AN ALGORITHM-ADAPTIVE SOURCE CODE CONVERTER TO AUTOMATE THE TRANSLATION FROM PYTHON TO JAVA

Eric Jin[1] and Yu Sun[2]

[1]Northwood High School, 4515 Portola Parkway, Irvine, CA 92620, USA
[2]California State Polytechnic University, Pomona, CA, 91768, USA

ABSTRACT

In the fields of computer science, there exist hundreds of different programming languages. They often have different usage and strength but also have a huge number of overlapping abilities [1]. Especially the kind of general-purpose coding language that is widely used by people, for example Java, Python and C++ [2]. However, there is a lack of comprehensive methods for the conversion for codes from one language to another [3], making the task of converting a program in between multiple coding languages hard and inconvenient. This paper thoroughly explained how my team designs a tool that converts Python source code into Java which has the exact same function and features. We applied this converter, or transpiler, to many Python codes, and successfully turned them into Java codes. Two qualitative experiments were conducted to test the effectiveness of the converter. 1. Converting Python solutions of 5 United States Computer Science Olympic (USACO) problems into Java solutions and conducting a qualitative evaluation of the correctness of the produced solution; 2. converting codes of various lengths from 10 different users to test the adaptability of this converter with randomized input. The results show that this converter is capable of an error rate less than 10% out of the entire code, and the translated code can perform the exact same function as the original code.

KEYWORDS

Algorithm, programing language translation, Python, Java.

1. INTRODUCTION

There are nearly as many programming languages in this world as human languages, but the conversion between these languages of computers is still a developing field [1]. There are comprehensive translations between all human languages, but there aren't many for programming languages [3]. The solution to this problem is hidden in coding itself, algorithms can be built to convert code between coding languages. This paper is about an algorithm we built that does this task: an algorithm that can translate Python code into Java code. Why choose Python and Java? Because they are among the list of the most used programming languages in the current world [4].

The converter is able to take in a file containing Python source codes and convert it to Java source codes that have the same performance. However, perfection in translation is impossible to achieve due to some of the fundamental differences between the two languages [5]. The converter we built is able to achieve more than 90% of correct translation. This program is useful in many aspects. First it will be a helpful tool to beginners learning these languages, it is convenient with a tool where one can just type in a line of code one already knows and receive the exact same code

David C. Wyld et al. (Eds): COMIT, CRBL, BIOM, WiMNeT, SIP, AISO - 2021
pp. 215-229, 2021. CS & IT - CSCP 2021 DOI: 10.5121/csit.2021.111719

in the language one is learning. Especially in situations when multiple coding languages of the same code might be needed. For example, the United States of America Computer Science Olympics (USACO), a national competition open to all high school students, often have problems that are only doable with certain languages. It saves time and works to avoid code the same program again in another language. Also, actual programming projects, like building an application, might need to have different versions in different languages to meet the needs of the users of different platforms [6].

There has been existing algorithms aiming to transpile one programing language to another [7]. Google's Google Web Toolkit (GWT) turns Java to JavaScript, Facebook's hiphop compiler compiles PHP into C [8]. What our converter is doing is parallel to the transpilers of the two great tech giants: turning one programming language to another on the source code level. Our converter is unique since it is doing transpilation between Java and Python, which is different from the other existing transpilers. However, google and Facebook's transpiler optimize the original source code during the transpilation, this is something that our algorithm is not able to do [8].

Just within the field of transpiling Java and Python, there also exists a method called Jython [9]. It is a plugin of Java which allows users to "freely mix the two languages both during development and in shipping products." according to their website. While looking similar to our converter, this is not the same thing as transpiling. With Jython developers can switch part of a Java code to Python code, having the same ability. but it does not have the functionality of turning one source code to another source code, which is the main purpose of our algorithm.

The approach we took to solve the problem of converting one coding language to another is a method similar to the enumeration method [10]. Which means, using if statements to list out and detect every possible structure that exists in the Python code, and convert each part of the code into its Java version. First of all, after a Python file is inputted, the converter breaks it line by line, for each line it quickly converts the simple parts of the spaces for indentation and comments, so only the meaningful code is carried into enumeration. In a certain order, the algorithm checks for unique words which represent a list of commonly used Python code structures, for example a line containing an isolated "=" sign is dealing with a variable: the right side value needs to be stored into the left side. The sutures are the following: "defining a function/methods", "calling a function", "using a list to store values", "interactions like for loop and while loop", "if statements and booleans", "casting variables to another type", and finally "creating or updating a variable". The order is needed since multiple structures can occur in the same line, for example an if statement which checks some value in a list using a method. Plus all kinds of edge cases not included in the common used code structure listed above, like "open and reading files" "for each loop" "the in function".Each of these code structures were taken out and convert into Java codes, while the other parts of the Python line remain the same. After the Python line comes out of all these checking cases, almost every single part will be converted into Java code, thus it is written to the Java file as output.

This algorithm is focusing on the transpilation of Python source code to Java should contain the following abilities: the width of convertible Python code, the correctness of the output Java code, and the accommodation to any user with different coding habits. We designed two experiments to measure these activities. For Experiment 1, First, we decided to convert solutions written in Python for five United States of American Computer Science Olympic (USACO) algorithm problems [13]. We will measure how much of the converted code, which is in Java, is needed to fix before it can run properly, since the transpilation cannot be perfect. Then we check if the converted Java code is able to output the correct results to the problem just as their Python version, in order to prove the relation actually works.

For the second experiment, we gathered raw Python source code from 10 random coders that have length from short to long. Similar analysis from experiment 1 is applied: the percentage of error is calculated by the amount of incorrectly translated characters out of the total length. Also, a graph of length vs error is plotted to show if there's any correlation between the length of the code and the effectiveness of the converter.

The rest of the paper is organized as follows: Section 2 gives the details on the challenges that we met during the construction of the converter. Section 3 takes an overall go through of all the codes within thealgorithms, and some close look at the details of the code. Section 4 presents the structure of the two experiments conducted along with analysis of the data generated, followed by the introduction of related similar works in Section 5. Finally, Section 6 gives the conclusion remarks, as well as pointing out the future work of this project.

2. CHALLENGES

In order to build the tracking system, a few challenges have been identified as follows.

2.1. Arrays

One fundamental difference between the language of Java and Python is that Python has a data structure called list, which can store any kind of elements with any length in one single list structure [11]. In simple words, the type and length of the list is modifiable. However, this feature brings convenience at the cost of using extra memory spaces and slowing the speed of the code. Meanwhile, Java has two similar structures: array and ArrayLists. A Java array has the similar syntax as a Python list (e.g., list[0]) is a code that gets the first element in both a Python list and a Java array. However, the array must be declared with a fixed length and a fixed type, which does not fit with the flexible features of the Python list. On the other hand, the ArrayList needs a fixed type for the element it contains, while having the modifiable length of the list, but its syntax is very different. In the end, after successfully handling the syntax, We had chosen to use ArrayList in the output Java code to replace every list created in the input Python code. So, a (list = [ ]) will be converted into (e.g., ArrayList list = new ArrayList<>();). To solve the fixed data type, we simply declared them all as Objects, which is the fundamental data type in Java. However, this triggers a more complicated issue: casting the elements saved as objects in the ArrayLists to their designated data type whenever they are used.

2.2. Variable

In Python, you can declare a variable as an integer, but later on change its value to a string, the Python syntax generally disregards data types and will not generate a compiling error for any non matching data type mistakes. However, Java syntax is strict with data types. Once a new variable is declared, it requires a fixed data type. Thus, there's no useful information for the type of the variable in the input Python line except the value of the variable itself. The algorithm keeps track of every variable created, both the variable name and its value, by detecting the lines of codes that contain an isolated single "=". If the variable name is not the record, it launches a series of complicated case work to categorize the value of the variable and grants the corresponding Java variables their proper type. This also includes the special case of initializing an ArrayList. With this method, the algorithm is able to cast the elements with type objects in the ArrayListonce they need to be accessed.

2.3. Functions

The ultimate challenge is that, no matter how comprehensive the converter is, it can never cover the entirety of Python to Java conversion. Because each language has their own libraries and functions, the number is countles and increasing everyday. Our converter did not solve this challenge but offers a fairly clean fix to the problem, which is to find the matching pair of functions. For example, both languages contain a math library and can call functions to do mathematical operations. Python's pow(a,b) is the same thing as Java's Math.pow(a,b). Therefore, our converter contains a dictionary of corresponding functions. Once a typical structure of calling functions in the input Python code is identified, the dictionary returns the corresponding Java function. Now it only contains the common functions of Java and Python, but it is easy to add to the dictionary if a new matching pair function is needed.

3. SOLUTION

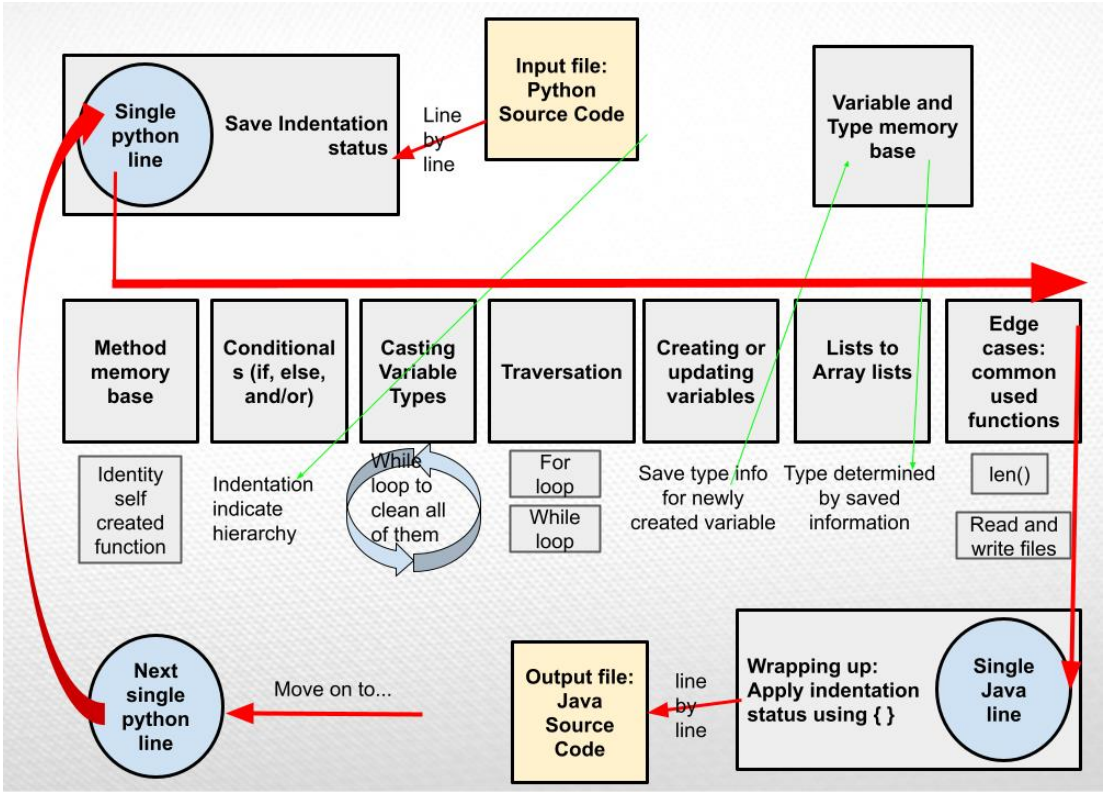

Figure 1. Overview of the solution

The converter is an algorithm coded in Python that converts Python source code to Java source code with the same function [3]. It does it in a line-by-line process, in a way similar to the enumeration method. Usually each line is analyzed separately, meaning the algorithms treat each line as converting the same thing as if it is the only imputed Python line. Focusing on the actual process, it uses string parsing to identify certain structures in the code and convert it to Java code. The structure it is looking for are the following in this order: calling or creating methods/functions; conditional; casting variables; for loop and while loop; dealing with variables; Python list; and other edge cases of commonly used functions. Besides the line-by-line process, there is certain information that is meaningful to the entirety of the code, such as variables and methods used throughout code. The algorithm observes variables and functions when they are

first created and saves them with their parameters and type for later use. Reading one Python line, writing one Java line after all Python code is converted, the Java code is outputted as a Java file.

3.2. Handling with indentation:

Example:

```
list = []                                   ArrayList list = new ArrayList<>();
#    a for loop adding value to list   →    //    a for loop adding value to list
for i in range(5):                          for(int i = 0; i < 5; i+=1 ) {
list.append(i)                          list.add(i);
print(list)                                 }
                                          System.out.println(list);
```

```
for i in range(length):
  line = f.readline()
  javaLine = ""


  space = 0   #Indentation
  try:
    char = line[space]
  except:
    continue
  while char == " ":
    space+=1
    char = line[space]


  line = line.strip()

  if len(line) < 1:  #empty line
    java.write("\n")
    continue
  if line[0] == "#": #comment line
    javaLine = "//"+ line[1:]
    print(javaLine)
    java.write(javaLine +"\n")
    print("")
    continue
  while "#" in line:
    line = line[:line.index("#")] +"; //"+ line[line.index("#")+1:]

  spaces.append(space)
  try:
    if spaces[-1] <spaces[-2]:
      # for j in range(spaces[-2]):
      #   java.write(" ")
      java.write("}"+"\n")
  except:
    print()
  print(space,"-", line)

  for j in range(space):
    java.write(" ")
```

Figure 2. Code for handling with indentation

The body of this algorithm is a for loop, which reads the file containing the Python source code line by line. the “JavaLine” variable is created so save the Java version of the currentline. First the situation of an empty line is detected, so the code can go on counting how many spaces are in front of the first character in this line. This number indicating the indentation is recorded in a list, then all the spaces are stripped off the line so only the meaningful code is passed on to the following analysis. This section of the code focuses on the important information revealed by indentation. Since the Python indentation determines the hierarchy of the code, it needs to be converted into pairs of “{ “ and “}”, the corresponding structure in Java. The opening “{" will be taken care of later in the algorithm when a “:” is see at the end of the code, so the current section only needs to add closing “}” when needed. To do this, the algorithm traverses through the list of spaces, whenever a line has less space than its previous one, it means the closure of a prior level. Thus, a closing “}” will be added to the javelin. However, certain indentation structures cannot be identified by this method, so sometimes the user needs to manually fix the “{“ and “}” in the Java code.

3.2.1. If statements and logical operations:

Example:

```
if a == 2 and (True or False):                  if( a == 2 && (true || false)) {
        print("case 1")              →                  System.out.println("case 1");
                                                    }
else:                                           else {
        print("case 2")                                 System.out.println("case 2");
                                                    }
```

```
#-------------------------------------------------------------
  if "if" == line[0:2]:    #if statements
    line = "if("+line[2:line.index(":")]+") {"
  if "else" == line[0:4]:
    line = "else {"
   #elif  : to else if()
  if "elif" == line[0:4]:
    line = "else if("+line[4:line.index(":")]+") {"
  while "True" in line:
    line = line[:line.index("True")]+"true"+line[line.index("True")+4:]
  while "False" in line:
    line = line[:line.index("False")]+"false"+line[line.index("False")+5:]
  while "and" in line:
    line = line[:line.index("and")]+"&&"+line[line.index("and")+3:]
  while " or " in line:
    line = line[:line.index(" or ")]+" || "+line[line.index(" or ")+4:]
```

Figure 3. Code for statements and logical operations

How the algorithm handles if statements work like this: picks out the part of the code where Java and Python is different and turns it into the Java version. This specific segment is simple, which makes it a typical example. First, if “if” , “else” or “else if” is seen in the front of the line, it immediately makes this line part of an if-else structure with no exception. Therefore, the only thing needed to do is to do some small editing of syntax, adding parentheses and braces. Another thing different between Python and Java is how they write “true” “false” “and” “else”, while loops are needed to find all of them since multiple can exist in the same line.

3.2.2. Functions:

Example:

list1.append([ ]) → ArrayListnewArray = new ArrayList<>(); list1.add(newArray);
list2.pop(0) → list2.remove(0)

```
if "." in line:   #functions: caller.function(parameter)

  if not ('"' in line and line.index("(") < line.index('"')):
    #check if ("blah.blah")

    try:
      a = int(line[line.index(".")-1])
    except:

      front = line[:line.index(".")+1]
      line = line[line.index(".")+1 :]
      function = line[: line.index("(")]
      para = line[line.index("(")+1 : line.index(")")]
      back = line[line.index(")")+1:]
      # print("func. "+function, para)
      if para == "[]" or para ==  "[ ]":
        para = "newArray"
        java.write("ArrayList newArray = new ArrayList<>();")
      line = front+str(func(function, para))+"("+para+")"+back
```

Figure 4. Code for functions

The first line of code detects if a dot"." is presented in the line variable, which is a string containing the Python line. The second line checks if the dot exists as part of a string by checking if the quotation mark is also in the line. However, even if it is not in a string, another unwanted situation is when a dot is used as a decimal point between numbers like "1.25", the try conditional excludes it by checking if the character before the dot is a number. If those are not the case, the algorithms got what it is looking for: calling functions. Normally a dot in the Python syntax has the usage of calling a function from an imported dependency, which usually has the structure of "caller.function(parameter)" . The algorithm uses some string slicing to separate each part, among these parts the function is replaced by the Java version of the function by calling the func() dictionary containing this info, which will be explained in delta below. The parameter is also important, when the parameter is a list, Java needs a new ArrayList to be created, which is what the last chunk of the code is doing. After getting all the parts in Java, the algorithm reassembles them into one Java line calling a Java function.

```
def func(function, para):
  javafunc["append"] = "add"
  javafunc["readline"] = "readLine"
  javafunc["read"] = "readLine"
  javafunc["write"] = "println"
  javafunc["close"] = "close"
  javafunc["remove"] = "remove"
  javafunc["index"] = "indexOf"
  javafunc["pop"] = "remove"

  return javafunc[function]
```

Figure 5. Part code for the function memory base

This is part of the code for the function memory base, which is used for saving the Python and Java functions that are interchangeable. It can be updated to include more pair as times goes on, depending on the dependencies used in the Python code.

3.2.3. Casting variable types:

Example:

int(number1) → (int) number1;
float(number2) → (double) number2;
str(x) → String.valueOf(x);
float (str (int (a))) → (double) String.valueOf((int) a) ;

```
while "int(" in line:  #casting int() float() str() bool()
  exist = False   #(int) (double) String.valueOf()
  parentheses = 0
  inside = ""
  for i in range(3,len(line)):
    if not exist and line[i] == "(":
      if not exist and line[i-len("int"): i] == "int":
        front = line[:i-len("int")]
        exist = True
        parentheses=1
        continue
    if exist:
      if line[i] == "(":
        parentheses+=1
      if line[i] == ")":
        parentheses-=1
      if parentheses == 0:  #found its parter ()
        line = front+"(int)"+inside + line[i+1:]
        break
      inside+=line[i]
```

Figure 6. Code for casting variable types

The code section above it for converting the casting of integers which means turning int(a) into (int) a, for other variable types (string, boolean float) the code is nearly the exact same as this one so it is enough to look closely at just this one for integer. Recognizing the casting of variables is different from other structures since it may appear many times in one single Python line, making the while loop a fit tool to find and change all of them. This also means the algorithm would need to identify the pair of parentheses belonging to this specific cast action. After identifying any “int(“ still existing in the current line, the algorithm starts to count the parenthesis after it. Only if the counter (variable parenthesis) notices that the number of open parenthesis and close parenthesis matches, it is certain that which part of the code is the cast actually casting. Thus, the same process of breaking up the line and resembling it into Java syntax is applied, and this specific cast has been successfully converted. However, the algorithm will not move on until all the casting in the current line is finished, making it capable of handling nested casting operations.

3.2.4. For loop

Example:

for i in range(10): → for(int i = 0; i < 10; i+=1) {
for a in range(0,50,2): → for(int a = 0; a < 50; a+=2) {
for b in range(n): → for(int b = 0; b < n; b+=1) {
for value in elements: → for(Object value : elements) {

```
if line[0:4] == "for ": #for loop
  # print(line)
  parts = line.split()
  # print(parts)
  var = parts[1]
  if "range(" in parts[3]: #for i in range(start,end,step)
    ranges = ""
    start = 0 #defualt value
    try:
      end = int(parts[3][parts[3].index("(")+1:parts[3].index(")")])
    except:
      end = parts[3][parts[3].index("(")+1:parts[3].index(")")]
    for i in range(3,len(parts)):
      ranges+=parts[i]
    # print(ranges)
    step = 1
    if ranges.count(",") == 1:
      start = (ranges[ranges.index("(")+1:ranges.index(",")])
      end = (ranges[ranges.index(",")+1:ranges.index(")")])
    if ranges.count(",") == 2:
      start = (ranges[ranges.index("(")+1:ranges.index(",")])
      ranges = ranges[ranges.index(",")+1:]
      end = (ranges[:ranges.index(",")])
      step = (ranges[ranges.index(",")+1:ranges.index(")")])
    # print(var, start, end)
    javaLine = "for(int "+var+" = "+str(start)+"; "+var+" < "+str(end)+";
     "+var+"+="+str(step)+" ) {"

  else:#for i in List:
    list = parts[3][ : len(parts[3])-1]
    print(list)
    javaLine = "for(Object "+var+" : "+list+") {"
```

Figure 7. Code for loop

If the Python line starts with “for” it is definitely a for loop (or a for each loop). A Python for loop consists of up to 3 parts: start, end, step, in the form of “for i in range (start, end, step). The Java version will be for (int i = start; i< end; i+= step). However, usually the only parameter used by programmers is only the end, to repeat the loop for x times. So the algorithm recognizes the different situations by splitting the Python line and counting how many commas within the range () string. if either step or start is not used, they will be set to the default value 1 and 0. Then, all the parts including the variable (var), start, end and step are resembled back in Java syntax. However, the string slicing has the shortcomes of being unable to deal with extra spaces in the Python line, so it is dependent on the assumption of the syntax of the imputed Python code.

Similarly, a for each loop is converted by grabbing the single element and the group of elements from the Python line and putting it in Java format of “for (Object element: elements) {“. notice that the type is settled to be Object since it cannot be determined from the original Python code.

3.2.5. While loop

while a== 2: → while (a == 2) {

```
if line[0:6] == "while ":  #while loop
    condition = line[6:len(line)-1].strip()
    javaLine = "while ( "+condition+" ) {"
    print(javaLine)
    java.write(javaLine +"\n")
    print("")
    continue
```

Figure 8. Code for while loop

On the other hand, the while loop is much simpler than the for loop. Just detecting the line starting with “while” will eliminate other possibilities. The process of conversion is extracting the boolean statement and putting it in parentheses.

3.2.6. Variables

Example:

a = 2		int a = 2;
a = a * 3		a = a * 3;
name = “Eric”	→	String name = “Eric”;
name = name + “Jin”		name = name + “Jin”;
list = []		ArrayList list = new ArrayList<>();

```
if "=" in line:  #assign statements
    equal = line.index("=")
    if line[equal-1] in ["+","-","*","/",">","<","!"]:
        javaLine = line+";"
        print(javaLine)
        java.write(javaLine +"\n")
        print("")
        continue
    if line[equal-1] in [">","<","!"]:
        javaLine = line
        print(javaLine)
        java.write(javaLine +"\n")
        print("")
        continue
    if line[equal+1] in ["="]:
        javaLine = line
        print(javaLine)
        java.write(javaLine +"\n")
        print("")
        continue
    # print(line)

    varName = line[:equal].strip()
    value = line[equal+1:].strip()
    existed = False
    if varName in variables and "." not in varName:
        existed = True
    else:
        variables.append(varName)
    type = ""
    # print(varName+":-----"+value)
```

Figure 9. Code for variables (1)

```
if ".size" in value:
  if not existed:
    type = "int"
if value[0] == '"':
  if not existed:
    type = "String"
if "(int)" in value:
  if not existed:
    type = "int"
try:   #integer
  intValue = int(value)
  if not existed:
    type = "int"
except:
  pass
if value == "[]":  #list
  if not existed:
    type = "ArrayList"
    value = "new ArrayList<>()"

#finished
if not existed:
  varType[varName] = type

javaLine = type +" "+ varName+" = "+value+";"
print(javaLine)
java.write(javaLine +"\n")
print("")
continue
```

Figure 10. Code for variables (2)

The algorithm identifies equal signs "=" in the current line, if the equal sign is preceded with a mathematical operation sign, that means the Java version is the same as the Python version. So the line simply gets written to the Java file, since at this point of the algorithm other Python parts in the current line are already converted. However, the case of an isolated equal sign is where a variable is created, or the value of a variable is changed. Here a major difference between Python and Java occurs: Java requires a fixed type to be assigned to variables that are first created. The algorithm counters this issue by saving created variables to a dictionary(varName : type), so excited variables can be extinguished from newly created variables. A line involving creating a variable has two components, the variable name on the left side, and whatever the value is on the right side. By detecting certain features for the value, like quotation marks for string and "[]" for ArrayList, in most cases a fixed type for the can be determined. Then this information is saved to the dictionary mentioned earlier. However, due to the major difference that Python treats everything as an object, in some case the variable type cannot be determined by just looking at the Python code, thus the user needs to manually provide this information.

4. EXPERIMENT

Experiment 1: Correctness

The convertor takes in Python code and converts it line by line to Java code. In order to determine the correctness of the translated code, we designed an experiment using Python code samples from United States of America Computer Science Olympic (USACO) questions. The USACO is a competition consisting of coding problems of different difficulties about algorithms,

and different numbers of test cases required to pass for each question [13]. We first randomly selected 5 problems from the USACO, 2 bronze level and 3 silver level, and wrote Python solutions to each of them which successfully passed all the cases. Then, put the 8 Python code into the converter one by one, to generate 8 Javatranslations of the code. Therefore, since the converter cannot produce 100% perfect Java codes, we are able to count the number of characters that needed to be manfully fixed in the Java code before the code executed with no trouble. This number indicates the correctness of the translated code. Finally, the ability of the Java code is supported by observing if it can pass the same problem just like the original Python code does.
Premise: the 8 Python codes are able to pass the 5 bronze problem and 3 silver problems.

Table 1. Data of correctness

correctness (changed characters / total characters)	proof of functionality
79 / 917 = 8.6 %	pass (bronze)
28 / 883 = 3.2%	pass (bronze)
132 / 1131 = 11.7%	pass (silver)
75 / 725 = 10.3%	pass (silver)
158 / 1263 = 12.5%	pass (silver)
average: 9.26%	

The percentage on the left indicated the ratio of characters was manually fixed in the converted code due to unavoidable syntax and logical error, until the code executed without mistake. This is the measurement of the correctness of the translation process. The average percentage of error is 9.26%.

After fixing the code, the experiment demonstrates it works just as fine as the original Python code by seeing if the Java code is able to pass for each USACO problem. Fortunately, the result is indicating that after fixing a certain numberof mistakes. The converted code has the exact abilities and functions as the original code. Overall, the average percent of characters needed to be fixed in the translated code is 9.26%. The conclusion can be drawn from this experiment is: if the Python code is in the field of algorithms building, after fixing about 9% of the converted Java code, it will have the exact same function as the original Python code.

Experiment 2: Adaptability

Experiment one uses Python code inputs from only one coder, who is the builder of this convertor. However, different coders have different coding habits, which could influence the function of the convertor. To test the effectiveness of this convertor on different coders, we gathered 10 Python code samples from 10 different coders, ranging from 10 lines to 200 lines, and used the converter to transpile it into Java code. Analysis done on the 10 outputs is shown below:

Table 2. Data of Adaptability

total characters	117	225	102	32	605	289	473	632	1154	1141
changed characters	9	13	6	2	51	19	23	16	45	73
percentage of changed	7.7%	5.8%	5.9%	6.3%	8.4%	6.5%	4.8%	2.5%	3.9%	5.1%

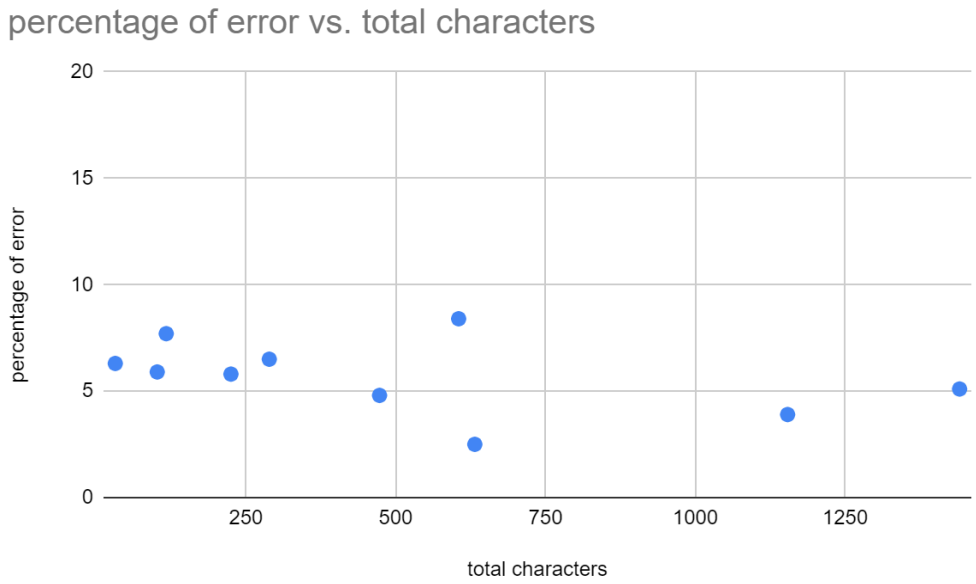

Figure 11. Percentage vs. Characters

The 10 translated Java codes each have length from 60 characters to more than 1000 characters. Using the same calculation method as experiment 1: the number of characters that did not get converted successfully out of the total characters = the percentage of error. As can be seen, for different coders, the percentage of error of translation fluctuates between 2% - 9%. Thus, we can say that the converter has good adaptability to different coders. However, more question remains: could the length of the code be an issue? would the converter be less effective if the code is longer? To answer this, a graph of percentage of error vs total characters is constructed showing there's no obvious correlation between the length of the code and the correctness of translation. Thus, the conclusion can be made is this: the converter has a 2% -9% error rate in the translation of Python to Java code, depending on the coder and length of the input code.

The goal of the two above experiments is to prove the trustability of the converter in terms of measuring its correctness of translation and adaptability to various kinds of inputs. Experiment 1 shows that the complicated algorithmic Python codes from USACO have only 9.2% of error rate, and the converted code can perform the extent function as the original code. Experiment 2 is testing how well the converter performs under different situations. Observation supporters that the error rate is 2% - 9%, and the converter works effectively with different coders and various input lengths. Although the sample size used in the two experiments cannot be judged as large

data, overall, the error rate never exceeds 10% of the entire code piece. Thus, the converter is trustable with a translation rate more than 90% under any situation.

5. RELATED WORK

Lachaux et al. presented a programming code translator using machine learning [14]. They selected a number of source code GitHub repositories and trained a machine learning model to automatically translate the code from one language to another. Although machine learning allows a more automated approach to perform the translation, it cannot capture all the possibilities or rules, and there is always an accuracy issue. Our work is totally based on the rules, so we have a more reliable foundation to guarantee the accuracy. In addition, our work specifically focuses on Java to Python translation in order to improve the accuracy, while their work targets a more generalized language transition.

Abazyan et al. demonstrated their version of an interlanguage translation for Python and Java, it is an algorithm that translates from source to source, and it uses machine learning to correct the translation whenever the accuracy is lower than 60% [15]. Their algorithm is able to conduct back translating, which means it can perform both Python to Java and Java to Python translation. Although their method does have a more general usage, the reliance of machine learning to fix low accuracy translations shows the instability of translation. Our work focuses on only handwriting algorithms and involves no AI, which generates a more stable accuracy among the translation from Python to Java.

Aggarwal et al. had constructed a tool to convert Python 2 codes into the higher version Python 3, it uses statistical machine translation, which is a technique used in natural language translation [16]. Their score for evaluating the accuracy is as high as 99.37. The conversion of code from a previous version to a newer version of the same coding language looks similar to cross language translation, but they are fundamentally different. While our converter is doing the task of Python to Java translation, similar challenges exist in the analysis of Python source code are also mentioned in Aggarwal‘s article. However our work can not achieve that high of an accuracy due to the other challenges in cross language translation.

6. CONCLUSIONS

In this paper, we presented the converter we built for the task of cross language translation among programming languages, more specifically, from the wide-use language Python to Java. Our converter takes in Python source code and outputs Java source code as close as the original code. Our converter is an algorithm handcrafted in Python, the philosophy it uses is enumeration, which means testing out every possible structure contained in a Python code. The translation process is done line by line, with the algorithm detecting certain structures in the Pythoncode and turning each piece into its corresponding Java version. The accuracy of our converter is about 90%, without considering the involvement of Python dependency in the input. This is supported by the two experiments we had conducted. The first one tested the converter on 5 USACO programming problems, the Java translation has about 9% error compared to the original Python solutions. The second experiment tested the converter with code inputs from 10 different users, as well as varying length. The accuracy fluctuates around 92% to 98%.

Accuracy: all though that full accuracy in the translation from Python to Javacannot be achieve with the enumeration method that we had implemented, the current accuracy at roughly 92% is definitely not the upper limitation

Accessibility to the public: the goal of building this converter is never about keeping it to ourselves, instead it should be a tool benefiting the public, in the current moment it is still a piece of code. We will work on how to make it accessible to everyone.

Error detection system: since the converter will encounter parts of the code which it cannot convert into Java, it will be reasonable for it to have a feature of labeling those parts in the code to notify the users.

Our team will continue to work on modifying our algorithms to improve the accuracies of the converter and add the error detection function to it. Thus, it will be ready as a convenient tool open to the public. We will publish it as a web application, anyone can go to the domain name and use our tool for their own purpose.

REFERENCES

[1] Pierce, Benjamin C., and C. Benjamin. Types and programming languages. MIT press, 2002.
[2] Dijkstra, E. W., and EdsgerWybe Dijkstra. "Programming languages." Co-operating Sequential Processes. Academic Press, 1968. 43-112.
[3] Qiu, Lili. Programming language translation. Cornell University, 1999.
[4] Cass, Stephen. "The 2015 top ten programming languages." IEEE Spectrum, July 20 (2015).
[5] Lo, Chieh-An, Yu-Tzu Lin, and Cheng-Chih Wu. "Which programming language should students learn first? A comparison of Java and Python." 2015 International Conference on Learning and Teaching in Computing and Engineering. IEEE, 2015.
[6] Floyd, Robert W. "The syntax of programming languages-a survey." IEEE Transactions on Electronic Computers 4 (1964): 346-353.
[7] Chaganti, Prabhakar. Google Web Toolkit. Packt Publishing, 2007.
[8] Zhao, Haiping, et al. "The HipHop compiler for PHP." ACM SIGPLAN Notices 47.10 (2012): 575-586.
[9] Juneau, Josh, et al. The definitive guide to Jython: Python for the Java platform. Apress, 2010.
[10] Beckwith, Mary, and Frank Restle. "Process of enumeration." Psychological Review 73.5 (1966): 437.
[11] Arnold, Ken, James Gosling, and David Holmes. The Java programming language. Addison Wesley Professional, 2005.
[12] Lutz, Mark. Programming Python. " O'Reilly Media, Inc.", 2001.
[13] USACO, www.usaco.org/.
[14] Lachaux, Marie-Anne, et al. "Unsupervised translation of programming languages." arXiv preprint arXiv:2006.03511 (2020).
[15] Abazyan, Suren, Narek Mamikonyan, and Vakhtang Janpoladov. "Interlanguage Translation Utility with Integrated Machine Learning Algorithms." Open Access Library Journal 7.5 (2020): 1-5.
[16] Aggarwal, Karan, Mohammad Salameh, and Abram Hindle. Using machine translation for converting Python 2 to Python 3 code. No. e1817. PeerJPrePrints, 2015.