Scripting Languages

Fast development, extensible programs

Devert Alexandre
School of Software Engineering of USTC

Table of Contents

Introduction

You probably heard about those programming languages

- Javascript
- Perl
- Python
- Lua
- Ruby
- ...

Scripting languages

They are *scripting languages*, sharing common features

- Highly dynamic
- Automatic memory management
- Can compile to a virtual machine
- Can be *embedded*
- Can be extended with modules written in *C* or *C++*
- Cross-platform

Scripting languages

Scripting languages tends to help you to code *faster*

- less code to do the same things with other languages
- powerful default types
- very dynamic languages
- large, complete standard library

Usages

They are successful for a number of applications

- tool-making languages
- "glue" languages to write "glue code"
- building extensible, open-ended software
- ...

Table of Contents

A Python tour

Let's have a look at a scripting language, Python. What follows is true for most other scripting languages.

Friendly syntax

List and dictionaries are parts of the language syntax

```
a = [1, 2, 3, 4, 5, "six", 7]

b = { 'name' : 'apple', 'color' : 'red', 'cost' : 0.25 }

c = [[1.5, 3.0], [-2.0, 5.0], [-9.7, 5.2]]

d = {
  'fruits'   : ['apple', 'orange', 'peach'],
  'location' : 'Suzhou'
}
```

Friendly syntax

List manipulations are part of the languages

```
a = [1, 2, 3, 4, 5, 7, 8, 9, 10]

print a[2:7]
>>> [3, 4, 5, 7, 8]

print a[2:]
>>> [3, 4, 5, 7, 8, 9, 10]

print a[:5]
>>> [1, 2, 3, 4, 5]

print a[2:5] + a[7:]
>>> [3, 4, 5, 9, 10]

print [2, 3] * 5
>>> [2, 3, 2, 3, 2, 3, 2, 3, 2, 3]
```

Friendly syntax

Iterators are nearly invisible

```
a = [1, 2, 3, 4, 5]

for value in a:
  print value
```

Iterating over the elements of a list

Friendly syntax

Iterators are nearly invisible

```
myFile = open("input.txt")

i = 0
for line in myFile:
  i += 1
  print i, line
```

Iterating over the lines of a text file

Friendly syntax

You affect multiple variable in one statement

```
a, b, c = 1, 2, 3
print a, b, c

def myFunction(x):
  return x, x * x

a, b = myFunction(3)
print a, b
```

Dynamic variable type

Variable types is not fixed, it changes

```
a = 42
a = "apple"
a = ["wang", "fei", "yue"]
a = (1, 2, 3, "four", 5.0)
```

Everything is object

Numbers are objects

```
a = 2
print a + 3
print a.__add__(3)
```

Everything is object

Lists are objects

```
a = [1, 2, 3, 4, 5]
print a[3]
print a.__getitem__(3)
```

Everything is object

Functions are objects

```
def sqr(x):
  return x * x

print sqr(3)
```

```
class sqrFunc(object):
  def __call__(self, x):
    return x * x

sqr = sqrFunc()
print sqr(3)
```

Standard library

An extensive standard library

- OS-independent file and directory access
- data persistence
- XML, JSON, CSV, . . . parsing
- most common Internet protocols
- OS-independent GUI
- . . .

Standard library

Example: defining a command-line interface

```
import argparse


cmdLine = argparse.ArgumentParser(description = 'Read things and do stuffs')
cmdLine.add_argument('inputPath',
                     action = 'store',
                     help   = 'path to input file',
                     type   = str)
cmdLine.add_argument('-s', '--size',
                     action  = 'store',
                     dest    = 'fontSize',
                     default = 16,
                     help    = 'size of read buffer',
                     type    = int)

args = cmdLine.parse_args()
```

Standard library

Example: printing the content of a ZIP archive

```
import zipfile
import argparse



cmdLine = argparse.ArgumentParser(description = 'List the content of a ZIP archive')
cmdLine.add_argument('inputPath',
                     action = 'store',
                     help   = 'path to input ZIP archive',
                     type   = str)

args = cmdLine.parse_args()

try:
  archive = zipfile.ZipFile(args.inputPath, 'r')
  for info in archive.infolist():
    print info.filename, info.date_time, info.file_size, info.compress_size
except zipfile.BadZipfile:
  print args.inputPath, "is not a ZIP file"
except IOError as (errno, strerror):
  print strerror
```

Standard library

Example: reading/writing JSON data

```
import json

a = json.load(open('example.json'))
print a
```

```
import json

a = [
  'foo',
  {
    'bar': ('baz', None, 1.0, 2)
  }
]

json.dumps(a)
```

Fibonacci numbers

Fibonacci numbers are defined as follow

$$F_n = F_{n-1} + F_{n-2}, F_0 = 0, F_1 = 1$$

Fibonacci numbers

Let's code a function to compute Fibonacci numbers

```
def fib(n):
  if n < 2:
    return 1
  return fib(n - 1) + fib(n - 2)

print fib(36)
```

On my computer, this takes about 11 sec. to compute

Memoization

To make it faster, we should store intermediate results.
It's called *memoization*

```
fib_memo = { }

def fastFib(n):
  if n < 2:
    return 1

  if not fib_memo.has_key(n):
    fib_memo[n] = fastFib(n - 1) + fastFib(n - 2)

  return fib_memo[n]

print fastFib(36)
```

Same function, but takes 80 msec. to compute !

Memoization

But we might not be so happy of that solution

- mix the idea and the implementation
- have to do by hand this for any expensive recursive function

Functions are objects

But remember, functions are objects ! $\Rightarrow$ we can build a function object which

- contains a function f and a dictionary m
- if a not in m, $m[a] = f(a)$
- returns $m[a]$

Memoizing functions

Python implementation for this (works for all arguments)

```
class Memoize:
  def __init__(self, f):
    self.f = f
    self.m = { }

  def __call__(self, *args):
    if not self.m.has_key(args):
      self.m[args] = self.f(*args)
    return self.m[args]
```

Memoizing functions

And now, we have a general and clean way to do *memoization*

```
def fib(n):
  if n < 2:
    return 1
  return fib(n - 1) + fib(n - 2)

fib = Memoize(fib)
print fib(36)
```

Monkey patching

Methods of objects are functions too

```
class Animal:
  def makeNoise(self):
    print 'not defined'

def doGroaaaar():
  print 'groaaaar !'



tiger = Animal()
tiger.makeNoise = doGroaaaar

tiger.makeNoise()
```

Dynamic replacement of methods

Monkey patching

Monkey-patching allow to write very flexible and short code

```
import random

class Animal:
  def makeNoise(self):
    print 'not defined'

class RandomNoiseMaker:
  def __init__(self, noiseList):
    self.noiseList = noiseList

  def __call__(self):
    print random.choice(self.noiseList)



tiger = Animal()
tiger.makeNoise = RandomNoiseMaker(['groaaar !', 'grrrrr !', 'graaaaouuu !'])

tiger.makeNoise()
```

Monkey patching with a dynamically defined function

Table of Contents

Languages comparison

Let's compare *Python* and *Java* to do common tasks

Languages comparison

Printing “Hello, world !”

```java
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
}
```

```
print "Hello, world!"
```

Languages comparison

Opening a file

```
import java.io.*;
...

BufferedReader myFile =
    new BufferedReader(
        new FileReader(argFilename));
```

```
myFile = open(argFilename)
```

Languages comparison

Add an integer to a list, get an other from the list

```java
public Vector<Integer> aList =
  new Vector<Integer>;
public int aNumber = 5;
public int anotherNumber;

aList.addElement(aNumber);
anotherNumber = aList.getElement(0);
```

```
aList = []
aNumber = 5

aList.append(aNumber)
anotherNumber = aList[0]
```

Languages comparison

Print a list of numbers to a file

```java
import java.io.*;

public class IOTest {
  public static void main(String[] args) {
    try {
      File f = new File("out.txt");
      PrintWriter ps =
        new PrintWriter(new OutputStreamWriter
          (new FileOutputStream(f)));

      for(int i = 0; i < 1000000; i++)
        ps.print(String.valueOf(i));

      ps.close();
    }
    catch(IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

```python
f = open('out.txt','wb')

for i in xrange(1000000):
  f.write(str(i))

f.close()
```

Languages comparison

Defining a simple class

```
public class Employee {
  private String myEmployeeName;
  private int    myTaxDeductions = 1;
  private String myMaritalStatus = "single";

  public Employee(String EmployeName) {
    this(employeeName, 1);
  }

  public Employee(String EmployeName, int taxDeductions) {
    this(employeeName, taxDeductions, "single");
  }

  public Employee(String EmployeName, int taxDeductions, String maritalStatus) {
    this.employeeName    = employeeName;
    this.taxDeductions   = taxDeductions;
    this.maritalStatus   = maritalStatus;
  }
...
```

Languages comparison

Defining a simple class

```
class Employee():
  def __init__(self, employeeName, taxDeductions=1, maritalStatus="single"):
    self.employeeName    = employeeName
    self.taxDeductions   = taxDeductions
    self.maritalStatus   = maritalStatus
...
```

Table of Contents

Tools

It's quite common to have to do code which will not be in the final product

- Build
- Test
- Content generation
- Source-code generation

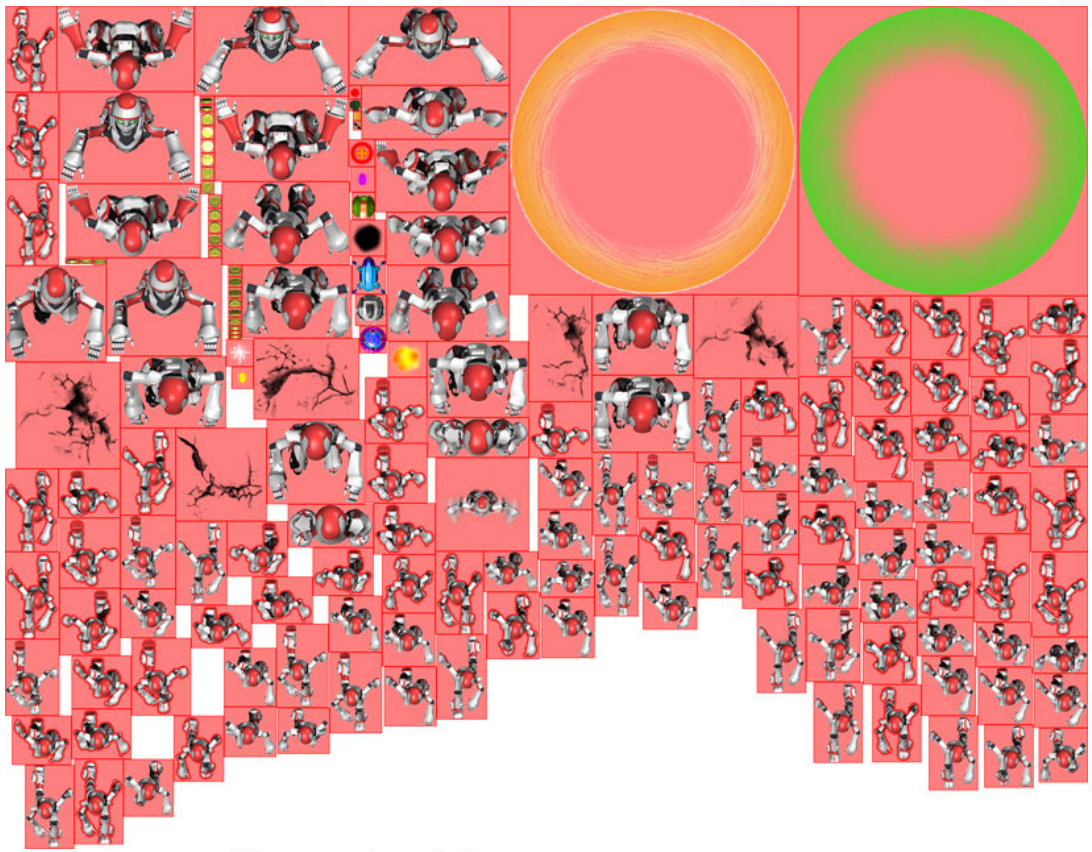
Content generation often need project-specific tools

Texture atlas

In games, graphics are usually made from lot of pictures

Texture atlas

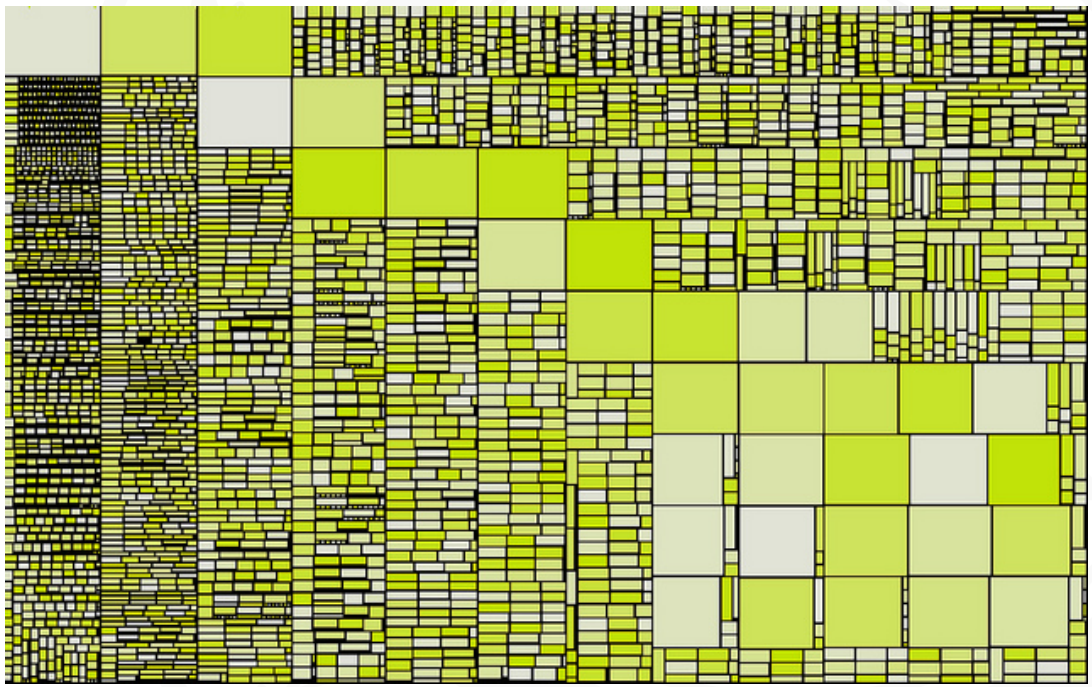
Save speed & memory ⇒ pack all graphics in one texture



On mobile devices, it is *essential*

Texture atlas

A texture packing many graphics elements is called a *texture atlas*



Building a *texture atlas* is very device and project dependent

Texture atlas

Automated texture atlas generation

- a *big* time saver
- artist can test many ideas
- does not need to be fast
- integration to build process

Texture atlas

Texture atlas generator should

1. read a list of pictures
2. read a list of rectangle coordinates
3. pack rectangles
4. generate rectangle coordinates
5. generate a picture

Texture atlas

Texture atlas generator as a script

- read/write pictures with one line of code
- read/write XML, JSON, plain text with very few code
- rectangle packing algorithm

Personal experience $\Rightarrow$ 1 hour in Python, 12 hours in C

Tool making

Scripting languages are *wonderful* to build such tools

Table of Contents

Glue code

Software are often built by *gluing* together various libraries

- project specific code
- some specialized processing libraries
- a communication library
- a data storage library (database, XML, ...)
- a GUI library

Glue code

Many important libraries have Python, Ruby, Lua or Perl *bindings*

- *Sqlite*, *MySql*, *Berkeley DB* $\Rightarrow$ database
- *wxWindow*, *QT*, *GTK* $\Rightarrow$ graphic users interfaces
- *Numpy*, *Scipy* $\Rightarrow$ math and linear algebra
- *Simple Direct Media Layer* $\Rightarrow$ basic graphic, sound and inputs handling
- *Ogre3D* $\Rightarrow$ 3d graphic engine
- *Cairo* $\Rightarrow$ 2d vector graphics
- ...

Glue code

A way to build cross-platform software quickly $\Rightarrow$ use a scripting language to *glue* libraries

- Glue code usually goes very well with very dynamic languages
- Fast prototyping
- Good performance (most bindings are coded in *C* or *C++*)

Success stories

Some successful applications using this approach

- *Python* $\Rightarrow$ *DropBox*, *Civilization IV*, *Mercurial*, *Django*, *Blender*, *BitTorrent*
- *Ruby* $\Rightarrow$ *Ruby On Rails*, *Google Sketchup*
- *Erlang* $\Rightarrow$ *Wings3D*, *CouchDB*, *Goldman Sachs* robot traders

Project specific code

You might want to create your own extensions for a scripting language

- using legacy code
- for the 10% code which takes 90% of the time

Project specific code

All scripting languages provides a way to create extension in *C/C++*

Writing Python extensions

A "Say hello" Python extension

```
#include <Python.h>

static PyObject*
say_hello(PyObject* self, PyObject* args) {
  const char* name;

  if (!PyArg_ParseTuple(args, "s", &name))
    return NULL;

  printf("Hello %s!\n", name);

  Py_RETURN_NONE;
}

static PyMethodDef HelloMethods[] = {
  { "say_hello", say_hello, METH_VARARGS, "Greet somebody." },
  { NULL, NULL, 0, NULL }
};

PyMODINIT_FUNC

inithello(void) {
  (void) Py_InitModule("hello", HelloMethods);
}
```

Writing Python extensions

The "Say hello" Python extension setup

```
from distutils.core import setup, Extension

module1 = Extension('hello', sources = ['hellomodule.c'])

setup(name = 'PackageName',
      version = '1.0',
      description = 'This is a demo package',
      ext_modules = [module1])
```

Writing Python extensions

Using the ‘Say hello” extension

```
import hello

hello.say_hello("World")
```

Writing Python extensions

A Fibonacci numbers Python extension

```c
#include <Python.h>

int
_fib(int n) {
  if (n < 2)
    return n;
  else
    return _fib(n-1) + _fib(n-2);
}

static PyObject*
fib(PyObject* self, PyObject* args) {
  const char *command;
  int n;

  if (!PyArg_ParseTuple(args, "i", &n))
    return NULL;

  return Py_BuildValue("i", _fib(n));
}

static PyMethodDef FibMethods[] = {
  {"fib", fib, METH_VARARGS, "Calculate the Fibonacci numbers."},
  {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC
initfib(void) {
  (void)Py_InitModule("fib", FibMethods);
}
```

Writing Python extensions

The Fibonacci Python extension setup

```
from distutils.core import setup, Extension

module1 = Extension('fib', sources = ['fibmodule.c'])

setup(name = 'PackageName',
      version = '1.0',
      description = 'This is a demo package',
      ext_modules = [module1])
```

Writing Python extensions

Using the Fibonacci extension

```
import fib

print fib.fib(10)
```