Adding Const Keyword to Python

Elizabeth Duncan, Cypress Frankenfeld, Aaron Greenberg, Geoff Pleiss

Abstract

The goal of this project is to add a new keyword to Python's grammar by modifying the CPython implementation of the Python programming language. We investigated possible types of a const keyword and weighed the pros and cons of each. This section is paramount in order to make any design decision for future work on this const keyword. For the actual implementation, we described our desired syntax in the abstract syntax description language and implemented a `const` *keyword in the script that generates an abstract syntax tree and the actual compiler script that generates bytecode. We discovered that, due to the dynamic state of Python, implementing optimization based on the const keyword would require significant changes to CPython, including numerous case-specific fixes.*

Introduction

One of the key features of Python—and many similar languages—is dynamic typing. Because types are not specified by the program, they are determined at runtime instead of at compile time. As a result, the Python compiler is capable of limited amounts of code optimization.

For this project, we modified the C implementation of Python (CPython) to include a keyword for declaring constant variables. This modification is based on two major goals. First of all, as a team we decided that we wanted to learn about compilers, and this project would help us gain knowledge of the steps of the compilation process. Secondly, we wanted to see what performance gains were possible by implementing a static typing feature in a dynamic language. More specifically, we wanted to know if declared constant variables decrease the runtime for a program.

The CPython Compiler

CPython (written in C) is the original, and most commonly used interpreter of Python. To compile Python source into Python bytecode, CPython uses a four-step process. Initially, the source code is parsed into a "parse tree". Then, the parse tree is converted to an abstract syntax tree (AST) using Python's abstract syntax definition language (ASDL). The AST is transformed into a control flow graph (CFG) by creation of an intermediate bytecode representation in a graph structure. The CFG is then flattened and bytecode jump instructions are calculated to replace the edges of the graph.

To make CPython recognize and compile the `const` keyword, we needed to tweak each step in the compilation process. The parser would need to recognize `const` as a legal and valid keyword in Python's grammar. The AST would need an abstract syntax form of the `const` keyword (defined in the Python ASDL). Finally, there would need to be compilation logic and opcodes to generate functional bytecode from the AST.

The design of CPython is such that some of the required code is automatically generated. Nevertheless, there are some key files that must be manually modified to make the language recognize and respond to the new keyword. The following is a basic outline of the required steps for modifying the CPython compiler[1]:

1. The parse-tree grammar and abstract syntax description language must be modified.
2. The code that generates the AST must be modified to handle the new grammar.
3. The code that creates the CFG and bytecode (the actual compilation code) must be modified to understand the new AST object. This includes defining the appropriate operations (in Python machine code) that must take place when the keyword is run, and error checking to make sure that the const is not used improperly in the code.

Design

Syntax and Behavior

To declare a const variable, we use the following syntax:

```
const var = expr
```

where *var* is any previously undeclared variable and *expr* is any type of expression that evaluates at compile time. The following are examples of valid expressions:

```
10
"cat" + "dog"
```

Examples of invalid expressions are ones that could not be evaluated at compile time, such as:

```
1 if x < 5 else 2
myfunc(x, y)
```

Regardless of what form the expression takes, *var* would be assigned the value that *expr* evaluates to. Once assigned, *var* could not be reassigned for the lifetime of the variable. However, *var* could still be referenced like a normal variable, and it would have the same scope of a normal variable.

[1] Based on: http://www.python.org/dev/peps/pep-0306/

Location of Const Functionality

There are three different stages where we could implement the const functionality: 1) preprocessing, 2) compilation, or 3) runtime. We have decided to implement a compile-time constant, as this provides an interesting challenge with the possibility of performance enhancements.

Constants that are handled in preprocessing are equivalent to macros in C. Before the code is compiled, the preprocessor replaces every instance of the macro with its associated value.

A compile time constant, unlike a preprocessor macro, is an actual variable stored in the stack. Only one copy of the constant is stored in memory, unlike a preprocessor macro which is duplicated every time it is referenced by the code. Compile time constants must have non-dynamic values -- i.e. they must have values that can be calculated at compile time.

A runtime constant is similar to a compile time constant; however, the value for a runtime constant can be dynamic. An example helps to clarify:

```
const x = 3
const y = "Allen Downey"
```

These could be compile time constants because the compiler can easily tell that x stores a integer of value 3 and y stores a string representation of "Allen Downey". Contrast that example with this:

```
const z = a * func (b)
```

The compiler doesn't know what a * func (b) is without running the program, so this must be a runtime constant.

In choosing how to implement const functionality, we examined a) how challenging it would be to implement and b) the opportunities for performance optimization. While preprocessor macros make code easier to read, and improve the performance of the code by placing literals wherever they are used, but they are not as interesting from a project standpoint because all the work to implement them would be in the preprocessor code, and we would not need to touch the other parts of the compiler. Runtime or compile time constants also both present the opportunity for performance enhancements. We decided that we wanted to implement a `const` keyword for basic, non-dynamic assignment statements. Because such constants can be handled at compile time, we would just need to modify the compiler.

Optimizations

Though we did not have enough time after implementing the functionality for a constant keyword, we had originally planned to build in some optimizations using variables declared constant. We researched different compiler optimizations (especially in the area of data-flow optimizations—optimizations related to how data flow through a program during run-time), and decided to implement constant folding and, if possible, (because we were optimistic) constant propagation. Constant folding is an optimization technique where the compiler simplifies expressions assigned to constants because it knows they will not change. These simplifications might be arithmetic of literals or string concatenation.

For example, if the compiler reads the statements

```
const x = 10 * 20 * 30
const y = “cat” + “dog”
```

It can “fold” these expressions like so:

```
const x = 6000
const y = “catdog”
```

Interestingly, we learned that the CPython compiler already does folding of literals and basic variables (e.g., not variables assigned to function return statements). We learned by using Python’s disassembler module to disassemble some Python bytecode. We learned that for every line that included basic arithmetic or string concatenation, the compiler had a single store instruction that stored the result of the operation.

Constant propagation involves substituting known constants throughout the code during compile time to simplify it. Here’s an example from Wikipedia:

```
const x = 14
y = 7 - x / 2
return y * (28 / x + 2)
```

We propagate x to get

```
const x = 14
y = 7 - 14 / 2
return y * (28 / 14 + 2)
```

Then by folding these expressions, we get

```
const x = 14
y = 0
return 0
```

If our compiler was *really* smart, it would remove the constant and y altogether. Constant propagation and folding are often used together iteratively to simplify the code. We will start with constant folding because it is a simpler execution, and time permitting, we will move on to implement constant propagation.

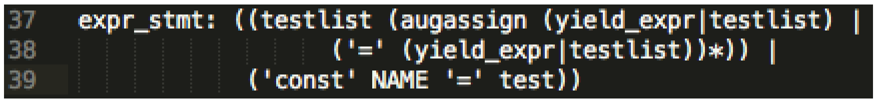
An important note: constant folding can occur without the use of a `const` keyword. If upon inspection of the CPython compiler we discover that constant folding already occurs, we will instead go straight to implementing constant propagation.

Ultimately, the decision of which optimization and type of constant should depend on the expected use of the keyword. Specifically, if the keyword will be used in multiple arithmetic operations, one should implement folding. If these arithmetic operations will be compiled once and run multiple times, a compile-time constant should be used. Such a constant would speeds up the runtime because the interpreter runs less instructions while the program is running, but compilation time increases.

Implementation

Parsing and Grammar

CPython defines its grammar in a simple grammar syntax file that is then auto-generated into a multiple C scripts that control the functionality of Python. Therefore, by adding a simple addition to this grammar file, one can flesh out make additions to multiple files and right near a hundred lines of code.



```
expr_stmt: ((testlist (augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist))*)) |
                ('const' NAME '=' test))
```

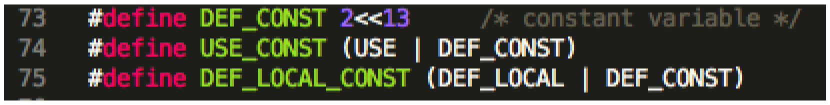
We added our const keyword to the definition of the expression statement because all assignment statements are expression statements. Therefore, we were able to implement expr_stmt's already existing infrastructure to handle any const assignment. It is also important to note that with this const definition, a legal const assignment must contain a '=' or the compiler will throw an error. Additionally, a const statement can only be used to assign a value to a name (only a variable).

The Const Bit

Memory management in python is handled by two data structures: the arena and symtables. The arena keeps track of all the variables that are in use, all of the references to each variable, and information about that variable (type, scope, etc.). The arena is primarily dealt with in the runtime environment, so we did not deal with it at all.

Symtables keep track of all the variables used in a namespace, and determines how they are used. The symtable divides the code into blocks—which is a collection of code that contains a group of variables (i.e. a function, a for loop, etc). For each variable that is used in a block of code, the symtable knows how it is used (is it being assigned to, or just read) and the scope of the variable (was it defined inside or outside the block). The purpose of the symtable is for the compiler to check that no variables are used in illegal ways—i.e. that the line `x += 1` is not run before x is defined, or that `del y` is not called if y is a defined outside the block's scope. To keep track of which variables were consts, we decided to add additional information to the variables in the symtable.

CPython utilizes flags that are bit-mapped to mark and recognize variables. To earmark any variables declared as const (and therefore should not be reassigned), we utilized this flag system. Each variable in the symtable contains an integer flag. Every bit of the flag is used to describe the scope and operations performed on the variable. Many of the lower bits were taken to earmark global assignments, local assignment, function parameter assignment, etc. For our const bit, we assigned the 15th bit to be 1 if the variables was a constant, and 0 otherwise. This bit choice was made because it was significantly above any known bit assignments and thus would not rewrite any pre-assigned bits. The const bit is shown below:



```
#define DEF_CONST 2<<13      /* constant variable */
#define USE_CONST (USE | DEF_CONST)
#define DEF_LOCAL_CONST (DEF_LOCAL | DEF_CONST)
```

When a variable is declared as a const (regardless if this statement is legal or not), we store the variable in the symtable with the const bit turned on. If we are then referring to the variable later in the scope, we will recognize that it was declared const.

The one problem with this implementation is that it does not notice variables that were defined outside the scope. The symtable knows all the variables that are used in the block, but if a variable was defined outside the block (i.e. a global or a closure variable), then the symtable won't know how the variable was used before. Thus it is hard with our approach to tell if variables defined outside the scope are constant variables or not. We'll talk more about this in the discussion section.

The Compile-Time Decision-Maker

All name operations (ensuring that no variables are referenced before assignment, making sure that variables are assigned the proper scope, etc) occur at compile time. Every time a variable name is referenced, the compiler looks up the variable the symtable. In particular, the compiler looks up the scope of the variable and the type of operation that is going to be performed on the variable.

We took advantage of this pre-existing structure to implement extra case statements that checked the const bit to determine if actions were legal. If the const bit is 0, it is safe to assign that variable. If the bit is 1, the compiler throws an error because the compiler is trying to reassign a const. Although cumbersome, this is the most effective method of keeping a const variable safe.

Discussion

In most situations, our const keyword had the desired functionality. In certain cases, we ran into limitations of working with a dynamic language, so we were not able to get the desired functionality. In addition, our implementation decreased performance a little because of the additional checking that had to be performed. We will discuss how we could potentially address this with code optimization.

Our implementation worked the best when we were defining and using const variables in the same scope. For example, the compiler threw an error for this function (trying to reassign a variable that has been declared const):

```
def a():
    const x = 6
    x = 5
```

Similarly, the compiler would not compile this function (trying to make an already declared variable a const)

```
def b():
    y = 8
    const y = 3
```

Once declared, const variables would function as normal variables -- with the exception that they could not be assigned to. For example this function worked as expected:

```
def c(x):
    const z = 3
    return x + z
```

In simple situations like the ones above, we had the desired functionality. However, there were many cases where our const statement didn't work. Here are some problems that we noticed with our implementation:

1) **Does not work at top-level namespace.** When we attempted to create a const variable at the top-level namespace (i.e. just typing code into the interpreter), the compiler did not throw an error when we tried reassigning it. At the top-level namespace, the python symtable uses a new block for each line of execution. A block refers to a group of code that shares a collection of variables. Since each line is viewed as its own block, python determines at runtime (not compile time) if variable names have been defined. Therefore we were not able to implement compile-time const functionality at this level.

2) **Global/closure const variables aren't always recognized.** As an example, let's examine the following function:

```
const z = 5
      def d():
            global z
            z += 1
            return z
```

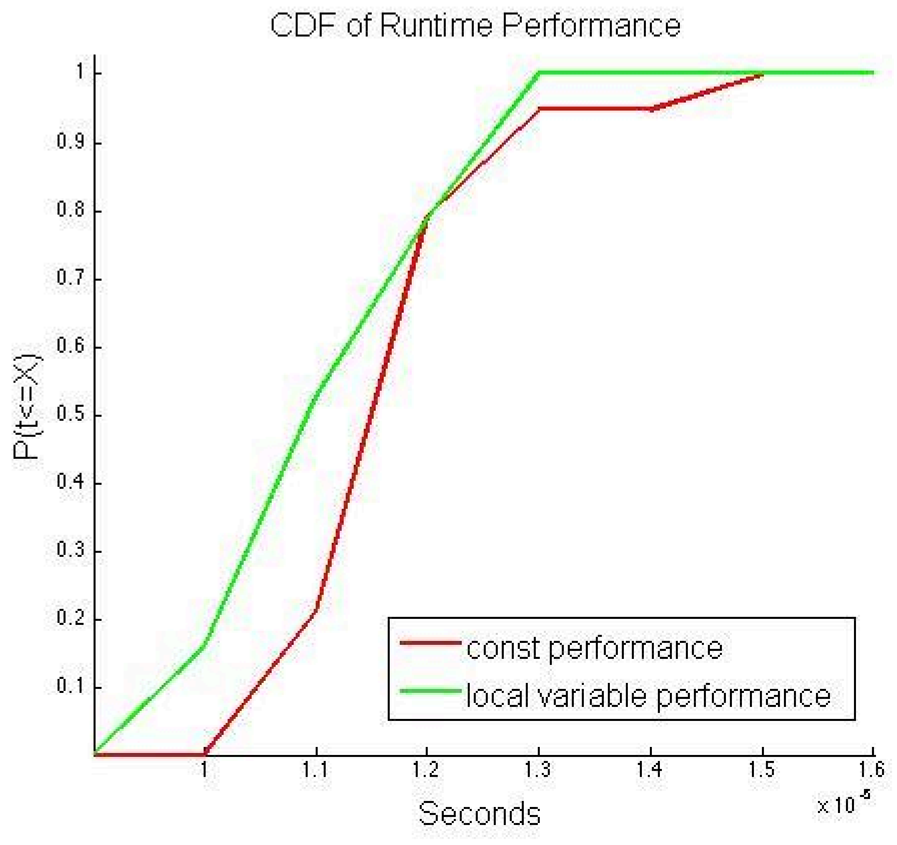
Because we define z as a constant, the desired behavior we'd like to see from this function is that an error would be thrown when we try to reassign z inside the function. However, when we ran this function, no error was thrown. Again, this has to do with how the simtable deals with global variables. Python recognizes that z will not be defined in the local namespace; however it does not look up how/where z was defined until runtime. Therefore we cannot check during compile-time how z is defined.

We believe that these limitations stem from the fact that Python is a dynamic language. Key to Python's philosophy is the idea that many decisions are made at runtime. As a result, it is not entirely possible to keep track of which variables have been used at a given point during the compilation process. This is made especially difficult with global variables and closure variables. If we are dealing with variables that are only defined and used in a local scope, than our implementation will work since decisions will all be made during compile time. However, as soon as we start mixing scopes, we end up trying to fight the fundamental principles of python.

While it would be possible to modify the runtime code of python to achieve const functionality in all situations, this would not be ideal. Firstly, we chose to do a compile-time implementation, and a runtime implementation would require modifying different parts of CPython. In addition, checking at runtime would not be ideal for performance at all. This sort of implementation would produce the desired functionality, but it would not be practical or applicable at all.

Effects on Performance

With implementation of a const bit, storing and checking for const variables did impact (although minimally) the performance of CPython. To verify this, we ran a script with numerous simple arithmetic calculations—one version with local variables and the other with const variables—multiple times and graphed the results. A graph of the cumulative distribution function distribution is shown below.



As is evident, the performance of the const variable is slightly less than the local variable. This makes sense because the const bit does have to be assigned and checked. However, this lag is barely noticeable; the maximum lag is only 0.001 ms. We did not implement folding and propagation optimizations, so we would want to measure the performance *after* optimizations before deciding how useful the const functionality is.

Conclusion

In doing the project, we were exposed to the main steps that CPython uses to compile Python source into Python bytecode. We tweaked the Python grammar so the parser recognizes the `const` keyword and modified the compiler so the keyword would have the desired functionality.

We were able to implement const functionality for variables that were defined and used in a single scope. However, the functionality did not work when using global or closure const variables, and we did not implement optimizations.

In theory, the use of `const` enables the compiler to perform operations at compile-time and thus avoid extra instructions during runtime. However, because we weren't able to get the const functionality to work in every case, we were not able to implement optimizations. As a result, we made python run slightly slower because of the additional steps it was taking to check over const variables.

Future Work

There are a number of different optimizations possible using a compile time constant keyword. We chose to implement one such optimization but future work on our project would involve increasing the functionality of the keyword. For example, we could optimize the CPython compiler by implementing constant folding and constant propagation. Going further, we could implement sparse conditional constant propagation, which provides a more efficient and accurate way of propagating a constant through the code's instructions by making use a code form known as static single assignment (SSA) form (which basically means that the each variable in the code is only assigned once). We could also optimize for factoring out invariants—for example, if a constant is assigned inside a loop, it can be moved outside the loop, since it will not change over iterations.

Acknowledgements

CPython 2.7.2 Core Development Team
Allen Downey
Guido van Rossum