Deceptive Python Decompilation

Software obfuscation is the science and art of modifying programs to hide certain aspects of it, for example what the program does or how it accomplishes a certain task. There are many ways of doing this and new creative ideas regularly pop up. The methods all have their own goals and trade-offs. The goal is to slow down reverse engineering of the program to exhaust the analyst's "budget" whether that is time, money or interest. Some obfuscation techniques are better at thwarting automated analysis, for example by exploiting assumptions and limitations in tools or detecting sandboxes and altering their behaviour, while others are more aimed at making life a pain for a human reverse engineer. The latter type can be achieved for example by adding a lot of useless stuff to the program or writing code that subverts expectations by seemingly doing one thing while actually doing something else[1].

Python Bytecode

The technique we will discuss here is a way of obfuscating Python bytecode. Before Python code is executed[2], it is parsed and transformed into Python bytecode. Usually programs are shipped as Python code which can be run but it is possible to only use the .pyc files containing the compiled bytecode. This is for example what py2exe does when building a stand-alone executable from Python code. The bytecode is then executed in the stack-based VM inside CPython.

Bytecode Decompilation Tricks

When trying to analyze Python bytecode it is desirable to turn it back into regular Python code for readability. There are multiple tools for doing this but the most popular is uncompyle6 which usually works amazingly well for decompiling Python bytecode. There exist multiple ways to fool it however. One way to mess up the decompilation is to craft Python bytecode that can't be produced from valid Python code, such as abusing exceptions for flow control. This is powerful because the decompilation will fail with no chance of recovery since the original code isn't actually Python to start with. The downside is that you need to either write the bytecode by hand or create your own compiler.
Another way is to abuse variable names to mess up the decompilation. The Python bytecode retains all the variable names to enable correct execution and interaction with other scripts. In contrast to the Python language, the CPython VM itself has no restrictions on variable naming. This can be abused by replacing all variable names with whitespace. It till transform code from:

into bytecode which decompiles to this:

The resulting code isn't even valid Python code. The downside with this technique is that it is very obvious that something went wrong and a slight adjustment to the decompilation process completely neutralizes it.
Inspired by this method, we can do something slightly more subtle. Consider the following code which almost implements RC4:

By replacing the name of the variable "OBFUSCATION" with "i = 0\n j", the code will decompile into this:

The decompiled code now implements RC4 correctly and would typically not warrant any further scrutiny since it's just an implementation of a well known algorithm. This is the key element because the decompiled code is now functionally different to the original code and its corresponding bytecode. In the initial version, the value of the variable i will be 255 when it enters the second loop but in the decompiled version it will be 0. If this function is used as part of an unpacker it will mean that even though the reverse engineer uses the correct key, the payload will never be succesfully decrypted. This could easily throw many reverse engineers off and make them waste a lot of time.
The key idea here is that the resulting decompilation is completely incorrect while at the same time looking completely correct and matching the expectations of the reverse engineer.

[1] See the *Underhanded C Contest* for great examples
[2] In the CPython implementation