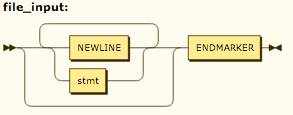
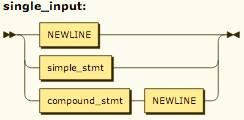
**Question 1: Grammar & Rail Diagrams**

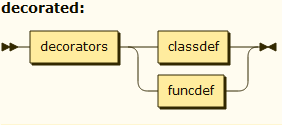
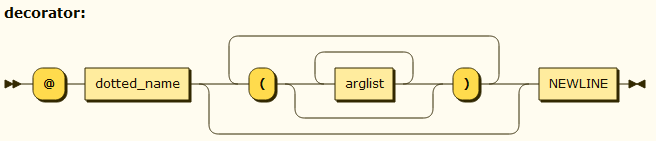
Start symbols for Python’s Grammer consists of:

single\_input: NEWLINE | simple\_stmt | compound\_stmt NEWLINE  
file\_input: (NEWLINE | stmt)\* ENDMARKER  
eval\_input: testlist NEWLINE\* ENDMARKER



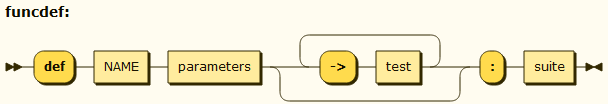
A function decorator is applied to a function definition by placing it on the line before that function definition begins:

decorator: '@' dotted\_name [ '(' [arglist] ')' ] NEWLINE  
decorators: decorator+  
decorated: decorators (classdef | funcdef)



Function definitions and parameters:

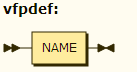
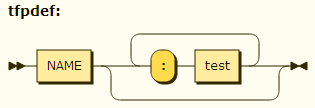
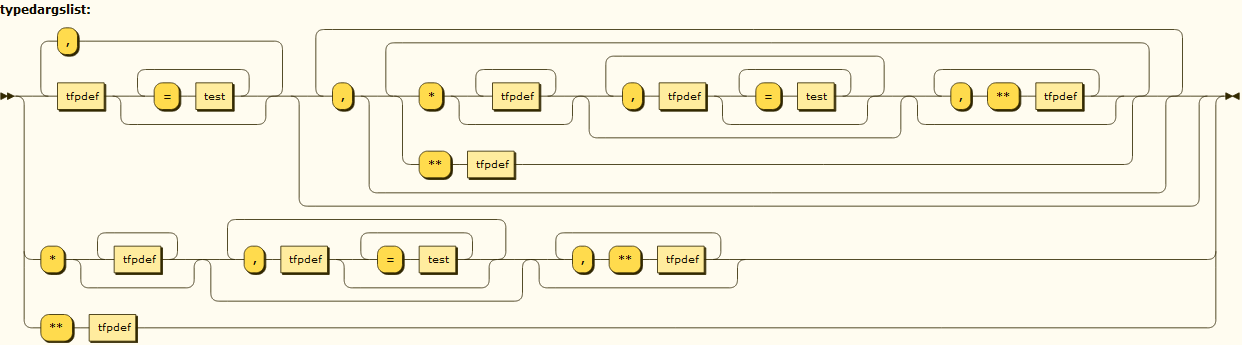
funcdef: 'def' NAME parameters ['->' test] ':' suite  
parameters: '(' [typedargslist] ')'



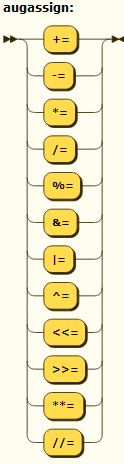
Variable assignments; notice that typedargslist and varargslist have the same grammar, therefore it is redundant to draw both of it’s railroad diagrams :

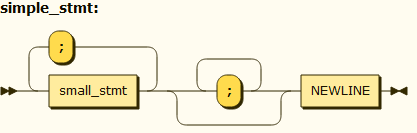
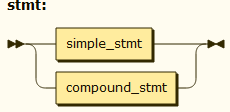
typedargslist: (tfpdef ['=' test] (',' tfpdef ['=' test])\* [','  
 ['\*' [tfpdef] (',' tfpdef ['=' test])\* [',' '\*\*' tfpdef] | '\*\*' tfpdef]]  
 | '\*' [tfpdef] (',' tfpdef ['=' test])\* [',' '\*\*' tfpdef] | '\*\*' tfpdef)

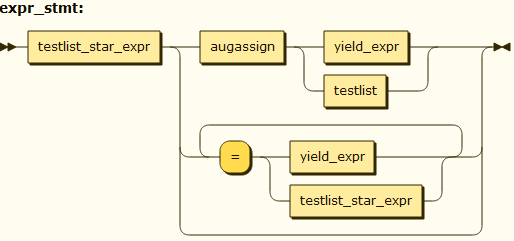
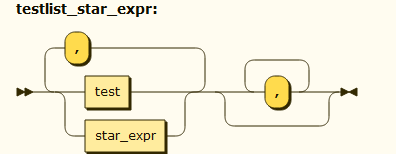
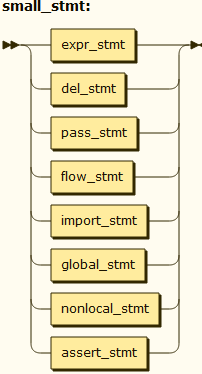
tfpdef: NAME [':' test]  
varargslist: (vfpdef ['=' test] (',' vfpdef ['=' test])\* [','  
 ['\*' [vfpdef] (',' vfpdef ['=' test])\* [',' '\*\*' vfpdef] | '\*\*' vfpdef]]  
 | '\*' [vfpdef] (',' vfpdef ['=' test])\* [',' '\*\*' vfpdef] | '\*\*' vfpdef)  
vfpdef: NAME



Statements:

stmt: simple\_stmt | compound\_stmt  
simple\_stmt: small\_stmt (';' small\_stmt)\* [';'] NEWLINE  
small\_stmt: (expr\_stmt | del\_stmt | pass\_stmt | flow\_stmt |  
 import\_stmt | global\_stmt | nonlocal\_stmt | assert\_stmt)  
expr\_stmt: testlist\_star\_expr (augassign (yield\_expr|testlist) |  
 ('=' (yield\_expr|testlist\_star\_expr))\*)  
testlist\_star\_expr: (test|star\_expr) (',' (test|star\_expr))\* [',']  
augassign: ('+=' | '-=' | '\*=' | '/=' | '%=' | '&=' | '|=' | '^=' |  
 '<<=' | '>>=' | '\*\*=' | '//=')

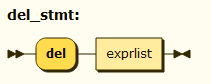
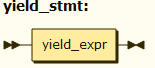
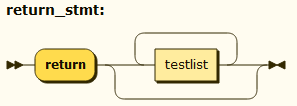
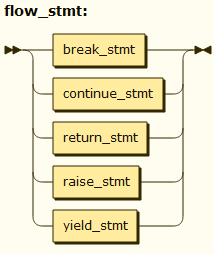
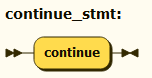
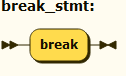
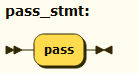


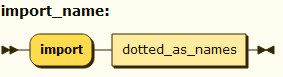
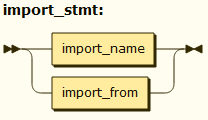
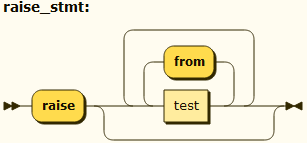


For normal assignments and additional restrictions enforced by the interpreter:

del\_stmt: 'del' exprlist pass\_stmt: 'pass'

flow\_stmt: break\_stmt | continue\_stmt | return\_stmt | raise\_stmt | yield\_stmt

break\_stmt: 'break' continue\_stmt: 'continue'  
return\_stmt: 'return' [testlist] yield\_stmt: yield\_expr  
raise\_stmt: 'raise' [test ['from' test]] import\_stmt: import\_name | import\_from  
import\_name: 'import' dotted\_as\_names

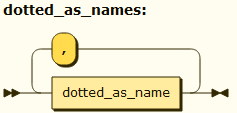


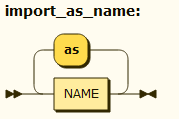
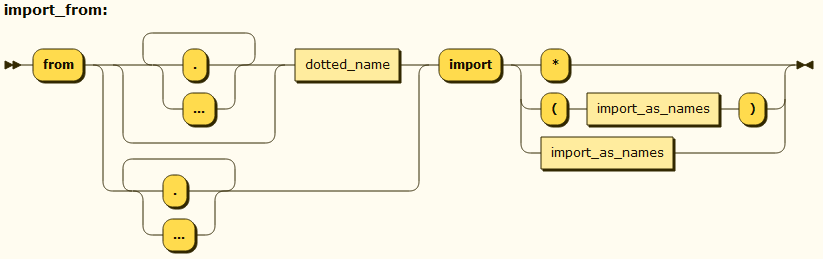
import\_from: ('from' (('.' | '...')\* dotted\_name | ('.' | '...')+)  
 'import' ('\*' | '(' import\_as\_names ')' | import\_as\_names))  
import\_as\_name: NAME ['as' NAME]

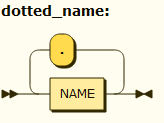
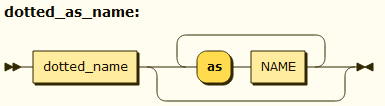
dotted\_as\_name: dotted\_name ['as' NAME]

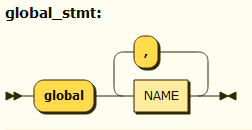
import\_as\_names: import\_as\_name (',' import\_as\_name)\* ','

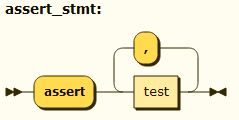
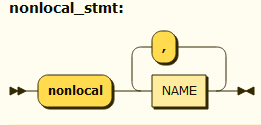
dotted\_as\_names: dotted\_as\_name (',' dotted\_as\_name)\*

dotted\_name: NAME ('.' NAME)\*



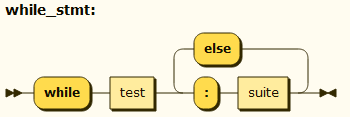
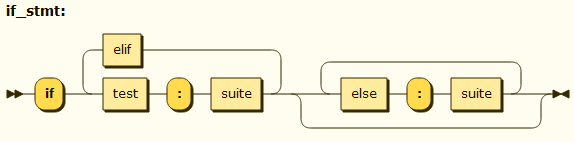
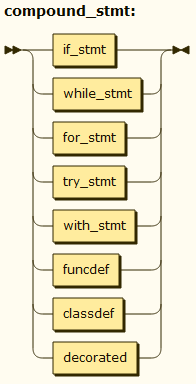


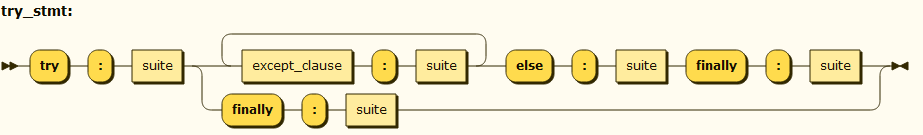
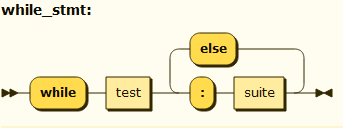
global\_stmt: 'global' NAME (',' NAME)\*  
nonlocal\_stmt: 'nonlocal' NAME (',' NAME)\*  
assert\_stmt: 'assert' test [',' test]

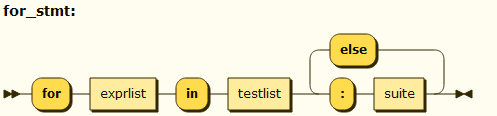


Compound Statements and Loops:

compound\_stmt: if\_stmt | while\_stmt | for\_stmt | try\_stmt | with\_stmt | funcdef | classdef | decorated  
if\_stmt: 'if' test ':' suite ('elif' test ':' suite)\* ['else' ':' suite]  
while\_stmt: 'while' test ':' suite ['else' ':' suite]  
for\_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]

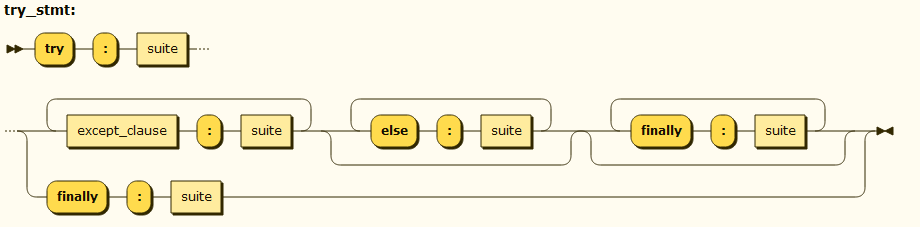
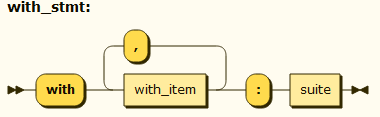


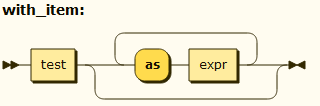




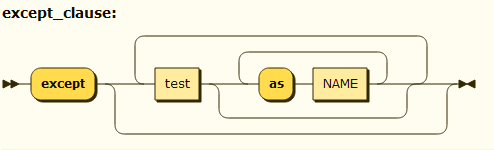
try\_stmt: (‘try’ ‘:’ suite((except\_clause ‘:’ suite)+[‘else’ ‘:’ suite] | ‘finally’ ‘:’ suite))

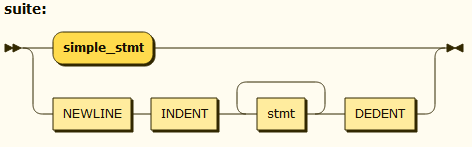
with\_stmt: 'with' with\_item (',' with\_item)\* ':' suite  
with\_item: test ['as' expr]



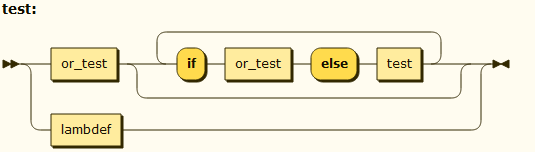
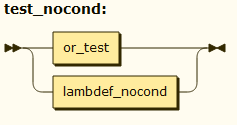
except\_clause: 'except' [test ['as' NAME]] 

suite: simple\_stmt | NEWLINE INDENT stmt+ DEDENT



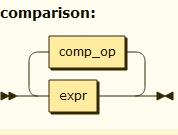
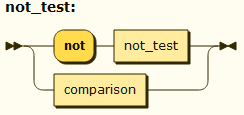
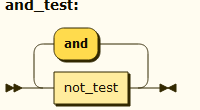
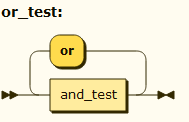


test: or\_test ['if' or\_test 'else' test] | lambdef  
test\_nocond: or\_test | lambdef\_nocond



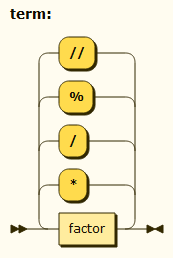
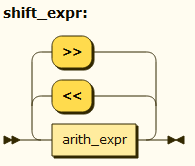
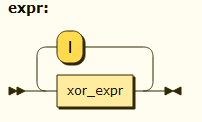
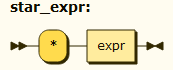
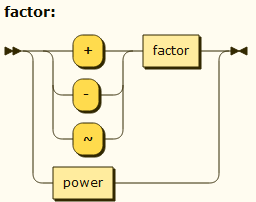
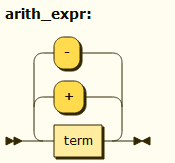
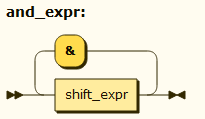
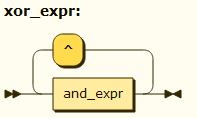
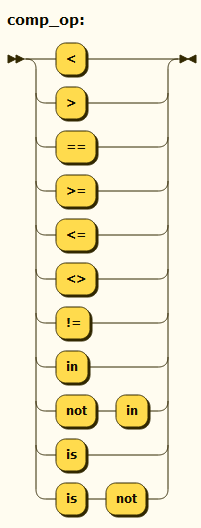
lambdef: 'lambda' [varargslist] ':' test  
lambdef\_nocond: 'lambda' [varargslist] ':' test\_nocond  


or\_test: and\_test ('or' and\_test)\*  
and\_test: not\_test ('and' not\_test)\*  
not\_test: 'not' not\_test | comparison  
comparison: expr (comp\_op expr)\*

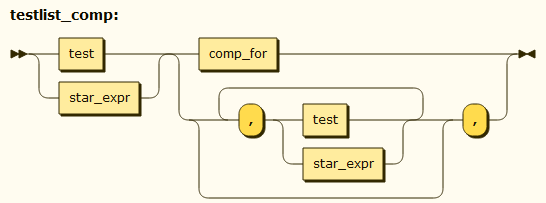


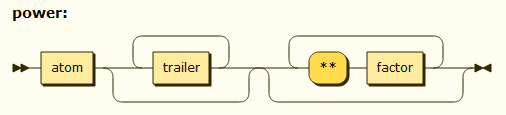
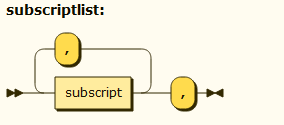
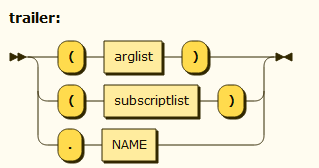
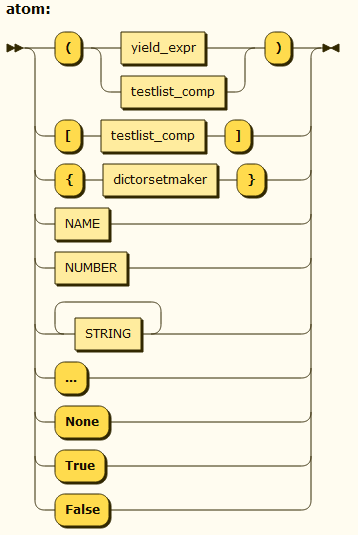
comp\_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'  
star\_expr: '\*' expr  
expr: xor\_expr ('|' xor\_expr)\*

xor\_expr: and\_expr ('^' and\_expr)\*  
and\_expr: shift\_expr ('&' shift\_expr)\*  
shift\_expr: arith\_expr (('<<'|'>>') arith\_expr)\*  
arith\_expr: term (('+'|'-') term)\*  
term: factor (('\*'|'/'|'%'|'//') factor)\*  
factor: ('+'|'-'|'~') factor | power



power: atom trailer\* ['\*\*' factor]  
atom: ('(' [yield\_expr|testlist\_comp] ')' |  
 '[' [testlist\_comp] ']' |  
 '{' [dictorsetmaker] '}' |  
 NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')  
testlist\_comp: (test|star\_expr) ( comp\_for | (',' (test|star\_expr))\* [','] )

trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME  
subscriptlist: subscript (',' subscript)\* [',']  


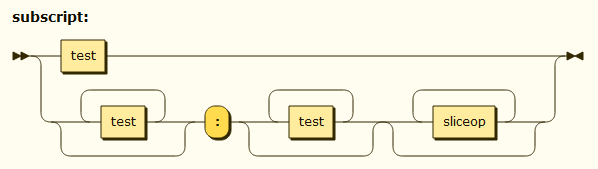
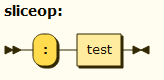


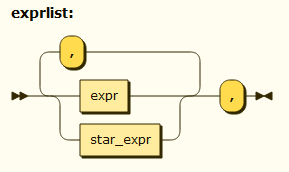
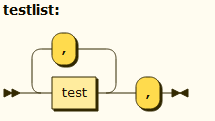
subscript: test | [test] ':' [test] [sliceop]

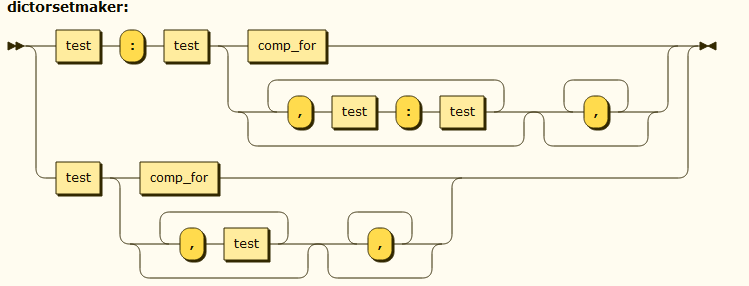
sliceop: ':' [test]

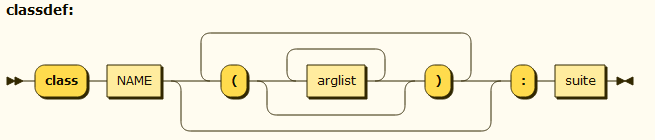
exprlist: (expr|star\_expr) (',' (expr|star\_expr))\* [',']

testlist: test (',' test)\* [',']

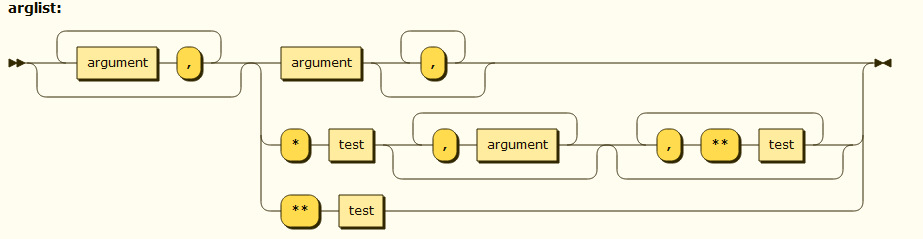




dictorsetmaker: ( (test ':' test (comp\_for | (',' test ':' test)\* [','])) |  
 (test (comp\_for | (',' test)\* [','])) )  
classdef: 'class' NAME ['(' [arglist] ')'] ':' suite  


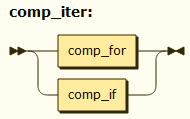
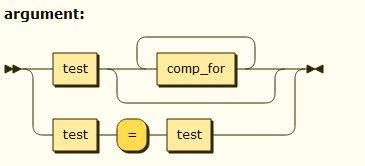


arglist:(argument ',')\*(argument [',']|'\*' test(',' argument)\* [',' '\*\*' test]|'\*\*' test)

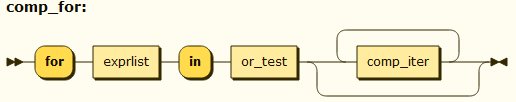
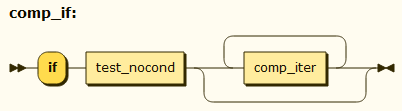


argument: test [comp\_for] | test '=' test

comp\_iter: comp\_for | comp\_if

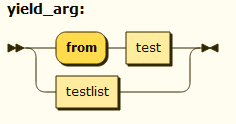
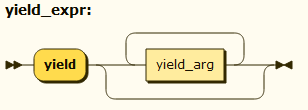
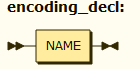


comp\_for: 'for' exprlist 'in' or\_test [comp\_iter]  
comp\_if: 'if' test\_nocond [comp\_iter]



encoding\_decl: NAME

yield\_expr: 'yield' [yield\_arg]  
yield\_arg: 'from' test | testlist

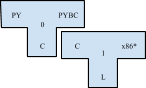


**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

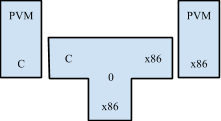
**Question 2: Python Implementations & T-Diagrams**

Python comes in many flavours. The main implementation of Python, today, is CPython, a reference implementation. CPython is based on the original compiler written by Guido van Rossum. He chose to use C because it had recently been standardised (in 1989), and he was able to distribute his new language easily this way. This implementation translates the Python input code into Python Byte Code (.pyc files). This byte code is run on a Python Virtual Machine (much like the JVM), which itself is written in C. Other implementations include Jython, a Python to JVM (Java Byte Code) translator, IronPython, Microsoft’s Python to .NET translator, and PyPy, an interpreter and JIT compiler for Python. These implementations are written in a number of different languages. Jython, for example, is written in Java while IronPython is written in C#.

CPython: CPython is written in C. Like most Python implementations, CPython takes Python source code and translates it into Python Byte Code, .pyc files. This is opposed to translating directly into machine executable code such as x86 or ARM. These .pyc files are taken and used by the Python Virtual Machine, a C program which has been compiled to run natively on the system of preference. The Python group even points out that, even if there is not a Python executable available for your system on the web, if there is a C compiler for your system, and you can compile the source yourself to work on your system.



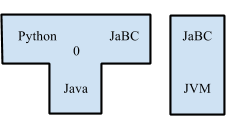
T-Diagram #1 is a C compiler written in some language(L).



T-Diagram #2 shows how a PVM written in C is compiled to work on, in this case, an x86 machine. So long as your system has a C compiler written for it, Python can be run on that machine (by compiling Python source into a PVM using the C compiler written for that machine)

IronPython: this implementation of Python compiles into Microsoft CLR, a language used by Microsoft’s .NET framework. Again, this is not compiled directly into native machine code, instead being translated into an intermediate language and handled from there by existing tools. This implementation is written largely in C#, although certain parts are written in Python itself and handle automatic code generation. IronPython is an executable provided by Microsoft, compiled to run on Windows and even Android.

Jython: this implementation takes the Python source code and translates it directly into Java Byte Code (instead of Python Byte Code). This code can be run on the Java Virtual Machine. One benefit to this implementation is that the entire Java class library can be used in place of the Python library.



T-Diagram #3 shows Jython. It translates Python into Java Byte Code which can then be run via the Java Virtual Machine.

PyPy: a Python interpreter for the Python language that focuses on speed and efficiency. It accomplishes this by using RPython, a subset of Python, and a JIT(Just In Time) Compiler, which compiles Python instructions directly into Machine Code.

Shed Skin: this is an experimental Python compiler that translates it into C++. To use this compiler, however, the developer is restricted to a static subset of Python.

Pyjamas: this implementation allows developers to write their web applications in Python, and then translates them into Javascript. It is equipped with an API and a GUI as part of the software.

**Question 3: The History of Python**

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum in the Netherlands as a successor of a language called ABC("Python documentation"). Python has been influenced by many different languages, and taken some of the best features of each. Some of the biggest influences were C, Perl, and TCL.

“One of the design rules for Python has always been to beg, borrow or steal whatever features I liked from existing languages, and even though the design of C is far from ideal, its influence on Python is considerable.”(van Rossum) Some of these features that were “borrowed” from C include C’s definitions of literals, identifiers operators and expression syntax which have been copied almost verbatim. Like C, Python utilizes a mixed mode arithmetic, where it uses integer division and floating point division to give different results. Other syntax taken from C includes, the break, continue, and return statements. Although Python borrows plenty of ideas from C, Python code and C look very little alike. In Python “there are no curly braces, fewer parentheses, and absolutely no declarations”. This makes Python code much simpler to read and write than C code. Python also eliminates the pointer type that is used so frequently in C. Pointers functionality is instead taken over by Python’s more powerful data types. In fact, Python takes more than just syntax and semantics from C, but Python’s built-in operations are coded in C. This helps Python to hold onto some of the performance benefits that C offers while still giving the programmer the ease of use Python is famous for.

The language which had the greatest influence on Python was ABC. Like Python, ABC was designed to be a language whose code looks simple and is easy to learn. Python code uses a similar structure to ABC. Like ABC, Python was designed with elegant syntax in mind. This makes code for both languages simple and easy to read. Much of Python’s syntax seems to be lifted straight from ABC. Neither language uses semi-colons to denote the end of a line. In addition, neither language makes use of brackets or braces to designate functions, condition statements or loops. Instead, these languages use a colon to mark the end of function declarations and conditional statements and loops. Python also uses ABC’s forced indentation and strong data types as well. Python does go beyond ABC in a couple aspects. These include its Object-Oriented nature, its use of modules, its exception mechanisms, and its extensivity. Doing so costs Python only a small degree of elegancy.

Perl, the well known scripting language has many similarities to Python. Such similarities include but are not limited to, “powerful string, list and dictionary data types, access to high- and low-level I/O and other system calls, syntax”(van Rossum). Like Perl, Python is also suited for smaller scripts as well as larger programs. Both languages can be extended with code written in C. Python however extends the power of some of Perl’s types such as dictionaries. In Perl, dictionaries may only contain strings and integers as elements, however in Python, any type may be used. Perl also differs from Python by requiring the programmer to prefix variables with a special character to indicate their type. Python improves on Perl in the implementation of recursive solutions. Recursive data structures can be difficult for novice programmers to understand in Perl whereas in Python, they are much more simple to read and understand.

Python borrowed concepts from a handful of other languages as well. Python’s modules and exceptions were provided by Modula-3. Slicing operations were taken from Icon. The constructor, destructor, and user defined operators were provided from C++. The notion of classes being first class as well as run-time type checking came from SmallTalk.

**Question 4: Type Inference**

Type inferencing is a task completed by the compiler of a language. The compiler attempts to identify which types are most accurate for a program that does not specify the types of its variables. Type inferencing is difficult to implement in Python due to the nature of dynamic languages. Because type checking occurs at run time rather than compile time, there is no way to infer the types before the code executes. Although atomic type inference is possible (atomic types are standard in the language and will not change from compile-time to run-time), abstract types are unknown to the language and cannot be ‘type checked’ before run-time.

There are two frequently used type inference algorithms: Hindley-Milner and Cartesian product. While these are widely used, neither can be applied to Python code. Complete information on control flow of an application is required for type inference. A lack of this may lead to invalidation of types. All type information or source code must be present to have complete control flow information. This may not be the case for interpreted languages such as Python.

There have been many attempts to implement type inference into Python. One such attempt, Starkiller (written by Michael Salib), achieves type inference in Python by converting the source code into C++ and using the Cartesian Product algorithm to determine types. Although Starkiller technically allows Python code to be executed using type inference, it is not compatible with dynamic code or dynamic modules which are standard Python features.

Another example is Psyco which was developed by Armin Rigo. After Rigo finished, it was maintained and further developed by Christian Tismer. Psyco is an extension module for Python that may greatly decrease the runtime of any Python code. It is written in C and outputs x86-based code. It speeds up run time by at least two times, but is expected to perform at 4x the speed. A fault in Psyco’s design is that it uses a lot of memory. Although development of Pysco has ceased, it’s author has created a follow-up known as Pypy.

Type inference provides many advantages regardless of language. Code can be produced more efficiently when types are inferred. This is especially true of function calls, which gain much flexibility when argument types aren’t strictly defined. Obviously, languages with type inference require less writing by the programmer, resulting in more concise code. Additionally, compilers with type inference are able to catch ‘type mismatch’ problems before they cause errors during run-time.

**Bibliography:**

<http://www.artima.com/intv/pythonP.html> // breif history

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

. *Python documentation*. N.p., n.d. Web. 2 Oct 2013. <http://docs.python.org/2/license.html>.

van Rossum, Guido. "An Introduction to Python for UNIX/C Programmers." . N.p., 1993. Web. 2 Oct 2013. <http://liuj.fcu.edu.tw/net\_pg/python/Intro-Python.pdf>.

<http://stackoverflow.com/questions/8452396/does-pypy-translate-itself>

<http://stackoverflow.com/questions/2591879/pypy-how-can-it-possibly-beat-cpython>

<http://psyco.sourceforge.net/index.html>

<http://mike.salib.com/writings/thesis/thesis.pdf>

<https://wiki.csc.calpoly.edu/590/raw-attachment/wiki/Reading/cannon.pdf>