base classes
  - may override methods in the base class
  - call directly via `BaseClassName.method`
Multiple inheritance

```python
class DerivedClass(Base1, Base2, Base3):
    <statement>
    # depth-first, left-to-right
    # problem: class derived from two classes
    # with a common base class
```

Private variables

- No real support, but textual replacement (name mangling)
- `__var` is replaced by `__classname_var`
- prevents only accidental modification, not true protection

~ C structs

- Empty class definition:
  ```
  class employee:
  pass
  
  john = employee()
  john.name = 'John Doe'
  john.dept = 'CS'
  john.salary = 1000
  ```

Exceptions

- syntax (parsing) errors
  ```
  File "stdin", line 1
  while 1 print 'Hello world' 
  SyntaxError: invalid syntax
  ```
- exceptions
  ```
  run-time errors
  e.g. ZeroDivisionError, NameError, TypeError
  ```

Handling exceptions

```python
while 1:
    try:
        x = int(raw_input("Please enter a number: "))
    except ValueError:
        print "Not a valid number"
    # First, execute try clause
    # if no exception, skip except clause
    # if exception, skip rest of try clause and use except clause
    # if no matching exception, attempt outer try statement
```
# Language comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>TO</th>
<th>Perl</th>
<th>Python</th>
<th>JavaScript</th>
<th>Visual Basic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>regexp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breadth</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extensive</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>embeddable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>easy GUI</td>
<td>✓</td>
<td>✓</td>
<td>✓ (TM)</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>net/web</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Enterprise</td>
<td>cross-platform</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>SQL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>thread-safe</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>database access</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>