Python: Grammar, Compilation, History, and Types

**1) History of Python**

The Python programing language is used by many programmers and for a large variety of applications. The following sections outline its evolution in terms of syntax, semantics, pragmatics and implementation techniques in comparison with previous programming languages.

**1.1 Founders and Evolution**

Python was first released in 1991 by Guido Van Rossum at Stichting Mathematisch Centrum (CWI) in the Netherlands, the same place that invented Algol 68 (under the name the Mathematical Centre)[2]. Van Rossum originally worked at CWI on the ABC group then moved to work on the Amoeba project, which was developing a micro kernel-based distribution system. His motivation for developing Python was as a way to “bridge the gap between C and the shell,” as C was taking too long to implement the project.

Syntactically, Van Rossum’s “tastes...were strongly influenced by languages like Algol 60, Pascal, Algol 68, and ABC.” As a successor to ABC, Python tried to correct some of ABC’s flaws, as Van Rossum saw them. It is built on different concepts from all of these languages, including Modula-3.

Python was originally implemented in C taking advantage of “the extensive, well understood, portable C libraries”. Because of this, it is portable to Unix, Linux and POSIX environments[15].

Python got its name from a BBC comedy series from the seventies "Monty Python's Flying Circus.” The designer needed a name that was short, unique, and slightly mysterious[13].

**1.2 Syntax**

Python’s first and foremost syntactical influence during its development was the ABC language [2]. Van Rossum had been working with ABC for 4 years and had clear ideas of how Python could become a much more extensible ABC. Since ABC was already a well-known concept of programming style it was unnecessary to deviate far from the syntax of ABC. For example, the indentation used in Python’s syntax is ported directly from ABC, and ABC got it from Donald Knuth. The indentation was promoted by Donald Knuth from a desire to simply have the text look neat and be suggestive of the meaning. This was previously done but never enforced. Like ABC, these indentations are used for statement grouping. An example of indentation use in python is:

\*\*\*

for i in range(0,10):

if(i > 5):

print ‘hello’

\*\*\*

Another ABC syntax feature that is used by Python and separates it syntactically from other languages like Java or C is the non-use of semicolons for ending statements. Instead, end of statements are simply indicated via a new line or the ‘/n’ symbol. This is also illustrated in the example above.

One last interesting Python syntax that was directly ported over from ABC is the colon. Also shown in the example, the colon was developed in ABC to reinforce the function of the indentation. The colon clears the ambiguous nature of which lines a conditional statement is applied to. With only indentation it was unclear to beginner programmers which statements were tied together.

**1.3 Semantics**

Semantically, Python is mainly derived from ABC, with close ties to both LISP and SCHEME. Being object-oriented, an object has an identifier and a set of attributes. The identifier can be an integer, as with most objects, or a string, in the case of built-in functions. Python’s strings, although they have different notation and 0-based indexing, have similar immutable semantics to ABC strings. Lists, tuples and strings were all incorporated into the general sequence type, which many core operations could then be applied to. Exceptions are always raised when no correct return value can be computed, therefore program failure due to unknown undefined values is not an issue. In Python, numeric variables are distinguished by the user attaching a capital ‘L’ to the end of the number, signifying that a long is being used. For example, 32\*\*32%10 = 6L, which indicates that the answer is a long. [20]

According to Dwight Guth, in his thesis *A Formal Semantics of Python 3.3 [11],*

*“Of the rules, roughly 40% are associated with assigning semantics to components of Python syntax, roughly 40% are associated with assigning semantics to built-in functions, and the remaining 20% are associated with the organizational structure of the semantics. Semantics define 45 modules, 1406 sentences (mainly rules),608 productions, and 32 cells in its configuration, defined over 4611 lines of code.”*

**1.4 Pragmatics**

In terms of pragmatics, Python’s ease of implementation stands out in comparison to other programming languages. A Python program is typically 3-5 times shorter than a Java program or similarly 5-10 times shorter than a C++ program. This is due to the use of dynamic typing and high level data types such as polymorphic lists and dictionaries in Python. These features allow the user to not be slowed down by declaring variables. However, a Python program generally runs slower than a Java program for the same reasons, since dynamic typing requires more work during compile time [1].

Dynamic typing can also create problems for inexperienced programmers, since incompatible types are only found at run time.

A common way to make use of Python’s strengths while limiting its weaknesses is to implement performance-critical parts in Java or C++. The object oriented nature of Python makes the conversion between these languages simple.

Many programmers prefer a certain type of language due to its ideology. Python abandons individual ideological standpoints by combining many language types such as object-oriented (Java), functional (Lisp), procedural (Pascal), Efficient (C), and Expressive (perl).

Python also has powerful standard libraries, a large support community and a free, compact implementation. In comparison, Scheme has multiple incompatible versions [1] [8].

**1.5 Implementation Techniques**

There are now many different compatible implementations of Python; it also has its own Standard Library. The major aspects of Python that make it so appealing as a language are its emphasis on clean syntax, the combination of objected-oriented with functional programming techniques as well as its use of dynamic typing. While variables in C need to be explicitly declared, variables in Python are simply names referring to objects. It uses an interpreter rather than an extra compilation step for type-checking.

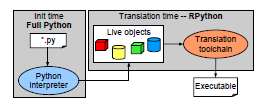
Other implementations of Python include: CPython (the original, most maintained implementation written in C), Jython (written in Java and runs in JVM), Python for .NET written in C and IronPython in Python (.NET applications) and PyPy (an implementation written completely in Python)[1].

**2) Type Inference**

The Python language was designed to be a very dynamically typed language and as such, contains many aspects that make type inference difficult. Some of the aspects of Python that make implementing type inference so difficult stem from one of Python’s most powerful features, dynamic execution[29]. Python provides developers with the ability to dynamically create and execute code while the program is running. This dynamic execution behaviour includes the loading of external modules at runtime, the *eval* and *exec* functions, which allow for dynamic execution of Python code[25], and the ability to create new methods and even change the class type of an instance variable. All these features of Python make type inference impossible or extremely impractical, since the control-flow and code being used to infer the types can actually change during runtime.

Even with the design and purpose of Python making the language inherently resistant to type inference, there have been some attempts to provide type inference through using a subset of Python. Some of these attempts have been successful, while others have long since died out. A few of the more successful attempts are listed below.

One attempt to bring type inference to Python is RPython, a restricted subset of Python. RPython makes type inference possible by initializing a full Python program, analyzing the program as a whole and inferring types, and then producing an executable[26]. RPython is used in implementing PyPy, a Python interpreter and just-in-time compiler. PyPy was created to be a faster alternative implementation of Python, and with its just-in-time compiler is able to translate Python programs to machine code at runtime[27]. The PyPy project has also created a toolchain that is able to translate RPython to a lower level language such as C (the PyPy interpreter is an example of this) Java bytecode, or Common Intermediate Language (CIL). The ability to translate into these lower level languages brings speed and compatibility that would otherwise not be possible without the static type checking provided by RPython’s type inference. The PyPy project, using RPython, is still actively being developed today.



RPython translation from python source to executable [32]

Starkiller, done by Michael Salib as his Masters project at MIT converts Python code to C++ code but does not function on certain aspects of Python, such as the ability to dynamically load in new code through modules, and eval or exec statements[30]. StarKiller’s algorithm is based on the Cartesian Product Algorithm (CPA), which infers the types of arguments passed to polymorphic functions, then creates and caches a statically typed version of the function[31].

A third and different approach is called Aggressive Type Inference (ATI) and is outlined in a paper by John Aycock from the University of Victoria[28]. In his paper, Aycock states the core idea of ATI as “*Giving people a dynamically-typed language does not mean that they write dynamically-typed programs.*”. The basic idea of ATI is that even though Python gives programmers the ability to write very dynamically typed code and provides the ability to load and execute new code at runtime, this does not necessarily mean that the majority of Python programs take advantage of these features. Aycock uses this idea to propose a type inference system that works based on the assumption of “Type consistency”. This type inference system, ATI, assumes that once a variable in a Python program can be determined to have a type T, then throughout the lifetime of that variable in its scope, the variable will always have, or be assigned, the type T. Using this kind of assumption makes type inference with Python much simpler (and even possible) for the programs that fall into this subset, and simply throw a compile-time error for programs that do not. While this attempt still follows the idea of only performing type inference on a subset of Python, that subset is not strictly enforced, i.e., the language itself is not changed; it instead relies on what Aycock believes is a more common programming style, which allows for type inference.

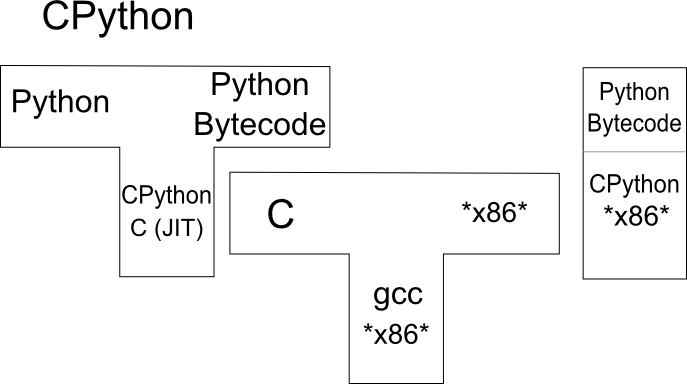
In the same vein as the previous example, a fourth method of type inference for Python could be to run the program with test cases and record the types of variables. Then, if the test cases were representative of real world cases, the types of variables for real world cases could be known and many of the advantages of type checking achieved.

There are a number of benefits that come from introducing type inference to any dynamically typed language. One of these benefits is the ability to further optimize the code being executed. These optimizations can range from the simple execution speed gained from not requiring more expensive runtime type checking, to being able to perform more complex optimizations based on the knowledge of type information throughout the program. Another benefit of introducing type inference to Python, previously encountered with the discussion on RPython, is the ability to compile down to a lower level language such as C, Java bytecode or Common Intermediate Language, a task not possible without the type information obtained from type inference. Additionally, type-related bugs can be found at compile-time rather than inconveniently during execution, and advanced IDEs can provide more support features such as code completion and refactoring. These benefits all contribute to the attempts, past, present and future, which endeavor to bring type inference to the Python language.

**3) T-Diagrams**

Historically, Python has been implemented using a compiler which produces Python bytecode. This bytecode is then interpreted. Python compiles code implicitly as modules are loaded, requiring that the compiler be available at runtime.

CPython is the standard Python implementation. The CPython interpreter and compiler are both written in C. Nonstandard Python implementations also exist, such as Jython, IronPython and PyPy, which are written in Java, C# and RPython respectively.



CPython T-Diagram

In versions prior to Python 2.4, Python compilation was broken into two steps. First, Python source code is translated into a parse tree. Second, the parse tree is used to emit Python bytecode to be executed by the Python interpreter. In version 2.5, the compilation process was further simplified by breaking it down further as follows:

1. The source code is converted to a parse tree (Parser/pgen.c)
2. The parse tree is converted into an abstract syntax tree (Python/ast.c)
3. The abstract syntax tree is converted to a control flow graph (Python/compile.c)
4. The control flow graph is used to emit the final executable bytecode. (Python/compile.c)

To understand the compilation of a Python program, we must understand the Python compiler. We will outline the components of the Python compiler and the functions of those components.

**3.1 The CPython Parser**

The Python parser is a standard *LL(1) Parser.* An LL(1) Parser is any parser that analyzes any grammars which are (1) context free, and (2) cannot generate a parsing table with multiple entries. An *LL(1) language* is any non-left-recursive, non-ambiguous language.

All LL(1) parsers

* Analyse input from left to right
* Construct the leftmost derivation of each sentence
* Use exactly 1 token of lookahead while parsing a sentence

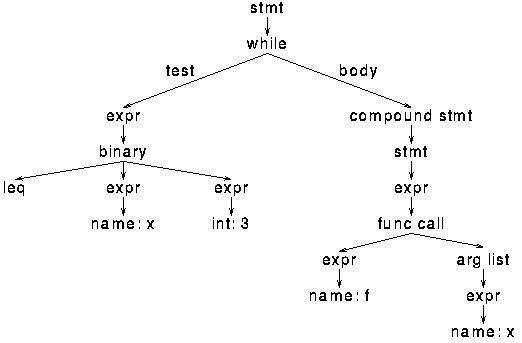
A numerical representation of the Python grammar rules is found in *Include/graminit.h*. The Python parser compares the input source code to grammar and then returns a *parse tree* of the program. A parse tree is any ordered, rooted tree, which represents the syntax of some formal grammar. The nodes of this tree consist of *node \*structs* as defined in Include/node.h. Data may be queried from these nodes via the use of several macros included in Include/node.h.

Supported queries include returning

* the 0-indexed *n*th child of a node
* the 0-indexed *n*th child of a node from the right side of the tree
* the number of children a node has
* the type of a node
* the line-number that the parse rule of the node originated from

After the parser generates the tree which corresponds to the source code, it is converted into an *abstract syntax tree.*

**3.2 CPython Abstract Syntax Tree Implementation**



An Abstract Syntax Tree [5]

An *abstract syntax tree (AST)* is an ordered, rooted tree, which represents the structure of some source code, but does not contain the code itself. Before discussing how the Python compiler converts parse trees to abstract syntax trees, we will consider how Python handles abstract syntax trees themselves.

The nodes in a Python AST represent source code components such as statements, expressions, and lists. Most nodes correspond to exactly one source code structure. Nodes are implemented in ASDL, a language designed to provide developers with a way to reuse tree-like compiler structures across various languages. These ASDL node structures are compiled to C using SPARK. The tree is now ready to be converted into a *Control Flow Graph.*

**3.3 Memory Management in the CPython Compiler**

The issue of memory management is handled using a large pool of shared memory called an *arena.* The arena allows the programmer to avoid explicit deallocation of many dynamically allocated objects, such as the nodes in the parse tree, AST, and *control flow graph.* When the compiler exits the entire arena is deallocated at once.

This is achieved using the *PyArena* structure. This structure returns pointers to each memory address it is supplied and keeps track of which blocks of memory need to be freed. *PyObjects* still must be explicitly deallocated, and the arena must be informed of their presence.

**3.4 Parse Tree to AST Conversion**

An AST is generated from an existing parse tree by walking through the parse tree.

As the tree is traversed

* Space is allocated for new AST nodes as necessary
* Whenever a node of type *t* is to be allocated, the type *t* AST node creation routine is called
* AST nodes are connected to other AST nodes as necessary

The grammar specification is not connected to the parse tree in any way, so it is the programmer’s responsibility to ensure that the grammar is upheld. For example, given an ‘*if’* statement, the programmer must check explicitly for a ‘*:*’ directly after.

Processing the nodes in the parse tree results in a sequence of *asdl\_seq structs*, which contain AST nodes.

**3.5 Control Flow Graphs**

A *Control Flow Graph (CFG)* is a digraph that models program flow. Nodes in the CFG are basic blocks of code written in the *intermediate representation* or *IR* of the source code. Python bytecode is the intermediate representation contained in these blocks.

Every basic block has a single in-edge, or *entry point* and zero or more out-edges or *exit points.* Entry points represent “targets” for operations which can change control flow. Such operations include function calls and jumps. Exit-points are, conversely, the instructions that can cause changes in control flow.

**3.6 AST to CFG Conversion**

Initially the AST must be converted entirely into Python bytecode. At this point, jump commands do not resolve to specific offsets. The result of this conversion is the desired CFG, where edges represent control flow.

This is a two-pass conversion. The first pass consists of generating a symbol table from the AST. This is achieved by walking through every node in the AST using the appropriate *symtable\_visit\_type* function. Whenever the border of a local variable’s scope is met, the converter calls an appropriate routine to signify the death of the variable. For example, if the end of a block is encountered, it would be appropriate to call *symtable\_exit\_block().* Variables in the symtable can be classified as *local*, *free*, or *global*.

CFG creation commences after the symbol table is generated. Each AST node type has an associated conversion handler of the form *compiler\_visit\_type.* These handlers are typically large switch statements which allow the description of the situation to become more and more precise.

After the AST is successfully converted to a CFG, all that is left is to emit the correct bytecode.

**3.7 Bytecode Emission**

Bytecode emission is handled by macros which correspond to code constructs such as opcodes with and without arguments, absolute jumps, relative jumps. These macros also use several helper functions to assist in bytecode emission. Basic blocks from the AST must be generated in the CFG. There are several functions provided to accommodate this as well.

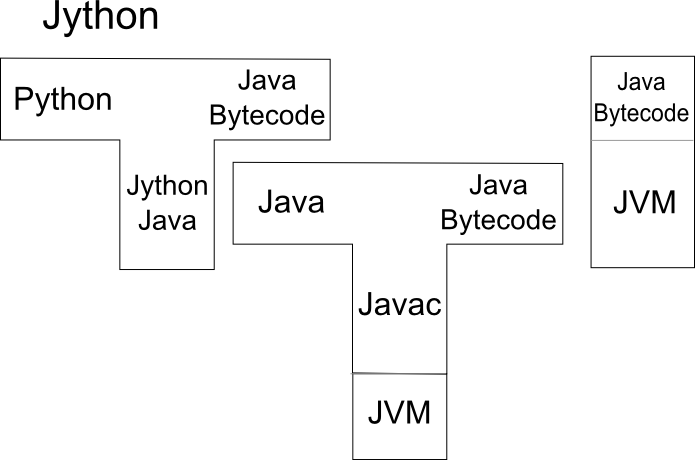
After the CFG is generated from the AST, the CFG is flattened by performing a post-order depth-first search. The jump offsets are computed based off of the flattening. The final product is an executable PyCodeObject file.

**3.8 Other Compilers**

There are several popular, well-maintained alternatives to CPython. For comparison, we will outline two.

PyPy is a Just-In-Time compiler for Python, written in Python. PyPy uses a subset of Python called RPython during the compilation process, and includes a toolchain which can compile RPython into lower level languages. Such languages include C and Java bytecode.

Jython compiles Python code to Java bytecode, which may be run in the JVM. Below is a T-Diagram for Jython for comparison with its CPython counterpart.



Jython T-Diagram

**4) BNF Grammar to Rail Diagram**

See attached file Python\_Rail\_Diagrams.pdf

**References**

[1][http://](http://docs.python.org/3.3/library/functions.html?highlight=exec" \l "exec)[www.python.org/](http://www.python.org/doc/essays/comparisons.html)

[2](2013, Oct 2). “History and License” [Online]. Available: [http://docs.python.org/3.4/](http://docs.python.org/3.4/license.html)

[3][26] (2013, Oct 2). Getting Started with RPython [Online]. Available:<http://doc.pypy.org/en/latest/getting-started-dev.html>

[4][27] (2013, Oct 2). PyPy Website [Online]. Available:<http://pypy.org/>

[5]<http://alumni.media.mit.edu/~tpminka/patterns/python/tree.jpg>

[6]<http://www.artima.com/weblogs/viewpost.jsp?thread=85551>

[7]<http://asdl.sourceforge.net/>

[8]<http://www.cio.com/article/185350/You_Used_Python_to_Write_WHAT_?page=3&taxonomyId=3038>

[9]<http://www.csd.uwo.ca/~moreno//CS447/Lectures/Syntax.html/node14.html>

[10][http://www.cs.ucsb.edu/~urs/oocsb/self/papers/cpa.htm](http://www.cs.ucsb.edu/~urs/oocsb/self/papers/cpa.html)[l](http://mike.salib.com/writings/thesis/final.pdf)

[11]Dwight Guth (2013, Oct 2). “A Formal Semantics of Python 3.3” M.S. Thesis, University of Illinois, Urbana, Illinois, 2013. Available at: <http://fsl.cs.illinois.edu/FSL/papers/2013/guth-2013-thesis/guth-2013-thesis-public.pdf>

[12]<https://gcc-python-plugin.readthedocs.org/en/latest/_images/sample-gimple-ssa-cfg.png>

[13](2013, Oct 2) The Python Programming Language [Online] Available at: [http://groups.engin.umd.umich.edu](http://groups.engin.umd.umich.edu/CIS/course.des/cis400/python/python.html)

[14]<http://homepages.cwi.nl/~steven/abc/language.html>

[15]Steven F. Lott (2013, Oct 2). “Building Skills in Python - Background and History,” [Online]. Available at: <http://www.itmaybeahack.com/book/python-2.6/html/p01/p01c01_background.html>

[16][29]Brett Cannon, "Localized [Type Inference of Atomic Types in Python](http://www.python.org/dev/peps/pep-0339/)," M.S. thesis, CPSU, San Luis Obispo, California, 2005. doi: [10.1.1.90.3231]

[17]<http://mike.salib.com/writings/thesis/final.pdf>

[18]<http://mvdirona.com/jrh/talksandpapers/guidovanrossum_21_years_of_python.pdf>

[19]<http://pages.cpsc.ucalgary.ca/~aycock/spark/>

[20]Guido Van Rossum - Blogspot (2013, Oct 2). The History of Python [Online]. Available at: [http://python-history.blogspot.ca](http://python-history.blogspot.ca/2009/02/early-language-design-and-development.html)

[21]<http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Parse_tree.html>

[22]<http://proquest.safaribooksonline.com/book/programming/python/40000python/1dot-python-introduction/navpoint-4>

[23]<http://www.python.org/dev/peps/pep-0339/>

[24]<http://www.csd.uwo.ca/~moreno//CS447/Lectures/Syntax.html/node14.html>

[25](2013, Oct 2). Python Documentation [Online]. Available:<http://docs.python.org/3.3/library/functions.html?highlight=exec#exec>

[26](2013, Oct 2). Getting Started with RPython [Online]. Available:<http://doc.pypy.org/en/latest/getting-started-dev.html>

[27](2013, Oct 2). PyPy Website [Online]. Available:<http://pypy.org/>

[28]John Aycock (2013, Oct 2). Agressive Type Inference [Online]. Available:

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

[29]Brett Cannon, "Localized Type Inference of Atomic Types in Python," M.S. thesis, CPSU, San Luis Obispo, California, 2005. doi: [10.1.1.90.3231]

[30]Michael Salib, “Starkiller: A Static Type Inferencer and Compiler for Python,” M.S. thesis, MIT, Cambridge, Massachusetts, 2004

[31]Ole Agesen, “The Cartesian Product Algorithm,” *ECOOP'95 Conference Proceedings, Århus, Denmark, August 1995.*

[32]D. Ancona, “Rpython: Reconciling dynamically and statically typed oo languages.” In DLS, 2007.

[33]Alfred Aho, Ravi Sethi, Jeffery Ullman, *Compilers: Principles, Techniques, and Tools,* Addison Wesley,1986