How Python* Works

Built-in data structures

Compiling and Interpreting Python

*The CPython implementation; there are others
Multiple Implementations

CPython: Python implemented in C
(What we have been using)

Jython: Python implemented in Java
• Uses Java classes in place of Python libraries

Iron Python: Python implemented in C#
• Runs on Microsoft CLR / .net framework

PyPy: Python in Python
• Originally a just-in-time translator; now Python -> C

...
CPython Implementation

Caveats:

Some simplifications
Some guesswork

(there was more guesswork before I found Laurent Luce’s blog: http://www.laurentluce.com/ )
**CPython is a program**

CPython is written in C
Compiled into machine code named “python” or “python3”

Every Python data type is implemented by a C data structure in CPython
The Python Compiler/Interpreter

```
x = 32
if x > 15 :
    y = 32
```

(file: .py)

Lexical analysis

Parser: Syntax Analysis

Bytecode generator

Interpreter

(tokens:)
```
x = 32 ; if x > 15 :
     y = 32 ;
```

(abstract syntax tree:)
```
= x 32 > if
   x 15 y 32
```

(.pyc)

```
2 0 LOAD_CONST 1 (32)
3  STORE_FAST 0 (x)
3 6 LOAD_FAST 0 (x)
9  LOAD_CONST 2 (15)
12 COMPARE_OP 4 (>)
15 POP_JUMP_IF_FALSE 27
4 18 LOAD_CONST 1 (32)
21 STORE_FAST 1 (y)
```

(simplified)

(more on this below)
Everything is an object

I’ve been lying oversimplifying a little in drawing diagrams likes this ...

```
x   332
y
z
```

```
<table>
<thead>
<tr>
<th>class</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rows</td>
<td>47</td>
</tr>
<tr>
<td>cols</td>
<td>52</td>
</tr>
</tbody>
</table>
```

```
GameBoard
__init__
...
```

"foo"
Everything is an object

I’ve been lying oversimplifying a little in drawing diagrams ... because reality gets a little messy.

(I’m still omitting some details, and getting others wrong.)
Data structures in CPython

For every built-in object type in Python ...

*integer, string, dict, set, etc*

... there is a CPython data structure implemented in C

(and compiled to machine language)
**Integer**

Even integers are objects, with methods (!)

\[ x + y \text{ is actually a method call to } x.__add__(y) \]

(Lots of cute tricks to make this reasonably fast)

\[
x = 5
\]
Strings in CPython

x = "foo"

String value refers to an array of characters, ending with a nul (zero byte), as in the C language.
Lists in CPython

\[ x = [3, 2, 4] \]

References an array longer than the current list length, so that \( x.append(7) \) will be fast. Re-allocates an array when necessary.
Alternative List Data Structures

Fast insert, delete from either end; slow to find lis[99] (like List in Java)

Fast insert, delete only at end; fast to find or change lis[99] (like Vector in Java)
Dictionaries are “hash tables”

hash('horse') == 12
hash('goat') == 29

Pseudo-random but deterministic “scatter storage” based on a “hash function”
Ways to implement dictionaries

Search tree: Like binary search, but “go left” or “go right” depending on comparison. Database files use a version of this. Complication: keeping the tree “balanced.”

Hash table: Fast on average, but potentially slow in the worst case because of “collisions” (equal hashes). Compilers use this for variable names. Complications: Handling collisions, expanding full tables.
Python sets are also hash tables

\{ 'horse', 'goat', 'zebra' \}

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>'horse'</td>
<td></td>
<td></td>
<td>'zebra'</td>
<td></td>
<td></td>
<td>'goat'</td>
<td></td>
</tr>
</tbody>
</table>

hash('horse') == 22 % 10 == 2
hash('goat') == 98 % 10 == 8
hash('zebra') == 34 % 10 == 4

Hash codes are actually large numbers; position is remainder when divided by table size. The hash table is expanded (copied to a larger table) if it becomes 2/3 full.

Java library equivalents: hashmap, hashset
The Python Compiler/Interpreter

```
x = 32
if x > 15 :
y = 32
```

- **Lexical analysis**
- **Parser: Syntax Analysis**
- **Bytecode generator**
- **Interpreter**

```
x = 32 ;
if x > 15 :
    y = 32 ;
```

```
stmts
  =
  if
  x 32 > =
  y 32
```

```
2 0 LOAD_CONST 1 (32)
3  STORE_FAST 0 (x)
3  6 LOAD_FAST 0 (x)
9  LOAD_CONST 2 (15)
12 COMPARE_OP 4 (>)
15 POP_JUMP_IF_FALSE 27
4 18 LOAD_CONST 1 (32)
21  STORE_FAST 1 (y)
```
The Python Compiler/Interpreter (simplified)

Conventional compiler: Similar to Java, C, C#, etc.

Code for Python virtual stack machine

Python virtual stack machine

Code:
```python
x = 32
if x > 15 :
y = 32
```

Tokens:
```
x = 32 ;
if x > 15 :
    y = 32 ;
```

Abstract syntax tree:
```
stmts
  - if
    - x
    - 32
    - >
    - =
        - x
        - 15
        - y
        - 32
```

Lexical analysis
Parser: Syntax Analysis
Bytecode generator
Interpreter
Python vs. C, Java, etc.

CPython compiler generates byte code

- vs: C compiler generates machine code (the interpreter is the computer)
- vs: standard Java compiler generates byte code (also for a stack machine)
- vs: Dalek Java compiler (Android) generates byte code for a virtual register machine

Python values are all objects in the heap

- vs: C values can be in the stack or the heap, untagged
- vs: Java “primitive” values are in the stack and untagged; objects are in the heap and tagged

Compiled? Interpreted? Both!