CS 131 Programming Languages

November 17, 2009

Python and Project Assignment

Twisted is a Python library that is to be used in this assignment. Here's a very high-level overview.

With Java, we've seen thread-based programming where there's a bunch of threads (t_1, t_2, t_3) running in parallel. The code involves synchronization primitives that imply *waiting*. With Twisted, we use event-based programming where there are no threads but there are events which come into the system and trigger event handlers.

e1 → start sending message: *start_sending()*
e2 → set '*deferred'* to represent what to do on reply
e3 → reformat messages, start sending transformed version to someone else

While threads can wait, event handlers can't. None of the event-handler code ever waits, it just uses events to get from one stage of the program to the other. A problem with event handling approach as opposed to the thread approach is this: in thread approach, it is known when the code is waiting and has more work to do. In Twisted approach, '*deferred'* is used to represent work that isn't done yet. This is just like event handlers or event listeners. One nice property of event-based programming is that there's no need for '*synchronized'* methods because every event handler is already synchronized (is running on its own). A bad property is that it's hard to have event-based programs to run on multi-core CPU's.

Messages to the other server have to be sent using a primitive like *start_sending()* because the thread can be blocked at any time (for example, when TCP connection is overwhelmed).

Python

When Python started out, people assumed that it isn't going to work (most programming languages don't work).

The language Basic that was invented in early 1960's and designed as an easier alternative to Fortran (a number of features was taken out, made as easy as possible). Basic introduced interactive programming (type and run). It was designed to be educational and therefore was used to teach programming. The programming constructs that original Basic had were quite simple: *goto*, *for*, *if-then-else*, and *gosub* (calls). In roughly 1980 people in the Netherlands decided to teach students in high schools how to program using Basic. This was back in the era of alphanumeric displays, 24x80 terminals.

Many were opposed to using Basic as a programming instruction tool, and ABC language was introduced in order to replace Basic for education. ABC was object-oriented and used *indenting* as part of its syntax: code not properly indented wouldn't compile correctly. ABC used *dictionaries* which were represented as a set of named objects and were to substitute arrays, hash functions, and other complicated concepts. ABC failed because many were not attracted to learning a language that had no use in real world.

Around then there was a scripting language called Perl that was invented by Larry Wall while sitting in the windowless basement room of a nameless research lab. Larry was frustrated with the shell scripting language deficiencies so he wrote Perl, a scripting language with lots of syntax. The model of Perl is "there's more than one way to do it." People liked having to learn just one scripting language instead of having to learn 2 or 3, so Perl became popular.

In the meantime, one of the leading developers of ABC, G. van Rossum, looked at Perl and was unhappy about its syntax. He took some of the good ideas of Perl and re-did them using ABC syntax, thereby creating Python. Python had scripting, interfaced well to modules in other languages, and retained the syntax and basic ideas of ABC. The name "Python" was chosen to reflect the change in attitude.

Python Examples

```
line = "CS131,29,5.13"
types = [str, int, float]
// we want result to be ["CS131", 29, 5.13]
result = [ty(val) for ty, val in zip(types, line.split(','))]
// ['CS131', '29', '5.13'] → [(str, 'CS131'), (int, '29'), (float, '5.13')] → [str('CS131'), int('29'),
float('5.13')] → ["CS131", 29, 5.13]
```

```
'''this is a
string with
two newlines'''               // multi-line string
'\na\b'                       // backslashes precede special chars
r'\na\b'                      // backslashes are regular chars
'ab%de%s' % (29, 'xyz')       // formatting/substituting
if a == b:
       print a;
else
       print "not a"; x();
       if c == d:
              print "eq"
print ( a + b * c – d
       + get_long_named_function() )        // use parentheses for a line of code that's too long
```

Python doesn't follow BNF because it uses indentation as part of its syntax instead of curly braces, etc. as most other languages.

Python Objects

Every value is an *object*, and it has an *identity*, a *type*, and a *value*. The first two can't be changed while the third can be changed if the object is mutable. `id(o)` returns integer that's the identity of object o, and `type(o)` returns the type object for `o`. Associated with object `o` are attributes (`o.a`) and methods (`o.m(37, 19)`).

Python Storage Management

Fundamentally, Python uses *reference counts* to manage its storage. Every object in Python has a little area that counts the number of pointers to that object. Assignment operations increment the reference count of the object that's on the right side of the equal operator.

What if `a = b` is an assignment with two pointers in it (both `a` and `b` are pointers)? We need to increment the reference count of whatever `b` points at, and decrement reference count of whatever `a` points at (reclaim storage if it reaches `0`). There's a little bug in this algorithm, though. What if we do something like `a[1] = a`? Then we get rid of other pointers to `a`. The reference pointer will reach `1` but won't reach `0` because `a[1]` still points at `a`. This results in un-reclaimable storage. For this reason, modern Python implementations have garbage collector that catches these problems when memory is low. Some other Python implementations don't use reference counts and only have a garbage collector (JPython uses Java garbage collector).

Classes in Python

```
class c(a, b):
…
       def m(self, x, y);
              return self.a + y + x;
…
o = c();
…
o.m();
```

Python uses multiple inheritance. When you look up a method for an object of type `c`, Python searches for that method depth-first left-to-right in the inheritance tree of the class. Inside a class one can define methods (the first argument to a method is the object on which behalf the method is involved).

Python Namespaces

A class is an object and each class has a member called `__dict__` which specifies the names of everything found in that class. Python looks in the dictionary on each method call to figure out what code to run.

Python Modules

Namespaces are also controlled by *modules* (typically a source file). Modules are imported using the `import` keyword. When module is imported (such as `import math`), a few things happen:

1. A new namespace (μ) is created
2. Source code from module file is read and executed in the context of μ.
3. A new name '`math`' is created in the context of caller namespace and bound to μ.

Inside a module, you might see something like this:

```
if __name__ == '__main__':
        // run some test cases
```

The `__name__` is bound to the module name; if invoked at the top level it will be set to `__main__`.

Python Packages

Packages can contain other packages as well as individual modules. `import twisted.internet.http.v1` is an import statement that takes package hierarchy into consideration. The way packages are implemented is usually through directory hierarchy. There's a special file called `__init__.py` that marks the directory as a package and whenever Python consults any member of the package, it reads and executes this file.

Python Types

Duck typing: we do not say what object's type is by knowing *a priori* what it is, we do it by looking at its behavior. Types are relatively informal and are defined by the operations available on objects. For example, *mapping* is a type in Python that lets one put key-value pairs in and take them out. Most common mapping is dictionaries, a built-in type. The rest of Python doesn't care how you implement mapping.

Much of the initial success for Python and its present popularity comes from the good set of built-in types and operations. Here's Python type menagerie:

- *None*: none
- *Numbers*: int, float, complex, …
- *Sequences*: string, list, tuple, xrange, buffer
    - *String* is a sequence of characters, is immutable
    - *List* is an mutable sequence of any kind of object (even length of the list is mutable)
    - *Tuple* is an immutable sequence of anything
    - *Xrange* is an immutable sequence of integers that follow an arithmetic progression
    - *Buffer* is a mutable sequence of characters
- *Mapping*: dictionaries
    - Dictionaries are collections of name-value pairs, names must be immutable