Grammar

See Appendix A for the rail diagram translated from the BNF grammar for Python 3.4.0a3. The grammar was translated one line at a time; most was fairly straight forward although some constructs required an extensive amount of space. An example of this is varargslist which had many paths.[8]

T-Diagrams

Python is a very high-level scripting language. It is implemented in a way that focuses on interactivity. Python contributes some features with scripting languages, but also it contributes some features with more traditional programming languages.[4]

Python is considered to be a dynamic language that cannot be compiled into machine code statically same as C. It needs an interpreter to execute the code which according to the definition of the language, is a dynamic operation. You can translate source code into bytecode, which is an intermediate process that the interpreter does to speed up the load of the code. It converts text files with comments, blank spaces, words like 'if', 'def', 'in', etc into binary code. The operations behind are exactly the same, in Python, not in machine code or any other language. This is what it's stored in .pyc files and it's also portable between architectures.[4,7]

Throughout the versions of python there haven't been many major changes to the compiler and interpreter. Prior to version 2.5, there were two major steps in compiling the source code to bytecode. The first step was to parse the source code into a parse tree using code written in c using Parser/pgen.c. The second step was to generate bytecode from the parsetree using Python/compile.c. However in versions 2.5 and onward the compilation step is broken down into three separate steps to be more fitting with a standard compiler. The revised steps are now as follows:

> "1. Parse source code into a parse tree using Parser/pgen.c.
> 2. Transform parse tree into an Abstract Syntax Tree using Python/ast.c
> 3. Transform AST into a Control Flow Graph using Python/compile.c.
> 4. Emit bytecode based on the Control Flow Graph using Python/compile.c." [5]

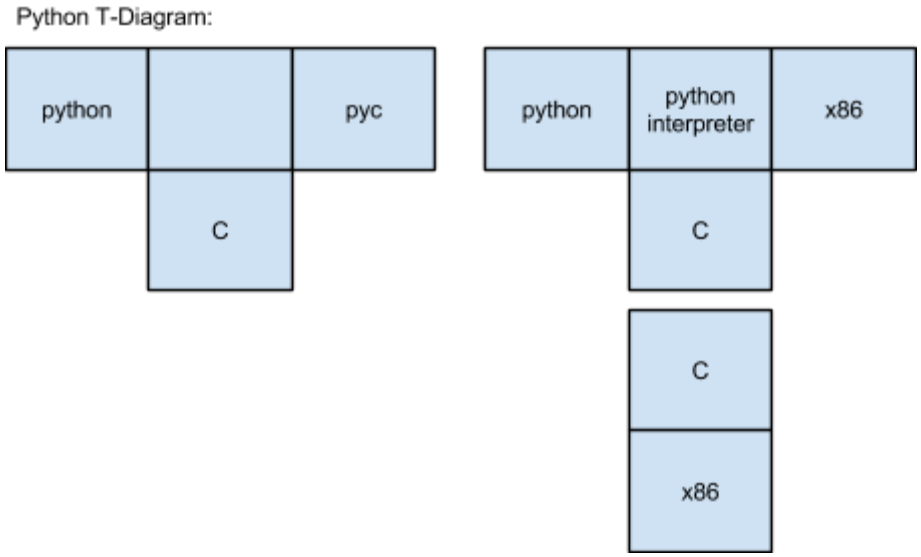

Figure 1: Python T-Diagram

While there haven't been many major changes to the interpreter, other developers have implemented their own. Some of the more popular implementations include: PyPy, Jython, IronPython, and Stackless. PyPy is an interpreter for python versions 2.7.3 and 3.2.3. It is designed to be fast and uses a Just-In-Time compiler. Jython is an interpreter designed to integrate with Java Virtual Machine, giving access to the Java library and tools.. IronPython is an interpreter which integrates well with the Common Language Runtime provided by .Net and Mono. Stackless Python is named like that because it averts relying on the C that is called stack for its own stack.[13,12]

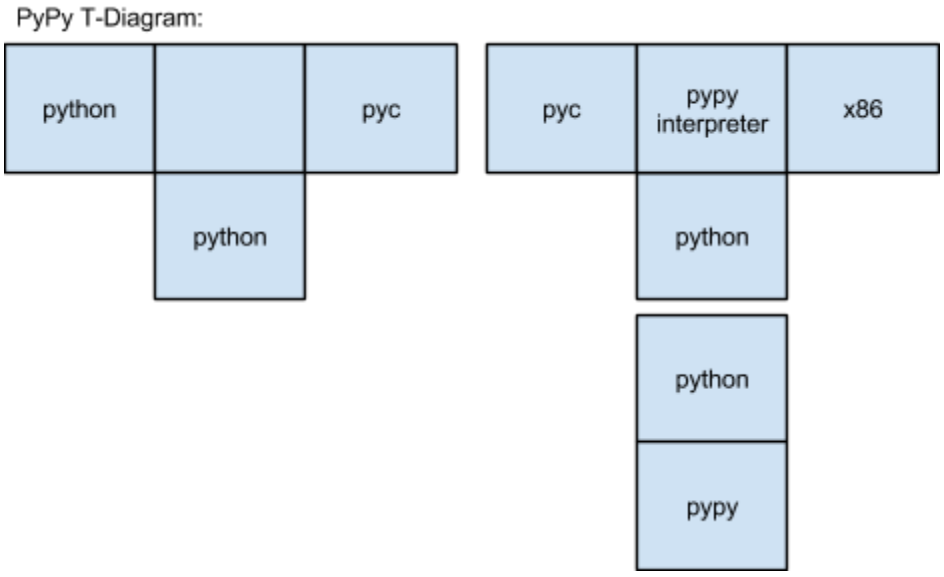

Figure 2: PyPy T-Diagram

[10]

History

Python was designed as a general-purpose language in the late 1980's. It's creator Guido Van Rossum had worked on the ABC programming language, which was developed at CWI as a teaching language that laid huge emphasis on clarity. Python was heavily influenced by ABC, borrowing most of the concepts used such as indentation, data types etc. Rossum wanted Python to be high-level enough to be easy to read and write, while also offering easy portability and direct access to low-level capabilities.

Python was designed to be highly extensible rather than allowing its desired functionality to be built into the language's core. Its major goal was to create a programming language where things would be as simple and readable as possible using less-complex grammar and syntax than those available in other languages. Another core philosophy in their approach to language design was that "There should be one-- and preferably only one --obvious way to do it" which went directly against Perl's approach. That ensured that Python was implemented to be as clear as possible. Additionally, Python is an object oriented language at its core preferring to make everything objects (including classes). The fact that everything in python is an object, was inspired by Smalltalk-80.

In recent history, Two versions of Python have been developed in parallel. Python 3 was intended as an overall cleanup to the Python language, that broke backwards compatibility with python 2. Python 3 was also the first version of Python to have a formally defined BNF grammar. Due to the work involved with porting a project from Python 2 to Python 3, as well as Python 3 not being as complete, Python 2 was still actively worked on. Many of the features and security updates from Python 3 were eventually ported to Python 2.7. Python 2.7 is going to be the last version of the Python 2.x family, and any more updates will only be for security purposes.[17]

Typing Mechanism:

Python is dynamically typed i.e all type checking is done at runtime. In Python, the concept of types resides with objects and not names. Python values are very generic in nature and refer to a particular object at a particular point in time. Objects contain a header field that tags the object with its type and its reference. That way, Python handles parameter passing by referencing the object in contrast to C where everything is passed by value. The object reference is passed to a function parameter as needed. Object references to integers, tuples or strings cannot be changed in the function nor anywhere else as every object reference is immutable. Python also allows for automatic garbage collection when an object is not referenced by another name or object. Python's basic types and collections are inspired by Pascal. Like Pascal It includes sets, lists, and maps(dictionaries) as built in collections. Python also includes tuples, which appear to be used first in LISP. [9]

Dynamic typing serves as the crux for Python's implementation of polymorphism. Polymorphism allows for flexibility, because the user does not need to define as many classes as potential data type it requires, as in statically typed languages. This way, Python is able to design more simple classes frameworks. Duck typing also allows for simpler classes because subtyping is unnecessary to guarantee that a class has a certain method. Duck typing was first implemented in Ada in 1980.[11]

Syntax + Semantics:

The base syntax of Python is inspired by the ABC programming language. The biggest things it takes from ABC, are the use of whitespace to denote a blocks scope and the use of english keywords instead of symbols. As in ABC, a block in Python is started with a colon, and ended by changing the spacing.

```
def block(): # startblock
  statement_one()
  statement_two()
# endblock
outer_statement()
```

Python has functions as first-class values. This feature is inspired by functional programming languages such as LISP and Haskell. The example for generators also contains an example of first-class functions.

```
def square(x):
  return x*x
s = square
print s(2)  # prints 4
```

Closures have been talked about since the beginning of computer science. The first example of closures in a programming language is probably from LISP. Overall, closures tend to be most commonly used in functional programming languages. In Python, proper closures were not implemented until version 3, by using the **nonlocal** keyword. The **nonlocal** keyword was eventually backported into Python 2.7.

```
def outer():
  x = 1
  def inner():
    nonlocal x
    x = 2
    print("inner:", x)
  inner()
  print("outer:", x)
```

```
outer()
# inner: 2
# outer: 2
```

Before this addition of this keyword it was possible to access, but not reassign, the outer x value. There are several workarounds for older versions of python, the easiest is to use a mutable object, such as a dictionary, which can be modified from the inner scope.

Generators, as seen in python by using the yield keyword, first appear in the **CLU** language (1975). In Python, calling **yield** will return from the current function, and store the stack frame for the current instance of that function.

```
def count_up():
  x = 0
  while(True):
    yield x
    x += 1

# c is an instance of this function
c = count_up()

print c.next() # prints 0
print c.next() # prints 1
print c.next() # prints 2
…
```

List comprehensions are another structure that is taken from functional programming. Python's list comprehensions are most similar to the list comprehensions from Haskell. Python also goes a step further with comprehensions, and adds set and dictionary comprehensions. These extra comprehensions were originally added to Python 3, but like many other features it was eventually backported into Python 2.7.

Haskell list comprehension:

```
upperString s = [toUpper c | c <- s]
```

Python list comprehension:

```
def upperString(s):
  return [toUpper(c) for c in s]
```

Python Set comprehension:

```
def upperString(s):
  return {c for c in s}
```

Python Dictionary comprehension:

```
def initDict(keys, value):
   return {key: value for key in keys}
```

Another interesting syntactic feature that Python has is context managers.[16] Python appears to be the first language to have context managers built in as a language feature. A context manager is just an abstraction to take care of clean up, after a block of code has been finished. Traditionally this would be done using **try** and **finally**, but by using the **with** statement the clean up code can be part of the object.

```
with open(filename) as f:
   # operate on f
```

In the previous example the file object will automatically be closed when this with block ends.

Types

Implementation Techniques:

Since the beginning the main implementation of Python has has been an interpreter written in C. There have been some implementations of Python that run in other environments. For example, Pypy is a Python just in time compiler written in python. There is an implementation for the Java Virtual Machine written in Python and Java, and a .net implementation written in C#.

Introduction

Python is an interpreted language which is dynamically typed, strongly typed, and uses duck-typing[type1]. As it is an interpreted language, does not have a compilation step and type checking is only performed at runtime. This type checking, however, is strong and does not attempt to convert types implicitly, even in cases which may be trivial to implement.

Python Types

Python does have a type system, however it is not specified syntactically. In languages which are statically typed, a variable is typically a name which is associated with both a type and a value(which also has a type). Type errors in such languages can be detected statically as values which are assigned to a variable can be compared with the type of that variable, and variables which are used together can be compared by their types. In Python, a variable is a name which is associated with only a value, and the value itself is associated with a type[18]. This allows type checking, but it may only occur at the time when variables are used.

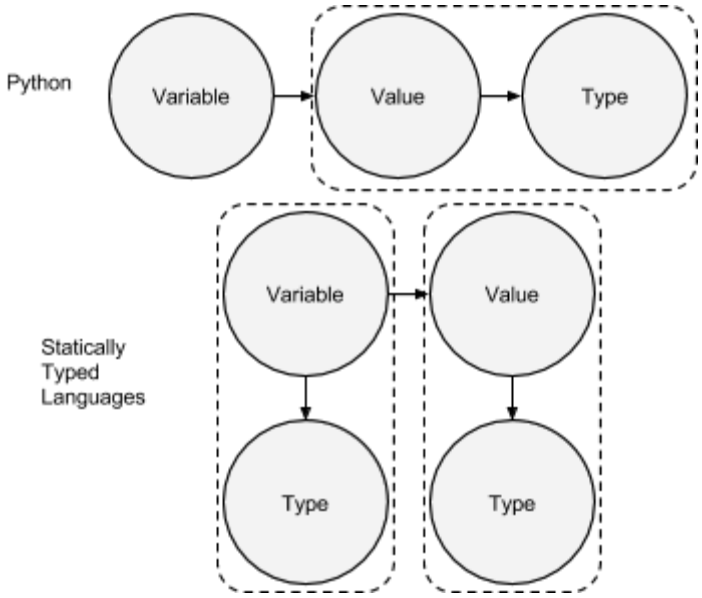

Figure 3: Python Types vs Static Types

Python is a strongly typed language. This seems counterintuitive to some, since variables themselves are not tied to a specific type, but Python requires explicit conversion in every case, and will never implicitly convert values. This seems counterintuitive, but is in line with the Python philosophy that “explicit is better than implicit.”[14] Ultimately, this does make the syntax more complicated than languages where type conversion happens implicitly, but by being strongly typed, Python rejects operations which are not properly defined, rather than attempting to run them. This satisfies another tenet of the Python of philosophy by throwing type errors immediately[14] rather than running optimistically and allowing them to propagate.

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck” is a quote attributed to James Whitcomb Riley. This phrase is where the term “duck typing” comes from. Python is a duck typed language, because types are determined by what members an object has, rather than what it implements or inherits from[2]. The philosophy of Python favors flat layouts to hierarchical ones[14], and duck typing exemplifies this by allowing types to be used interchangeably if they have the same capabilities, without requiring them to implement a shared interface. This is not necessarily a case of favoring implicit declaration to explicit, but rather considering the implementation of a set of members as a sufficient declaration itself.

Type Inference in Python

Python, because it is dynamically typed, does not have type inference. Type inference analyzes the context in which variables are used to determine their type, but in Python, variables do not themselves have a type. Type inference in a statically typed language eliminates redundancy and allows the compiler to figure out what level in a hierarchy of nested types a variable must be. Since Python is dynamic and favors duck typing over large hierarchical type structures, variables do not have a type to infer[18], and there is no large hierarchy to uncover. Type inference therefore is unnecessary in Python and the conditions which favor type inference go against Python philosophy.

Attempts at Supporting Type Inference

As stated above, having type inference generally does not fit the Python philosophy. If you need Python to be statically typed, then either you're using Python contrary to how it is intended to be used, or for the task you're trying, you really should not be using Python at all. However, there are indeed cases when we do need to give up dynamic typing, and people have come up with various kinds of solutions. Sometimes they do achieve the goals, but they always have some major drawbacks or trade-offs.

One of the most common ones is probably RPython. RPython is a subset of Python with a lot of restrictions (which is what the “R” stands for in its name). Its goal is to be a static, compiled, thus a sped up version of Python, while still maintaining the Pythonic style code.

One of the many restrictions is use of static variables:

"Variables should contain values of at most one type as described in Object restrictions at each control flow point, that means for example that joining control paths using the same variable to contain both a string and an int must be avoided." [6]

Type inference is supported on RPython, and it does make the language faster. However, the trade-off is not trivial. First, although its name has the word "Python", its code is not compatible with normal Python code. It can be run on a Python interpreter, but that won't increase its speed; for a higher speed, RPython code must be compiled. Also, for everything RPython is good at, there are already many more mature languages for the same job. And obviously, since it is no longer dynamically typed, its functionality is a lot narrower than normal Python.

Similar efforts to RPython, such as Starkiller, an ideal but unfinished project, or CPython, a strongly typed language, whose goals is to produce a different flavor of Python, all have similar drawbacks to RPython.

There are other reasons to try and support type inference on Python. One example, mentioned in John Aycock’s article, is to translate Python code into Perl to “supply an alternative for Python programmers who are forced to work exclusively with Perl.” [3] The next code block is an example (example taken directly from Aycock's paper [3]):

Python	Perl

`a = 123`
`b = [a,456]`
`c = {‘yyj’:a}`
`print a,`
	`$a = 123;`
	`@b = ($a,456);`
	`%c = (‘yyj’ => $a);`
	`print $a;`

Equivalent Code Block in Python and Perl

Obviously in this situation, it is very important to know the types in Python, so it can be correctly translated into corresponding Perl code.

Aycock proposed a strategy for type inference for Python. The core idea is called "aggressive

type inference (ATI)[3]". The key idea of it is that:

"Giving people a dynamically-typed language does not mean that they write dynamically-typed programs."[3]

The central idea for ATI is very similar to the previous example: if you want type inference support for Python, you'll have to give up dynamic types. Inferred types and dynamic types are still mutually exclusive, and there is no existing hybrid solution yet. (Starkiller project is trying, but it was never implemented due to technical difficulties.)

Summary

Python has a strong, dynamic type system which, by dissociating a variable with a type but throwing errors rather than silently converting types allows for a system which is flexible, but not too flexible. By requiring explicit type conversions, Python solves the problem found in other dynamic languages where implicit type conversion allows invalid data to propagate until it is no longer able to be processed. This causes bugs which must be tracked down to their root and is against the Python philosophy. Python’s lack of an inferred type system does cause any issues as Python’s dynamic type system fits its philosophy well.

References

[1] About Python (Undated) Python [Online] Viewed 2013 Oct 2. Available: www.python.org/about

[2] Adding Optional Static Typing to Python (Jan. 2005) All things Pythonic [Online] Viewed 2013 Oct 2. Available: http://www.artima.com/weblogs/viewpost.jsp?thread=86641

[3] Aggressive Type Inference (Nov. 1999) Python [Online] Viewed 2013 Oct 2. Available: http://www.python.org/workshops/2000-01/proceedings/papers/aycock/aycock.html

[4] Bill Venners The Making of Python (January 2003). Artima developer [Online] Viewed 2013 October 2. Available: http://www.artima.com/intv/pythonP.html

[5]Brett Cannon, Design of the CPython Compiler (February 2005). Python Programming Language [Online] Viewed 2013 October 2. Available: http://www.python.org/dev/peps/pep-0339/#abstract

[6] Coding Guide (Undated) PyPy Documentation [Online] Viewed 2013 Oct 2. Available: http://doc.pypy.org/en/latest/coding-guide.html#overview-and-motivation

[7] Dinil Divakaran Compiler Design with Python (June 2002). Linux gazette [Online] Viewed 2013 October 2. Available: http://linuxgazette.net/issue79/divakaran.html

[8] Full Grammar Specification (Undated). Python [Online] Viewed 2013 Oct 2. Available: http://docs.python.org/dev/reference/grammar.html

[9] Lutz, M., "The Dynamic Typing Interlude," in *Learning Python*, 5th ed., USA: O'riley, pp. 175–186, Jun. 2013.

[10] Owen Cliffe, T-Diagrams (February 2002). University of Bath [Online] Viewed on 2013 . Available:http://www.cs.bath.ac.uk/~occ/comp0029/t_diagrams.shtml

[11] Pérez, F.; Granger, B.E.; Hunter, J.D., "Python: An Ecosystem for Scientific Computing," *Computing in Science & Engineering* , vol.13, no.2, pp.13,21, March-April 2011 doi: 10.1109/MCSE.2010.119

[12] PyPy Frequently Asked Questions (2013). PyPy 2.10 Documentation [Online] Viewed 2013

August 13. Available: http://doc.pypy.org/en/latest/faq.html

[13] Python Developer's Guide (2012). Python v3.3.2 Documentation [Online] Viewed 2013 October 2. Available: http://docs.python.org/devguide/

[14] The Zen of Python (Undated). Python Philosophy [Online] Viewed 2013 Oct 2. Available: http://www.python.org/dev/peps/pep-0020/

[15] Type inference for Python (Undated) Lambda the Ultimate [Online] Viewed 2013 Oct 2. Available: http://lambda-the-ultimate.org/node/1519

[16] van Rossum, G., Coghlan, N. “PEP 343 -- The ‘with’ Statement” [online]. http://www.python.org/dev/peps/pep-0343/ (Accessed:02 October 2013)

[17] van Rossum, G., “The History of Python” [online] 2009 - 2011. http://python-history.blogspot.ca/2009/01/personal-history-part-1-cwi.html.(Accessed:02 October 2013)

[18] Why is Python a dynamic language and also a strongly typed language? (Feb. 2012) Python Wikipedia [Online] Viewed 2013 Oct 2. Available: https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language