Scripting languages

Typically a language used for short programs to manage other programs.
Interpreted, dynamically typed, permissive semantics
Usually minimal declarations
Usually rich set of string operations (the ultimate untyped data)
Easy interface to OS, file and directory manipulation
Specialized control structures:
  – regular expressions (Perl)
  – dictionaries (Python)
Python: a modern hybrid

A language for scripting and prototyping
Balance between extensibility and powerful built-in data structures

- genealogy:
  - ABC (Amsterdam, Meertens et al. 1980-)
  - Python (Van Rossum et all. 1996-)

- Very active open-source community
Prototyping

- interactive (like LISP, ML, etc).
- minimal translation to bytecode (like Java)
- dynamic typing (like LISP, SETL, APL)
- higher-order functions (LISP)
- garbage-collected, no pointers (LISP, etc.)
- Uniform treatment of indexable structures (like SETL)
- Built-in **associative structures** (like SETL)
- Light syntax, indentation is significant (from ABC)
No one will look at it without OOP

Simple model of modules and classes
inheritance of implementation
No type declarations, so interface inheritance as well
multiple inheritance
No information-hiding
simple visibility model
clumsy mechanism for operator overloading
limited nesting:
   built-in scope, global scope, local scope
What is looks like

rulers = { 'france': ['chirac', 2000, 7],  # general mapping
    'us': ['clinton', 1996, 4],
    'peru': ['toledo', 2000, 6],
    'romania': ['illiescu', 2000, 5]}

for country in rulers.keys():  # built-in iterators
    [pres, elected, term] = rulers[country]  # assignment
    if 2005 - elected <= term:
        print country, "：“, pres "has %I years to go” % (term - (2005 - elected))
    else:
        print country, "：“, pres, "is out of office"
Simple interactive model

python pres.py # load and execute

france: chirac has 2 years to go
us: clinton is out of office
romania: illiescu has 0 years to go
peru: toledo has 1 years to go

can also write
python

>>> import pres # load, execute, continue
Uniform treatment of indexable data

Strings, lists and arrays have common operations
characters are strings of length 1
name = “Python”;
courses = [“languages”, “compilers”] + [“databases”, “basketry”];
coordinates = (0.0, 1.5, -4.5, 2.0);

• indexing from 0
• negative index: indexing from end
name [-2] is “o”, courses [-3] is “compilers”
if ix is negative, lis [ix] is lis [len (lis) + ix]
Tuples and parallel assignment

\[ T = [(1, 2), (3, 4), (5, 6)]; \]

\[
\text{for } (a, b) \text{ in } T:\quad \# \text{ both } a \text{ and } b \text{ are bound}
\]

\[
\text{print } a + b
\]

Yields

3
7
11

Wherever a variable can appear, a tuple of names can appear, recursively
Slicing (every which way) and iterating

slicing:  s[m:n]

    from mth component, up to but excluding nth
s [m:]  to end,
s[:n]  from beginning,
s[:]  all components
s * 4  repetition

built-in iterators:
•  for c in name:  # c bound to each char
for course in courses:
Dictionaries

domain (keys) of arbitrary types
retrieval by key:

    rulers ['peru'] yields ['toledo', 2000, 6]

assignment / modification

>>> rulers ['peru'][2] = 10 # coup: another 8 years to go!

>>> rulers ['mexico'] = ['fox', 2000, 6]

>>> rulers ['khirgizstan'] = [] # no type consistency required
Set theory as a model of computation

Alternative to lists + recursion: sets + membership + iterators

set constructors in SETL:

\[
S2 := \{ f(x) : x \text{ in } S \mid P(x) \};
\]

in Python:

```python
S2 = [ ];
for x in S:
    if P(x):
        S2.append (f(x));
```
Loops

```python
for x in L:  # uniform syntax over composites
Iterators over dictionaries
  for k in mydict.keys(): ...
Explicit iterators:
  for x in [1, 1, 2, 3, 5, 8, 13]:
Numeric iterators
  for x in range(1, 100):
```
def intersect (seq1, seq2):
    res = [ ]
    for x in seq1:
        if x in seq2:
            res.append (x)
    return res

• assigned names are local unless declared global
• no possible hiding

# no type info
# initialize list
# iterate over list
# built-in membership
# in-place modification
Modules

Modules are namespaces: unit of encapsulation

Modules are objects: components can be accessed

Modules can be inspected dynamically:

```python
__dict__ provides dictionary for module:
keys are strings for entities

for attr in module.__dict__.keys():
    print attr,
    if attr [0:2] == "__":
        print attr, "built-in name"
```

# look at all entities
# comma prevents LF
# naming convention
Classes and inheritance

Standard notions: superclasses, derived classes, self (for this), dynamic dispatching

Each class and each object is a namespace with a dictionary

To locate an operation, lookup in dictionary of object (dispatch table). If not found, examine superclasses.

Operator overloading through predefined names:

__init__ constructor
__del__ destructor
__add__ operator “+”
__repr__ printing, external representation
Data members are created implicitly

```python
def __init__(self, start):
    self.data = start;  # data is defined

def __add__(self, other):
    # number + number
    return Number (self.data + other.data)

def __repr__(self):
    return `self.data`  # convert to string

• note: no way to overload (Number + integer) etc.
```
Any class can be a collection

class collection:
    def __getitem__ (self, i):
        return self.data[i]  # attribute data is indexable

X = collection ( );
X.data = [1, 2, 3]  # member exists, assignment ok
for item in X:
    print item << item  # equivalent to item * 2^item
Classes and methods are objects

- **class widget:**
  
  ```python
  def doit (self, message):
      print message
  Gizmo1 = widget ( );
  Gizmo2 = widget ( );
  ```

- **def factory (aClass, *args):**  
  # class parameter
  ```python
  return apply (aClass, args);
  thing = factory (widget);
  doer = thing.doit;
  doer ("show it");  
  ```  
  # self is already bound
Exceptions, etc.

```python
i = 0;
try:
    while 1:
        item = getitem (self, i)
        ...
        i = i +1
    except IndexError:
        pass
except:
    print "unexpected chaos!"
```

# no boolean type
# body of loop
# eventually i too large
# null statement
# all other exceptions
PERL

A very successful “scripting” language, originally for systems administration
Design influenced by shell programming, C, Lisp, SETL (?), mutual influence with Python
Regular expressions and rich string primitives for text and file manipulation
Compact, often cryptic syntax, has fervent devotees
Fully dynamic, weakly typed, text manipulation can easily be used for program generation.
A simple type model

Atomic types:
   Numbers (no distinction) strings
Standard operations, value semantics
$ used as qualifier on variable names to designate scalars.
   $count = 15;
   $count++;
   $fullname = first . Last;  # concatenation
   $count = "count";         # dynamic typing.
Arrays and Lists

Array is single composite type, list is corresponding value
Standard indexing operations
Multiple assignments for decomposition
@ used as qualifier to denote array variables.
@names = ("bach", "mozart", "springsteen");
($johann, $wolfgang, $bruce) = @names;
$names [1] is "mozart" (scalar)

Permissive semantics on sizes. Undef value plays critical role
Hashes

General mappings, i.e. ordered pairs to represent key=>value associations

Easy conversions from/to lists

%currency = (“us” => “dollar”, “china” => “yuan”, “eu” => “euro”, “japan” => “yen”);

keys %currency yields (“us”, “china”, “eu”, “japan”)
    (in some order)

Iteration, point-wise assignment and retrieval:

while ($country, $coin) = each %currency) {
...

$currency{“mexico”} = “peso”;
Context-dependent coercions

In a scalar context, a list is coerced to yield a scalar
$n = \@L$; # $n$ receives the size of $\@L$
$\@L2 = \@L1$ # $\@L2$ is a copy of $L1$

Single-quoted strings are constant.

print 'this has no new line 
'

Double-quoted strings are subject to variable substitution:

print “expands $variable and special chars \t\n”
Regular expressions

A well-known formalism for text manipulation
Notation from automata theory, augmented with pattern-matching primitives

\s(\w+)/  is a pattern that matches:
  A space:  \s
  Followed by 1 or more letters:  \w +
  Followed by a comma: ,

/(\S+), (\S+), (\S+)/  matches three sequences of characters, comma separated, that don't include spaces
  \S is the negation of \s

Matched strings are retrieved through $1, $2, etc.