Python case study

Python is an interpreted, dynamically-typed, object-oriented scripting language with a host of built-in data types. See Programming Language Exploration for an overview of the language. It is implemented in C, but in a very object-oriented fashion. The design is a good model for language implementation.

The Python interpreter works by loading a source file or reading a line typed at the keyboard, parsing it into an abstract syntax tree, compiling the tree into bytecode, and executing the bytecode. We will concentrate mainly on how the bytecode is executed, such as how inheritance and environments are implemented.

Parsing

The parsing process is fairly standard. The basic idea is to first convert the input characters into a more abstract representation like name: x, integer: 7, string: "hello", less-than-or-equal, etc. The abstracted characters are called tokens. (There are utilities, such as lex, for automatically generating tokenizers, but Python does not use one.) The tokenization process is fully described in the language reference.

For example, the statement

```python
while(x <= 3):
    f(x)
```

might be tokenized as

```plaintext
keyword: while
left-paren
name: x
leq
int: 3
right-paren
colon
indent
name: f
left-paren
name: x
right-paren
```

Then the tokens are assembled into expressions, statements, function definitions, class definitions, etc. Since function definitions contain statements which contain expressions which may contain nested expressions, and so on, the resulting data structure of tokens is a tree, called a parse tree or abstract syntax tree. The tokens above might be parsed into this tree:
There are standard utilities, such as yacc, which aid in generating parsers, but Python does not use one. Instead, it uses nested DFAs, which is a recursive tokenizer that fills the tree out downward. A similar technique was used in the book *Essentials of Programming Languages*. You can learn more about tokenizing, parsing, and compilation in MIT course 6.035 or from various books such as *Crafting a Compiler in C* or *Compilers: Principles, Techniques, and Tools*.

Once you have a parse tree, you can do type checking, type reconstruction, constant folding, liveness analysis, and many other kinds of optimizations and analysis. Python only uses it for compilation.

**Compilation**

The parse tree is then compiled into bytecode by a recursive walk. For example, a while syntax node contains an expression node for the test and a compound statement node for the body. A while node is compiled into:

```
loop:
    (code for test)
    jump_if_false done
    (code for body)
    jump loop
done:
```

where the test and body nodes are compiled recursively. Virtually all of the compilation rules can be described as rewrites like this. Compilation has the opposite structure of parsing: it flattens the tree, converting it back to bytes.

Bytecode is virtual machine instructions are packed into an array of bytes. The instructions are based on stack manipulation. Some instructions have no arguments and take up one byte, e.g. BINARY_ADD (pop two values from the stack and push their sum), while other instructions have an additional two-byte integer argument, e.g. LOAD_NAME i (push the value of the i-th variable name). The full list of bytecodes is [here](#). Bytecode is paired with a symbol table and constant pool so that names and literals can be referenced by number in the instructions.

The bytecode for the above parse tree is something like:
Python does not perform much optimization on the bytecode, except for speeding-up local variable access. The reasons for this are tied to how the interpreter works and will be discussed later.

This conversion pattern, where the input is abstracted, processed, then specialized, is a common one in many varieties of programs and is also presented in Abstraction and Specification in Program Development.

**Execution**

At face value, the execution algorithm is straightforward: fetch the next instruction, perform the required stack manipulation or, in the case of a jump, reposition the instruction pointer. However, much of the real functionality is hidden in the value objects. For example, there is only one BINARY_ADD instruction, yet Python must do very different things when adding integers, floats, strings, or user-defined objects.

**Value objects**

The trick is to decouple the kinds of values in the language from the interpreter core. This is allows Python to have many built-in data types without a complexity explosion. Each value supports the same interface, some of which is listed here:

- **add(v)**
  - Add yourself to v and return the result as a new object. It corresponds to x + v.

- **cmp(v)**
  - Compare yourself to v and return -1, 0, or 1 (a la strcmp). It corresponds to x == v.

- **repr()**
  - Return a string representation of yourself.

- **getattr(name), getitem(v)**
  - Subscript yourself by name or arbitrary value v. getattr corresponds to x.name and getitem corresponds to x[v].

- **call(args, keywords)**
  - Call yourself with positional arguments args and keyword arguments keywords. It corresponds to x(1, 2, 3, foo = 4, bar = 5).

Each method in this interface has a bytecode counterpart which invokes it, e.g. BINARY_ADD. The stack is simply an array of value objects, which each know how to add themselves. Numerical Python takes advantage of this open-ended design to add multi-dimensional arrays, e.g. matrices, to the language.

New kinds of values can even be defined from within Python. An object with a method called __cmp__ will use that method for comparison instead of the default object comparison method. All methods in the value
interface can be overridden in this way (including the getattr method which is used to look up object methods in the first place).

The built-in value objects are:

Primitives
  Integer/Long
  Float
  String
  File
  Function
Composites
  Tuple
  List
  Dictionary
  Class
  Class instance
  Module

Note that functions are simply values that implement the call method. Thus classes and their instances can just associate names with arbitrary values, i.e. act like dictionaries.

Inheritance

Classes and instances differ slightly from dictionaries, however, in that instances inherit from classes and classes can inherit from other classes. The inheritance is such that changes to classes are immediately visible in their descendants and instances, as illustrated in this example:

class p:
  x = 3

class c(p):
  y = "hello"

i = c()
i.y (prints "hello")
i.x (prints 3)
p.x = 4
c.x (prints 4)
i.x (prints 4)

Thus classes and instances differ from dictionaries in that if a read cannot be resolved, the request is passed to the parent class. Since classes can change at run-time, this makes the inheritance process highly dynamic.
This is a recurring pattern called a **Chain of Responsibility**. The object diagram for the above example is thus:

```
Class
P
  x  3

Class
C
  y  "hello"

Instance
  i
```

Note that if Python hadn't provided inheritance, we could recreate it by providing a `__getattr__` method to do the forwarding. The parent class would be stored explicitly in a variable. This would have the interesting side-effect that inheritance links could be changed at run-time by modifying this variable. The Chain of Responsibility pattern fully supports such changes.

What happens on a write? If requests were forwarded up the Chain, as with a read, there would be no way to override a slot in a parent class. In the above example, `c.x = 5` would be the same as `p.x = 5`. (If the variable turns out to be unbound, then the object written to would gain a new slot.) If requests were not forwarded, then the object written to would always gain a new slot, shadowing the ancestor slot. Python chooses the latter, so that parent methods can be overridden in children. Thus we get:

```
c.x = 5
p.x    (prints 4)
i.x    (prints 5)

i.x = i.x
i.c.x = 6
i.x    (prints 5)
```

Note that `i.x = i.x` performs useful work under this design choice.

Does Python need to distinguish between classes and instances? Both are essentially dictionaries and have the same read/write and inheritance mechanism. Perhaps it's for efficiency reasons: there are usually far fewer classes than instances, so certain optimizations can be applied to classes but not instances.

**Variable scope**

Python variable environments, aka "stack frames", have many of the qualities of objects. They are dictionaries of names with the same kind of inheritance semantics. That is, variables used in a function by default refer to global names, but if a variable is assigned to in the function, it becomes local. The same applies to class definitions.

One can imagine also designing variable scope with a Chain of Responsibility. In fact, some languages do this, even ones which don't have the corresponding notion of inheritance. *Structure and Interpretation of
Computer Programs uses it to describe Scheme's method of variable scope. The Self language actually treats environments as objects and uses inheritance to implement lexical scope!

However, Python does not take this approach, probably for efficiency reasons. Python does not allow more than two simultaneous environments (the global and the local), as noted in the PLE exercise on nested scopes. This allows certain optimizations, like the LOAD_FAST bytecode, but can confuse programmers used to lexical scoping. The following C++ fragment has no equivalent in Python:

```cpp
int x = 1;
if(x > y) {
    int x = 2;
    cout << x;  // prints 2
}
cout << x;  // prints 1
```

Scheme programmers may wonder: then how does Python implement lambda? The escape was to abandon lexical scoping for lambda. The body of a lambda only has the global and local environments.