CSc 330 Test #1

Group members: Lucas Main, Keiran Reilly, Philip Snead, Jordan Casoli, Warren Spencer, Kevin Dahl, Robert Martin, Samuel Willis

**A History of Python:**

Python was created in the Netherlands at Stichting Mathematisch Centrum (CWI) in 1989 by Guido Van Rossum, who remains Python’s principal author to date. In 1995 Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI) in Virginia. He released several versions of Python whilstat CNRI until in May of 2000, when he, along with a core development team, moved to BeOpen.com. After a brief stint there, the Python Software Foundation was created.

Guido decided to create Python whileworking at CWI. His immediate inspiration for Python came from the programming language ABC and his motivation came from working with the Amoeba distributed operating system. Whilstworking there he realized the need for a higher level programming language as writing the system administration utilities was taking too long in C and were also not feasible in the Bourne shell. When creating Python he took parts of ABC which he felt worked well, and then added features which he believed were missing. He also acknowledged that to write an Amoeba specific language would be foolish and decided to create a language that was generally extensible.

Much of Python’s syntax is based on ABC, specifically the use of indentation in order to denote blocks of code as well as the use of a colon to separate the lead in clause from the block. This colon in Python was a result of the meaning of indentation being unclear to beginner Python programmers. Furthermore, Python does not use braces to indicate scope. This differs greatly from other popular languages such as C, Java, and Perl, which strictly enforce the use of braces.

Although Python is dynamically typed, it is not radically different from the static typing system that ABC uses. This is due to the fact that unlike other statically typed languages, ABC did not rely solely on static type checking to prevent the program from crashing. Alongside the compile time static type checking, ABC also had a run time library, which would check the argument types for all operations each time they were executed. This was implemented in ABC because it was intended to be an interactive language. Python simply dropped the compile time checking that ABC utilized and moved to completely dynamic typing. This was not without its repercussions. In ABC the types of arguments could be deduced from the form of the operation, however in Python this is not always possible. For example, x + y could be a string concatenation or a numeric addition.

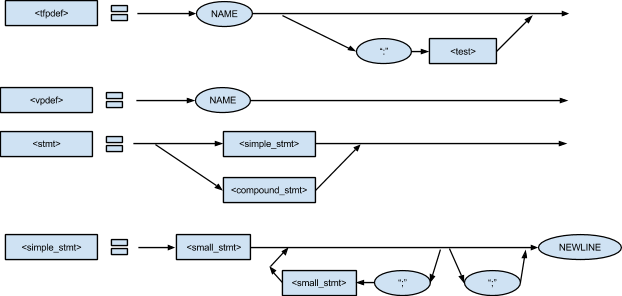
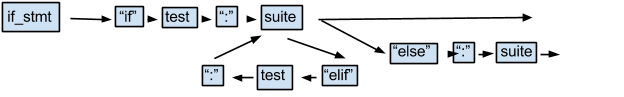
In Python everything is an Object, including classes and primitive types, and extensive introspection of types and classes is supported. This allows operations such as comparisons and the ability to extract the type attributes as a dictionary. The major data types in Python were also based upon those of ABC, with some implementation changes. For example, in ABC the list data type was really a multiset implemented using a modified B-tree implementation, and its tables were associative arrays sorted by a key. Guido foresaw this being a problem for many common use cases, such as representing the sequence of lines read from a file. To rectify this potential setback he decided to implement the list data type as a flexible array with insert and deletion operations. This gave users complete control over the ordering, but a sort method was also given for when sorted results were needed. Guido kept the immutable tuple data type from ABC, and added array-like slicing and indexing.

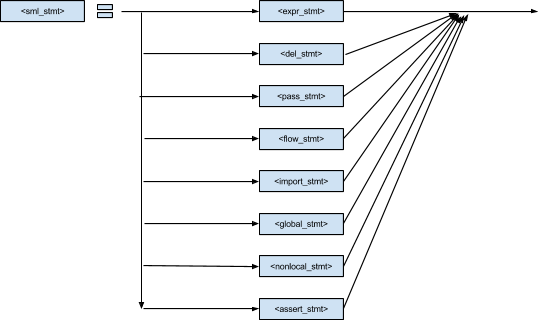
Guido also chose to implement numbers as machine integers and binary floating points. These are both simply represented by two C data types, double and long. He then tried to add an unbounded exact set of numbers called long. However in doing so he made several self-admitted design mistakes. Since there are two different Integer types, a way for distinguishing which type to use was needed. The solution Guido arrived at was to have the user explicitly state with a trailing L that they wanted to use long. This directly violated the ABC-inspired philosophy of not concerning the user with implementation details. This was only a minor set back in the implementation of numbers. A second issue with this implementation of Integers was that the semantics for ints and longs were slightly different in certain cases. These semantic errors were in part due to clipping problems and integer overflows, causing certain operations to produce different results depending on whether int or long was used. To resolve this, an OverflowError exception was added, and thrown when an overflow would occur. This, however didn’t fully solve the problem, and today, when Python encounters an overflow, it promotes the int to a long. This OverflowError functionality wasn’t made entirely obsolete, though, as it lead to the decision to have exceptions raised when no correct return value can be computed. This means that no Python program can ever fail as a result of undefined values being passed around in the Python VM.

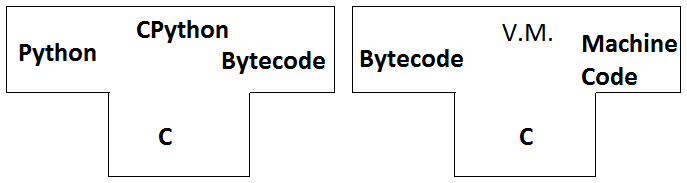
Python’s semantics are similar to those of most lambda-based functional languages, although no formal, systematic description of Python’s semantics has been made. Python’s semantics are used by the interpreter at compilation time while parsing the code. Python’s semantics for strings are heavily based on ABC. Because Python’s semantics were based on lambda calculus, problems appeared that were already apparent in other lambda-based languages. One example involves scope and how variables are passed to functions. The fix for this was not added until version 2.2 and involved the implicit carrying of references to local variables in surrounding scope.

**The Most Recent BNF Grammar as Rail Diagrams:**

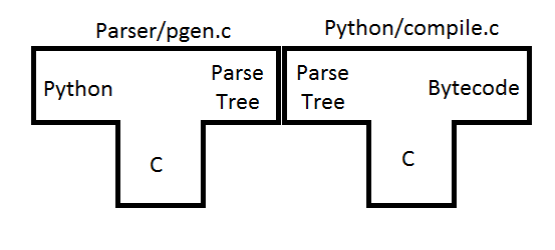
A small sample of the rail diagrams. Please see submitted pdf on the website.

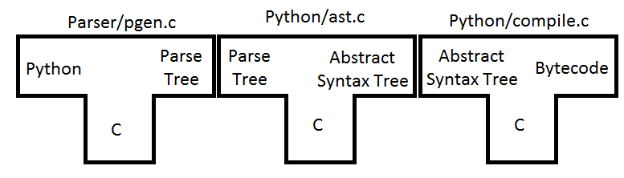


**T-Diagrams:**

Python typically compiles with CPython, which is a bytecode interpreter written in C. CPython parses the Python code, and writes it to Python Bytecode, which runs on the Python Virtual Machine. The virtual machine is written in C, and will output low level machine specific code. The following diagram shows the process. Note that we assume the system is already able to execute C code. 

Before version 2.4, the CPython compiler worked as follows:

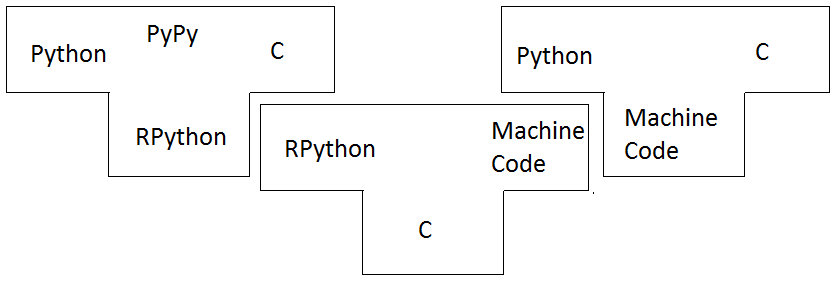


After version 2.5, some extra steps were added to make the compiler more in line with standard compilers. The new version of CPython is built as follows:

Note that Python/compile.c actually executes two steps. It turns the AST into a control flow graph, then translates that into Python bytecode.

Later, other compilers were made, following the same structure as CPython. These included Jython (java), CLPython (creative lisp), and IronPython (C#). Each of these compilers is a bytecode interpreter, following the same structure as CPython, shown above.

More recently, with Python 2.7, a Python compiler has been written in a restricted subset of Python. This compiler is called PyPy, and differs from the other compilers in that it is not a bytecode interpreter. It takes the Python code, and outputs C code, which is then compiled and run.



Basic excerpt from pdf below:

*When you invoke the “\python" command, your raw source code is scanned for tokens, these tokens are parsed into a tree representing the logical structure of the program, which is nally transformed into bytecode. Finally, this bytecode is executed by the virtual machine.*

**Type Inferencing in Python:**

Type inference allows the compiler for a program to make assumptions about and optimizations for the produced bytecode without requiring the programmer to specify types for any methods or variables. Implementing type inference would allow the Python compiler to perform bytecode level optimizations to improve performance of Python programs.

Type inference requires a statically typed language. This makes it difficult to implement with Python, as Python is dynamically typed. Specifically, Python uses a scheme called “Duck Typing”. When attempting to invoke a method or access an attribute of an object, Python does not check to see if the object is of the correct type, instead only checking that the object has the appropriate method or attribute. Thus, types in Python have little meaning. However, an offshoot of the PyPy project, entitled “RPython” is statically typed. This could be modified to support type inference.

Additionally, Python contains several constructs which greatly complicate type inference. Lists are a good example, being capable of holding any type of object at any index. Also of concern is the ability for classes to change their superclasses.

Python is also capable of creating and executing additional code during execution. To use static types with such code would require either that it be created with static typing in mind, or that the type inference routine be run again when this code is executed. The former solution places a great deal of responsibility on the programmer -something Python is meant to avoid- while the latter could introduce delay during program execution.

Two distinct algorithms are generally used for implementing type inference. The first, known as the “Hindley-Milner” algorithm, gives all variables and methods a polymorphic, “T” type. This T type can take the form of any other monomorphic type (for example, int or char). This algorithm has the drawback that it can only infer a polymorphic or monomorphic type, not a set of potential types. This can result in variables having either too specific or too general of a type. Finally, this algorithm will raise an error when two types could either be logically inferred. The second, is known as the “Cartesian Product Algorithm”. This algorithm is used to perform type inference on method calls, but can be joined with another algorithm, named “Iterative Type Analysis”, which infers the types of method bodies. The algorithm creates a constraint graph, which models the programs execution and can be used to infer the types of variables at each step of execution.

Implementing Type inference requires knowledge of control flow of the program. This is so the compiler knows what interactions the variables and methods have with each other. Control flow information is something which interpretive languages like Python generally lack. For example, the Python compiler makes no guarantee that imported code at compile time is the same code which will be executed. This makes inferences on interactions with such code dangerous.

Types of variables and methods in the local namespace can be much more accurately inferred than those in external namespaces because names in the local namespace remain static between compile time and execution time. Of this, only atomic types, (Integer, Longs, ASCII and Unicode strings, Floats, and Complex Numbers) can be inferred. Classes and atomic container types, such as dictionaries, lists, and tuples cannot be inferred, though since tuples are immutable their contents can have their types inferred.

There have been some previous attempts to create type inference in Python. Psyco is a just-in-time compiler that detects integers and strings which are static between compilation and execution. However, it executes outside the compiler, and doesn’t infer any other types.

Starkiller was another attempt to create type inference in Python. Starkiller used the Cartesian Product Algorithm to infer types within Python’s source code. Starkiller produced a complete type inference of Python source code, however did so while ignoring language semantics. It also assumed that the code that the program was compiled with was the same code that was used at execution time.

Brett Cannon created an algorithm for implementing type inference in Python. His algorithm was remarkably similar to the “Iterative Type Analysis” algorithm, though his took Python’s control constructs into account whilst inferring types. For conditionals, his algorithm gave variables a set of potential types, containing the types the variables had before reaching the conditional, unioned with any types the variable could be given within any branch of the conditional. For looping, his algorithm would parse through the loop in a flow-insensitive manner, not taking breaks or continues into account. This allowed the algorithm to union the set of potential types for a variable with a set containing any type it could have from any point within the loop. Finally, for exception handling blocks, he assumes any line could throw an error. Because of this, the set of potential types for each variable in the exception handling block contains the types they had before the block and any type they could be assigned within the block.

However, Brett’s type inference algorithm resulted in only small improvements, averaging roughly 1%, over the standard Python compiler. He concluded that his modifications to the compiler given their small performance gains and the amount of code maintenance which would be required were “in no way justified”.

In 1999, University of Victoria researcher John Aycock attempted to introduce static typing to Python bytecode, using a method he dubbed “Aggressive Type Inference”, or ATI. This method operates under the assumption that, despite Python being dynamically typed, programmers generally don’t write dynamic code. This assumption was supported by a study which found that, in a large sample of Python programs, 80% of variables remained the same type within their scope.

The ATI scheme is comprised of two phases. The first goes over the original Python and creates a file detailing several kinds of information: the scope of variables and methods; variable assignments; operations on variables; return types of methods; names of types; equivalences between names; import statements; and global declarations. The second phase then attempts to infer types through successive passes over the file.

Aycock acknowledges several shortcomings in the procedure. His implementation does not fully type lists, and can fail to find all possible types of a variable under several conditions, most notably when methods are called unpredictably, such as from a list of methods.

**References:**

**T-diagram:**

<http://docs.python.org/devguide/compiler.html>

<http://www.troeger.eu/files/teaching/pythonvm08.pdf> page 12

<http://tech.blog.aknin.name/2010/04/02/pythons-innards-introduction/>

<http://tomlee.co/wp-content/uploads/2012/11/108_python-language-internals.pdf>

http://docs.python.org/2/reference/expressions.html

http://docs.python.org/2/reference/grammar.html

**History:**

<http://www.troeger.eu/files/teaching/pythonvm08.pdf>

<http://python-history.blogspot.ca/2009/01/personal-history-part-1-cwi.html>

<http://www.tutorialspoint.com/python/python_basic_syntax.htm>

http://docs.python.org/2/license.html#history-of-the-software

<http://effbot.org/pyfaq/why-was-python-created-in-the-first-place.htm>

http://gideon.smdng.nl/wp-content/uploads/thesis.pdf

**Types:**

<http://lambda-the-ultimate.org/node/1519>

<https://code.google.com/p/rpython/>

<http://www.python.org/workshops/2000-01/proceedings/papers/aycock/aycock.html>

<http://citeseer.ist.psu.edu/viewdoc/download;jsessionid=5CD79215ADB1E853C2651EC965225F60?doi=10.1.1.90.3231&rep=rep1&type=pdf>

<http://lambda-the-ultimate.org/node/1519>

<https://code.google.com/p/rpython/>

**Power Point Link:**

<https://docs.google.com/presentation/d/1WLUy52pTVe-vnCOzRYoU8b0vRWN-VViEv6yhFgcuTog/edit#slide=id.g10a747292_022>