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 al. 1996-)
- Very active open-source community
Prototyping

- Emphasis on experimental programming:
  - 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

```python
rulers = {'france': ['chirac', 1995, 7],
          'us': ['clinton', 1996, 4],
          'peru': ['fujimori', 1998, 0],
          'romania': ['illiescu', 2000, 5]}

for country in rulers.keys():
    [pres, elected, term] = rulers[country]  # assignment
    if 2002 - elected <= term:
        print country, "has %I years to go" %
        (term - (2002 - elected))
    else:
        print country, "is out of office"
```

Simple interactive model

```
$ python pres.py # load and execute
france: chirac has 0 years to go
us: clinton is out of office
romania: illiescu has 3 years to go
peru: fujimori is out of office
```

- 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
- \texttt{name} = "Python";
- \texttt{courses} = ["languages", "compilers"] + ["databases", "basketry"];
- \texttt{coordinates} = (0.0, 1.5, -4.5, 2.0);
- Indexing from 0
- Negative index: indexing from end
- \texttt{name} [-2] is “o”, \texttt{courses} [-3] is “compilers”
- If \texttt{ix} is negative, \texttt{lis [ix]} is \texttt{lis [len (lis) + ix]}

Tuples and parallel assignment

\texttt{T} = [(1, 2), (3, 4), (5, 6)];
\texttt{for (a, b) in T:} \quad \# both a and b are bound
\texttt{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 \( m \)th component, up to but excluding \( n \)th
- \( s[m:] \) 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

- General-purpose associative maps
- domain (keys) of arbitrary types
- retrieval by key:
  - rulers ['peru'] yields ['fujimori', 1998, 0]
- assignment / modification
  >>> rulers ['peru'][2] = 10 # coup: another 8 years to go!
  >>> rulers ['mexico'] = ['fox', 2000, 6]
  >>> rulers ['pakistan'] = [] # no type consistency required
Set theory as a model of computation

- Alternative to lists + recursion: sets + membership + iterators
- set constructors in SETL:
  \[ S_2 := \{ f(x) : x \in S \mid P(x) \}; \]
- in Python:
  \[
  S2 = [ ];
  for x in S:
    if P(x):
      S2.append (f(x));
  \]

Loops

- Iterators over collections:
  \[
  for x in L:
  \]
- Iterators over dictionaries
  \[
  for k in mydict.keys( ) ... 
  \]
- Explicit iterators:
  \[
  for x in [1, 1, 2, 3, 5, 8, 13]:
  \]
- Numeric iterators
  \[
  for x in range (1,100):
  \]
Functions

def intersect(seq1, seq2):
    res = []                                # initialize list
    for x in seq1:                        # iterate over list
        if x in seq2:                      # built-in
            res.append(x)             # in-place
            return res

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

Modules

- Modules are namespaces: unit of encapsulation
- Modules are objects: components can be accessed
- Modules can be inspected dynamically:
  - __dict__ provides dictionary for module:
  - keys are strings for entities
    for attr in module.__dict__.keys(): # look at all entities
        print attr,                      # comma prevents LF
        if attr[0:2] == "__":            # naming convention
            print attr, "built-in name"
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

Class Number:
```python
def __init__(self, start):
    self.data = start; # data is defined
def __add__(self, other):
    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:  # for calls
    __getitem__
        print item <<= item  # equivalent to item * 2 ^ item

Classes and methods are objects

- class widget:
  - def doit(self, message):
    - print message
  - Gizmo1 = widget();
  - Gizmo2 = widget();
  - def factory(aClass, *args):  # class parameter
    - return apply(aClass, args);
  - thing = factory(widget);
  - doer = thing.doit;
  - doer("show it");  # self is already bound
Exceptions, etc.

- internally, iterator implemented as
  
  ```python
  i = 0;
  try:
    while 1:                                           # no boolean
      item = getitem (self, i)
      ...                                           # body of loop
      i = i +1
      except IndexError:                           # eventually i too
        pass                                         # null statement
      except:                                             # all other
        print "unexpected chaos!"
  ```