# Analysis of the Python Programming Language

Stephen McAleese, CASE3

December 2020

## Overview

The purpose of this analysis is to analyze and evaluate each component of the Python programming language and the language as a whole. This analysis is divided into several parts where each part is concerned with a single aspect of Python. The goal is to evaluate each aspect by considering its pros and cons as well as how the Python implementation compares to that of other programming languages and then conclude with a general sentiment on the language.

## Introduction to Python

Python is a high-level, general-purpose and interpreted programming language invented in 1991 by Guido van Rossum [1].

## Python is a High Level Programming Language

A high level programming language is one that is similar to the English language and abstracted from the hardware. To understand what a high level programming language is, it is useful to first examine the idea of a low level programming language.

Assembly language is a low level language because it has little abstraction meaning that the assembly instructions are similar to the machine code instructions that are executed. In assembly, every small detail and instruction has to be specified by the programmer. These extremely specific and explicit instructions are similar to the machine code that a CPU executes so assembly can easily be converted to machine code via an assembler. While assembly is convenient for a CPU to execute, the fact that every minute detail and instruction needs to be explicitly written makes it an unproductive and verbose programming language. The other reason why assembly is a low level programming language is that it is unlike English. Although there are some English words in assembly, an assembly program is very different to a more human-readable set of instructions such as a cooking recipe.

Whereas low level languages are designed to be easily executed by a computer, high level languages are designed to be easier to read and write for human programmers. High level programming languages such as Python have an English-like syntax as well as more abstraction between the hardware and the programming language. Abstraction is the process of removing many small and unimportant details from a system while preserving the most important details. This process is worthwhile because, while a small amount of information is lost, the system becomes much more simple to understand for humans. Instead of specifying every single instruction that the CPU needs to execute, a Python program only involves abstract statements which carry out many CPU instructions for each statement.

For example, here are the assembly and Python implementations of the "Hello World" program.

Python implementation:

print("Hello World")

Assembly implementation [2]:

JMP start

hello: *DB* "Hello World!" ; Variable

*DB* 0 ; String terminator

start:

MOV C, hello ; Point to var

MOV D, 232 ; Point to output

CALL print

HLT ; Stop execution

print: ; print(C:\*from, D:\*to)

PUSH A

PUSH B

MOV B, 0

.loop:

MOV A, [C] ; Get char from var

MOV [D], A ; Write to output

INC C

INC D

CMP B, [C] ; Check if end

JNZ .loop ; jump if not

POP B

POP A

RET

The Python implementation above is significantly more abstract because it involves only the high level instruction to print "Hello World" on the screen without the sub-instructions required to execute this instruction. Therefore, high level programming languages like Python are usually far more productive and almost always result in more succinct, readable code than low level languages.

There are some disadvantages to high level programming languages however. Other high level programming languages need to be compiled before they can be executed. Python needs to be interpreted before it can be executed as it is a scripting language. As this process is controlled by the computer, the programmer has less control over the final machine code implementation of the program. This is usually not a problem but can be an issue when performance is a constraint which is why performance-sensitive programs such as operating systems and video games are written in mid-level languages such as C and C++.

## Python Syntax

Many programming languages such as C, Javascript and Java use C-style curly braces to open and close functions and classes and parentheses for if statements and loops and semicolons at the end of lines. Python instead uses white space in place of curly braces and forgoes parentheses in if statements and loops. Python also does not have semicolons.

The advantages of Python's minimalistic syntax include improved code readability, less visual clutter and more compact code.

In the following example, the function 'fizzbuzz' is implemented in Python and C. The comparison of these two implementations side by side shows how the Python implementation is cleaner, more compact and more readable than the C implementation.

# Python implementation:

*def* fizzbuzz(*n*):

if n % 5 == 0 and n % 3 == 0:

print("FizzBuzz")

elif n % 3 == 0:

print("Fizz")

elif n % 5 == 0:

print("Buzz")

else:

print(n)

// C implementation:

*void* fizzbuzz(*int* *n*) {

if(*n* % 5 == 0 && *n* % 3 == 0) {

printf("FizzBuzz \n");

} else if(*n* % 3 == 0) {

printf("Fizz \n");

} else if(*n* % 5 == 0) {

printf("Buzz \n");

} else {

printf("%d \n", *n*);

}

}

However, there are some downsides to Python's syntax such as decreased flexibility and errors can result from the wrong amount of whitespace. Programming languages that have semicolons and curly braces allow programmers to write code in any style more easily such as having multi-line statements in their code.

Overall, I prefer Python's syntax because it enforces good coding practices such as whitespace and single line statement while saving lines and characters and making code more readable.

## Python is a General-Purpose Language

Python is a general-purpose programming language meaning that it supports most major programming paradigms including imperative, object-oriented and functional programming (but not logic and mathematical programming). This feature has many practical benefits. It gives the programmer freedom to implement a program in any style depending on which style is perceived to be most appropriate. It also makes Python a useful tool for solving a wide variety of programming challenges. Because of its generality, Python has been called "the swiss army knife of programming languages".

The disadvantage of supporting multiple programming paradigms is that sometimes, a single programming paradigm is the best way to implement a solution. However, when programming with Python, a programmer may use the paradigm that is the easiest or most familiar paradigm or even a complicated combination of paradigms instead of the single best one. More specific programming languages such as Java enforce a single programming paradigm (object-oriented programming in the case of Java) so the programmer never has to be worried about mixing up programming styles.

## Programming Paradigms in Python

Python supports imperative, object-oriented and functional programming meaning that it is possible to use any of these styles separately or mixed together in Python.

### Imperative Programming in Python

Imperative programming is the oldest form of programming because CPUs execute instructions in a step-by-step or imperative way. Naturally then, the first programming languages were imperative. Imperative programs are composed of variables, statements that change them and procedures. Imperative programs involve state changes where variables are declared and then modified iteratively over the course of the program.

Here is an example of an imperative programming in Python:

*def* main():

a = []

for i in range(1, 11):

a.append(i)

for i in range(len(a)):

a[i] = a[i] \* 2

This program is imperative because the list 'a' gradually changes state on each iteration.

The way imperative programming is implemented in Python is not particularly unique. Python supports while loops, for loops, variables and statements like many other programming languages such as C. The only significant factor that is different is Python's syntax.

One of the major advantages of imperative programming is in complexity analysis. Since each step of the program is explicit, the complexity of the program can easily be determined. These qualities are helpful when high-performing code is necessary. As Python is a slow language compared to lower-level languages such as C++, developing efficient algorithms is important.

One of Python's mottoes is "explicit not implicit" meaning that code should avoid hiding important details relevant to executing and debugging it. Since imperative programming is more explicit than functional programming, it logically follows that Python should support imperative programming.

Although Python is a compact and succinct language, writing only imperative code can make code more verbose as loops and variables take up more space than compact function calls. This factor should be taken into account when writing imperative code in Python.

### Functional Programming in Python

Python also supports functional programming. Functional programs are composed of functions which take in an output and produce an output. They are also stateless meaning that variables cannot change once declared. In other words, all variables are constants which have only one state. Since functional programming is part of the declarative family of programming languages, it emphasizes describing *what* programs should achieve over *how* they accomplish it. This more abstract philosophy results in programs that are often very succinct and readable. The fact that variables cannot change state allows the programmer to determine the value of variables more easily. Therefore, functional programs can be easier to reason about debug than classical imperative programs.

The following Python program does the same thing as the imperative code above but is implemented in a functional style. One of the key features of functional programming languages is that they have first-class functions. This means that functions can be treated like any other variable. In the example below, the lambda function 'double' is passed into the 'map' function just like a normal variable.

*def* main():

double = *lambda* *x*: x \* 2

a\_by\_two = *list*(map( double, [x for x in range(1, 11)] ))

This example also demonstrates the declarative nature of functional programming. In this example, we instruct Python to apply the 'double' function to every element. The implementation is declarative because we stated what to do without explicitly saying how to do it (eg. looping through each index). The input, a list, is constant and the output is also a constant which makes the program easier to understand because one only have to think in terms of a single input and an output instead of multiple state changes. It is also clear that the functional implementation of the program above is shorter than the imperative implementation: the functional version only takes up 2 lines of code whereas the imperative version uses 5 lines.

For comparison, here is the same function implemented in Haskell, a purely functional language:

doubleList :: [*Int*]

doubleList = map (\*2) [x | x <- [1..10]]

main = do

print $ doubleList

The Python and Haskell implementations are similar as they both of a similar length, use list comprehensions, the map function and an anonymous function as an argument to the map function. The only significant difference is that Haskell is statically typed whereas Python is dynamically typed.

Python's support for functional programming is more limited than Haskell as unlike Haskell, it does not have functions such as zip, fold, pattern matching or guards. Functional programming in Python is still well-equipped to handle most programming tasks. Also, limited functional programming support is less of an issue in Python as the code can also be written in an imperative or object-oriented style if necessary which is not possible in Haskell.

Python's support of functional programming enables programmers to write short, succinct code that is easier to debug. As Python is a general purpose language, other programming paradigms can also be used in conjunction with functional programming.

### Object-Oriented Programming in Python

The object-oriented model is arguably the best way to organize large software applications where the application is composed of hierarchical components. Fortunately, Python supports object-oriented programming for these kinds of use cases.

Consider the problem of creating a virtual library of books. The problem is hierarchical as a library is made of up many books and each book has properties such as a name and author. A Python solution is easy to create using classes because classes are good at representing hierarchical organizations of information:

*class* Library:

*def* \_\_init\_\_(*self*):

self.books = []

*def* add\_book(*self*, *book*):

self.books.append(book)

*def* \_\_str\_\_(*self*):

s = "Books: \n"

for book in self.books:

s += *str*(book) + "\n"

return s

*class* Book:

*def* \_\_init\_\_(*self*, *name*, *author*):

self.name = name

self.author = author

*def* \_\_str\_\_(*self*):

return self.name + ", " + self.author

*def* main():

library = Library()

book1 = Book("Harry Potter", "J.K. Rowling")

book2 = Book("The Lord of the Rings", "J.R.R Tolkien")

library.add\_book(book1)

library.add\_book(book2)

print(library)

if \_\_name\_\_ == "\_\_main\_\_":

main()

The output of this program is:

***Books:***

***Harry Potter, J.K. Rowling***

***The Lord of the Rings, J.R.R Tolkien***

For the sake of comparison, here is an equivalent implementation written in the popular object-oriented language Java:

import java.util.ArrayList;

class Library {

ArrayList<Book> books;

public Library() {

books = new ArrayList<Book>();

}

public *void* addBook(Book book) {

books.add(book);

}

public String toString() {

String result = "Books: \n";

for(*int* i = 0; i < books.size(); i++) {

result += books.get(i).toString() + "\n";

}

return result;

}

}

class Book {

String title;

String author;

public Book(String title, String author) {

this.title = title;

this.author = author;

}

public String toString() {

return title + ", " + author;

}

}

public class Test {

public static *void* main(String[] args) {

Library library = new Library();

Book book1 = new Book("Harry Potter", "J.K. Rowling");

Book book2 = new Book("The Lord of the Rings", "J.R.R Tolkien");

library.addBook(book1);

library.addBook(book2);

System.out.println(library);

}

}

I found the Python version more straightforward to implement. The Python version uses less lines and characters and does not require any external libraries. In a short object-oriented program, Python's proven productivity and simplicity makes it a good choice. Java is more suitable for large applications because of its speed. It's static typing is also an important factor for improving code maintainability and code readability which is important for large applications. Another detail worth mentioning is the fact that Java is purely object-oriented (apart from primitives) whereas Python often involves a mixture of imperative, functional and object-oriented code. This flexibility can be an asset but may be a problem if code consistency is a priority.

In a simple program such as the one above, Java and Python seem to similar object-oriented features. They both have classes, constructors, inheritance, static methods and operator overloading. However, Java has more advanced object-oriented features which are not available in Python such as interfaces and abstract classes.

Both Python and Java are both well-designed object-oriented programming languages in my opinion. The factors that should be taken into account when choosing between them are whether you want dynamic or static typing, the scale of the project, performance and code readability as well as experience and taste. Many large scale projects have been implemented using object-oriented Python and Java code so either language is a good choice for an object-oriented program.

## Python is Interpreted

High level programming languages are generally either compiled or interpreted. Whereas compiled languages need to be compiled and run, Python can be run immediately because it is interpreted. The fact that Python can be run in one step instead of two increases productivity. There are some downsides to interpreted programming languages however. The main downside of interpreted languages is that their performance is usually worse than compiled languages. There are some exceptions to this rule (eg. NodeJS) but Python runs much slower than a compiled language such as C++ [3].

## Python is Dynamically Typed and Garbage Collected

Python uses dynamic typing (also known as duck typing) to determine the types of variables. It is called dynamic typing because the types of variables are determined at runtime instead of being explicitly stated in the source code. While this feature can increase productivity as the programmer doesn't have to specify types manually (as in Java for example), sometimes the type of function parameters or outputs is not clear to a programmer reading the code. In a statically typed language such as Java, a programmer can easily determine the type because it is explicitly stated which is not true in the case of Python. Because of this problem, Python has introduced "type hinting", the ability to explicitly declare the types of variables [4].

In C and C++, programmers can manually allocate memory for programs as the following example shows:

*void* main() {

*int* \*arr = malloc(10 \* sizeof(*int*));

arr[0] = 0;

free(arr);

}

Python, in contrast, does all memory management automatically. An automatic system called a garbage collector regularly scans the program and deallocates memory for variables that are no longer being used [5]. In C, this process is done manually. Garbage collection is helpful for most programs because it increases memory safety by eliminating the risk of memory leaks - memory that is allocated but not deallocated - while also freeing the programmer from the task of having to allocate and deallocate memory by hand.

However, manual memory management is needed for high performance applications such as 3D video games. The reason why is that garbage collectors carry out garbage collection at unpredictable times. This is a problem in games that require high responsiveness because the process of garbage garbage collection uses CPU cycles that could be used for the game. In particularly demanding games, this process can decrease the performance and responsiveness of the game when it is carried out. The solution is to use manual memory management where variables are only freed from memory when it is appropriate to do so during the game. This is one of the reasons why game engines and other high performance programs are usually written in C++ and not Python.

## Productivity

From my experience, the productivity of Python is high relative to other programming languages such as Java. There are several reasons for this. In general, Python is highly productive because it is highly automated - the computer does a lot of low level work for you. In Python, typing is done dynamically at run time and memory management is automatic because of its built-in garbage collector. The fact that Python is interpreted also provides a major boost to productivity as programs only need to be interpreted once instead of having to be compiled and run as is the case with compiled programming languages. Two other productivity boosters Python has are its syntax and its support for functional programming. As explained above, the syntax of Python is minimalistic, clean and highly readable allowing developers to read and write Python code more quickly. Functional programming support in Python enables programmers to write succinct, declarative code that is easy to understand because of its stateless nature.

Overall, several significant factors - dynamic typing, the interpreter, syntax and functional programming - make Python a highly productive programming language.

## Performance

One of the major downsides of Python is its relatively low performance [3]. This problem has inhibited the spread of Python to high-performance areas of computing such as 3D graphics and operating systems. However, there are ways around these performance limitations. The popular Python AI library Tensorflow demonstrates that it is possible to get high performance from Python programs by using them as wrapper around fast C++ implementations [6].

## Standard Library

The standard library of a programming language has a strong influence on the productivity of programmers because of the concept of re-inventing the wheel: the more functions and classes that need to be implemented by a programmer, the slower the development time will be. Conversely, an extensive standard library enables programmers to quickly use functionality from the standard library instead of implementing it themselves.

For example here is an implementation of the quick sort algorithm in Python [7]:

*def* partition(*arr*, *low*, *high*):

i = (low-1)

pivot = arr[high]

for j in range(low, high):

if arr[j] <= pivot:

i = i + 1

arr[i], arr[j] = arr[j], arr[i]

arr[i + 1], arr[high] = arr[high], arr[i+1]

return (i + 1)

*def* quickSort(*arr*, *low*, *high*):

if len(arr) == 1:

return arr

if low < high:

pi = partition(arr, low, high)

quickSort(arr, low, pi - 1)

quickSort(arr, pi + 1, high)

If there was no Python standard library, a programmer would need to implement a sorting function to sort data. If the algorithm is common, the programmer might be able to copy it from the internet. Even then, more time would be spent implementing the sorting function than simply using the 'sort()' function from the Python standard library. In addition, the sort function would take up extra space. Thus an inadequate standard library leads to a larger code base and lower productivity.

Fortunately, Python has a large and useful standard library. One of Python's mottos is "batteries included" meaning that Python was designed to have a large library containing reusable functions and data structures. For example, instead of implementing a hashmap yourself, you can use Python's dictionary. These built-in tools are highly valuable to programmers because they save them time that would otherwise be needed to implement the tools by hand.

## Conclusion

In conclusion, Python is a well-designed and highly productive general purpose scripting language. It's strengths include simple and readable syntax, support for several programming paradigms, a large standard library and high productivity.

Python's generality is also a minor disadvantage. Because it is so general, the language is not perfectly-suited to any one problem domain. Perhaps the only significant disadvantage of the language is low performance: Python is known to be one of the slowest programming languages which makes it ill-suited for high performance applications such as operating systems and game engines.

While Python is usually not the perfect solution to many types of problems it is a useful general tool programmers can turn to whenever they want to solve a problem quickly and simply.

## References

[1]Wikipedia, *Python (Programming Language)*, December 2020, [Online]. Available: [https://en.wikipedia.org/wiki/rPython\_(progamming\_language](https://en.wikipedia.org/wiki/Python_(programming_language). [Accessed: 10/12/20]

[2] Marco Schweighauser, *Simple 8-bit Assembly Simulator*, August 2015, [Online]. Available: <https://schweigi.github.io/assembler-simulator/>. [Accessed: 10/12/20]

[3] Stephen McAleese. *Insertion Sort Speed Test,* November 2018, [Online]. Available: <https://github.com/smcaleese/insertion-sort-speed-test>. [Accessed: 10/12/20]

[4] Christopher Bailey, *Type Hinting*, October 2019, [Online], Available: <https://realpython.com/lessons/type-hinting/> [Accessed: 10/12/20]

[5] Wikipedia, *Garbage Collection*, December 2020, [Online], Available: <https://en.wikipedia.org/wiki/Garbage_collection_(computer_science>[)](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)), [Accessed 10/12/20]

[6] Renato Candido, *Pure Python vs NumPy vs TensorFlow Performance Comparison,* 2017, [Online], Available: [https://realpython.com/numpy-tensorflow-performance/#conclusion](https://realpython.com/numpy-tensorflow-performance/" \l "conclusion), [Accessed: 10/12/20]

[7] Anush Krishna, *Python Program for QuickSort,* August 2020, [Online], Available: <https://www.geeksforgeeks.org/python-program-for-quicksort/>