The Art of Self-Replication

This journal is separated into three sections, so the reader can easily skim and acquire the information that is of interest. The first section provides an overview on quines, the second section goes further into depth on the actual implementation of quines, whereas the third section provides possible applications on quines. All the code implementations are done in Python since it more translateable as a pseudocode.

Section I:

Quine: A quine is a program that doesn't require any input and copy its source code as an output. This is a key concept that will allow self-replication and self-manipulation without the authorization of a programmer, in which I will go into further (Section III).

You might ask… Why Quines? Why not just use a copy/paste command? Well here are multiple reasonings why quines are a preferred alternative. The benefit is (self) copying a program is O.S. dependent, since (ex.) a Linux and Windows will have different methods of implementing a copy. As such, one would need to implement multiple O.S. methods to ensure a proper copy. On the other hand, a Quine will consistently copy itself without any O.S. related problem, reducing clutter.

What are the benefits as to writing Quines? Quines allow your programs to be independent, with the ability to spread itself without any handling from the creator. This means you can essentially let your program run free and can be certain it will execute properly. Likewise, it is far more adaptable and can easily mutate itself.

Furthermore, quines are an interesting topic since can be represented as another piece of the infinity stone of the infinity gauntlet to self-autonomy. Artificial Intelligence has a similar goal of autonomy, but that mostly pertains to the automating machines, analyses, etc. Unlike Artificial Intelligence, we have not yet put substantial effort into self-maintaining programs. Quines can be a piece of the solution as it gives programs the opportunity to self-replicate and self-maintenance.

Section II:

One-Liner Quines

As a simple example, this code serves as proof of concept and accurately presents the gist of quines. However, it would be a deception to say that one-liner quines are simple, as they could be incredibly difficult to comprehend and discern a line of code and string.

This is an example taken from Wikipedia:

```
s = 's = %r\nprint(s%%s)'; print(s%s)
```

This is a simple program that will print itself, line for line. Notice that this simple quine is set up through utilizing a string which represents the entire program itself, which then has an execution function (print in this case) following it.

Generic Simple Quine Method

Writing a quine may seem simple at first, until you realize all the paradoxical tricks that comes along with it. After attempting to write your first quine, you will quickly notice that tiny details within the program can determine the difference between a failed quine and one that works. Fortunately, after writing a couple, you can easily gain insight of the traps and faults of a poor quine.

This example program simply prints “Hello World”, while writing its own code onto the console

Python Code:

```
def main():
    str = "Hello World!"
    quine = [
      "def main():",                # Line 1
      "    str = \"Hello World!\"", # 2
      "    quine = [",              # 3
                    # The Quine String itself will go in between
      "    ]",
      "    for init in quine[0:3]",
       "        print(init)",
       "    for mid in quine:",
       "        print(\"     \\\"\", end='')",
       "        for ch in mid:",
       "            if ch in [\"\\\\\", \"\\\"\"]:",
       "                print(\"\\\\\", end='')",
       "            print(ch, end='')",
       "        print(\"\\\",\", end='\\n')",
       "    for final in quine[3:]:",
       "        print(final)",
       "    print(str)",
     ]
     for init in quine[0:3]
         print(init)
     for mid in quine:
         print("     \"", end='')
         for ch in mid:
             if ch in ["\\", "\""]:
                 print("\\", end='')
             print(ch, end='')
         print("\",", end='\n')
     for final in quine[3:]:
         print(final)
     print(str)

main()

```

Output:

```
1. def main():
2.     str = "Hello World!"
3.     quine = [
4.       "def main():",
5.       "    str = \"Hello World!\"",
6.       "    quine = [",
7.       "    ]",
8.       "    for init in quine[0:3]",
9.       "        print(init)",
10.        "    for mid in quine:",
11.        "        print(\"     \\\"\", end='')",
12.        "        for ch in mid:",
13.        "            if ch in [\"\\\\\", \"\\\"\"]:",
14.        "                print(\"\\\\\", end='')",
15.        "            print(ch, end='')",
16.        "        print(\"\\\",\", end='\\n')",
17.        "    for final in quine[3:]:",
18.        "        print(final)",
19.        "    print(str)",
20.    ]
21.    for init in quine[0:3]
22.        print(init)
23.    for mid in quine:
24.        print("     \"", end='')
25.        for ch in mid:
26.            if ch in ["\\", "\""]:
27.                print("\\", end='')
28.            print(ch, end='')
29.        print("\",", end='\n')
30.    for final in quine[3:]:
31.        print(final)
32.    print(str)
33. Hello World!
34.
```

It would seem as if I had copy and pasted my code into the output, although I didn't. In this case, it is count as successful reproduction, just without the extra comment guidance

Ok… This Does look a little complex, just for a "Hello World!" program, but once you delve into the code, it really isn't.

Quine Breakdown:

A generic quine can easily be broken down into four main parts

1) Payload: Every program has a purpose and this program is simply to print “Hello World!”. As such, it is simply utilized to write the set-up code, a string in this case
2) Self String: The program will need to write/print itself elsewhere, so you will need to create a string that encompasses the function that calls it. In this case, the string is called quine. Through observation, you will notice it is a list in which each index represents a line in the code.
3) Writing Itself: Writing itself also takes three steps using this method
    a. The first part is to write a loop that will copy the write each line (index of the array) until it arrives to the strings very own variable, in this case, the variable “quine” at line 6.
    b. The middle portion pertains to re-writing the quine string itself, you will want to print/write the string with the proper format. This string of code is unique to python, in which you will have to ensure that the (/) (forward slash) and (“) (apostrophe) must have an extra “/” in front so that it will be formatted properly in the output. This problem could be simplified by utilizing a single apostrophe instead, interchanging (\”) with (‘) . Other programming languages *can* be much simpler if they have alternative methods of writing special characters.
    c. The final portion is to finalize the ending portion of the code (generally the writing and execution of the payload). It will be printing/writing the code after the variable “quine”, in this case, being line 7 and onwards.
4) Program Execution: This is the portion of the program where you will want to execute the payload. In this program, it will simply be to print out the string (line 33), that was set up at the beginning of the program.

EXEC / EVAL String Trick + More Realistic Implementation

This program demonstrates how you can modify a global variable and retrieve a value from a string function. It is a simple script that will modify the variable count.

```
1. count = 0
2.
3. func0 = """
4. def foo():
5.     global count
6.     count = 10    # Modifies the global variable count
7.     return 100    #
8. """
9.
10.  func1 = """
11.  def main():
12.      file = open("temp2.py", 'w+')
13.      file.write("count = 0\\n")
14.      for num in range(0, 2):
15.          fnc_name = eval("func" + str(num))
16.          file.write("func" + str(num) + " = \\"\\"\\"")
17.
18.          for c in fnc_name:
19.              if c == "\\\\":
20.                  file.write("\\\\")
21.              file.write(c)
22.          file.write("\\"\\"\\"\\n")
23.
24.      file.write(func1)
25.      file.close()
26.      exec(func0)
27.      eval("foo()")
28.  main()
29.  """
30.
31.  def main():
32.      file = open("temp2.py", 'w+')
33.      file.write("count = 0\n")
34.      for num in range(0, 2):                  # Since there are 2
   functions to write
35.          fnc_name = eval("func" + str(num))  # Sets the variable to the
   string
36.          file.write("func" + str(num) + " = \"\"\"")
37.
38.          for c in fnc_name:
39.              if c == "\\":
40.                  file.write("\\") # An additional "\"
41.              file.write(c)
42.          file.write("\"\"\"\n")
43.
44.      file.write(func1)
45.      file.close()
46.      exec(func0)
47.      eval("foo()")
48.  main()
```

Exec/Eval Trick Breakdown:

By utilizing Exec/Eval, it is much easier to modify a global variable and to retrieve the value from the string functions. However, one should keep in mind that the exec/eval method does present a security risk (although it is probably not of concern). This is simply a proof of concept code of its ability to execute payload while being able to achieve its goal of self-replication. This method is much simpler than the generic method in which can be represented in three steps.

1. Write the function in a string format, don't forget to write/print the main function.
2. Have the main function write the string and execute upon it

Note:

- Some interesting tricks I utilized was Line 46 in which I executed the function, foo(). This is a workaround trick that dynamically executes a function. If one wanted to retrieve the outputted value of the function, one could retrieve it by calling the function by utilizing eval() as seen in Line 47. Of course this is a very jittery method of retrieving a value after executing, but it will work.

Section III:

Application

Propagation:

Given that quines can replicate itself, you can utilize its primary function to replicate the program through a file system, which can easily be used in malware. As seen in the Exec/Eval string trick example, it can easily be written upon another file.

Morphing:

Self Replication is only a small portion of the program's potential and by introducing the ability to mutate itself. A simple example is having the program decide whether it wants to install a function onto its descendants. For example, if it were to enter into a file system and you want it to stay there without self-replicating, you can remove its self-replication functionality. All you would have to do is index each function and have the function be removed from replication function if the program were to enter the filesystem. Unfortunately, adding a function would be a bit more complex, but it is probably simpler to just determine whether a function would be executed or not.

Quine Bomb:

If your program is given the permission to execute other programs, you will be provided the opportunity to create a program similar to a fork bomb. The quine will be able to constantly reproduce itself while also being able to execute one another.

The Rest:

The Rest is up to your imagination. You need to answer to the question. What can YOU do with a self-propagating program that can inject its own code onto other files? What can you do with a program that can manage its own functions? This journal was only written so that you can gain insight on its fundamentals and its possible applications.

Conclusion:

Quine is a fascinating concept that does not have very many possible applications, but still in itself is a concept that should be learned. A self-replicating program generally does not find itself having much use in the software world, which is commonly dominated by techniques that prefers enhancing databases or quicker algorithms. As such, this program is limited to primarily malware creation and disgusting code (Recommended to look up disgusting quines/code). However, it is simply a fascinating concept that should be explored only because of how interesting it is.

Key Note:

A quine is a powerful technique against reverse engineers, especially because it can become a challenge to understand how the program works based on the string itself. It can obstruct the graph view of most reversing programs (such as IDA), since it is formatted as strings. If you want to take it a step further to have the strings become less legible, it could be an encrypted, so it would require painstaking decryption work and difficulty to graph a string code.