Advanced Python Programming

Assignment 2

'Russian hacker'

Welcome to another assignment! This one is about being real Russian hacker that all Western world feel the fear of. We keep our plank as usual – **5 tasks** and **2 points** per each. So 10 points is what you've got when all is done. The formal methodology of giving your points is following: 1 point – "well..that's ok", 2 – "well done!", 0 – "where is the task n?".

Please, don't worry though. Our kindness *still* has no boundaries and you can always ask for any appropriate help. To submit your assignment:

- Upload your assignment to MS Teams as f.lastname.zip.
- Commit your solution to the GitHub repository of your team.

Per each program attach the version (`python -V`) you are using. Place it as a comment on the top. For some of your intentional omissions (if any) or negative disagreements (if any), you can give the written answers and pack them as a comment or a file.

DEADLINE: Thursday, 3:30pm (just before the lecture)

Let's get started..

1. Write a program that takes N arbitrary .py files and creates a neat table out of their execution time ranking by who-is-faster. Here is example:

```
$ ./compare.py (no arguments throw help)
usage: compare.py [files]
This program ...

$ ./compare.py src1.py src2.py src3.py
PROGRAM | RANK | TIME ELAPSED
src3.py   1      0.011488118s
src2.py   2      0.020115991s
src1.py   3      0.045907541s
```

   You can ask, what kind of programs can it be of any interest for us to compare? And here is the answer:

```
# src1.py
import math
def f():
    total = 0
    i = 0
    while i <= 1000000:
        total += math.sin(i)
        i += 1
    return total
print(f())

# src2.py
import math
def f():
    total = 0
    for x in range(1000000):
```

```
        total += math.sin(x)
    return total
print(f())

# src3.py
import math
def f():
    total = 0
    m = math.sin
    for x in range(1000000):
        total += m(x)
    return total
print(f())
```

2. From now on, our engineering interest is to give an insight "Why almost the same three programs produced quite different execution time?". In order to give a reasonable answer we need to dive one level below the ordinary level of abstraction. Namely, look at the bytecode. So our task is to reveal what the beast the "bytecode" is. To start something from, write a program that yield opcodes (and their arguments) for ordinary python programs. The opcodes are "machine" instructions produced out of your program for executing them on Python Virtual Machine. (which in our case is CPython implementation we believe)

```
$ ./bc_printer.py
usage: bc_printer.py -py src.py
This program ...

$ ./bc_printer.py -py src1.py
LOAD_CONST 0
LOAD_CONST None
IMPORT_NAME math
STORE_NAME math
LOAD_CONST 0
LOAD_CONST None
IMPORT_NAME timeit
...
```

3. It's good that we can get the bytecode from ordinary files. But what if we want to get them out of other sources? For example, code snippets and already compiled `.pyc` files (which you might encounter under `__pycache__` folder). So we want you to extend the program to support these two as follows:

```
$ ./bc_printer.py
usage: bc_printer.py -format src
This program ...

    -py src.py      produce human-readable bytecode from python file
    -pyc src.pyc    produce human-readable bytecode from compiled .pyc file
    -s "src"        produce human-readable bytecode from normal string

$ ./bc_printer.py -pyc __pycache__/src1.cpython-36.pyc
```

```
LOAD_CONST 0
LOAD_CONST None
...

$ ./bc_printer.py -s "print('Hello world')"
LOAD_NAME print
LOAD_CONST Hello world
...
```

4. This task is going to be small. Extend your program to produce (compile) `.py` files or code snippets right into `.pyc`. Don't bother about processing multiple files, or providing additional flags to change output dir or name. Just keep it simple and stupid. Also, rename your program and reorganize it to introduce sub-commands (or actions). It simplifies transition to the last task. And the last note – don't use `argparse` module! We want you to parse arguments manually (to make it easier let's assume all the options after the action go always in *pairs* as `-flag value`).

```
$ ls
bc.py src1.py
$ ./bc.py
usage: bc.py action [-flag value]*

This program ...

compile
    -py file.py     compile file into bytecode and store it as file.pyc
    -s "src"        compile src into bytecode and store it as out.pyc
print
    -py src.py      produce human-readable bytecode from python file
    -pyc src.pyc    produce human-readable bytecode from compiled .pyc file
    -s "src"        produce human-readable bytecode from normal string

$ ./bc.py compile -py src1.py
$ ./bc.py compile -s "print('Hello world')"
$ ls
bc.py src1.py src1.pyc out.pyc
```

5. The last one and the most complicated. Introduce new action `compare`. It must compare bytecode among different sources and produce neat table with stats of the used opcodes (and *only* them). It's better just to look at the example:

```
$ ls
bc.py src1.py src2.py src3.py
$ ./bc.py
usage: bc.py action [-flag value]*

This program ...

compile
```

```
    -py file.py     compile file into bytecode and store it as file.pyc
    -s "src"        compile src into bytecode and store it as out.pyc
print
    -py src.py      produce human-readable bytecode from python file
    -pyc src.pyc    produce human-readable bytecode from compiled .pyc file
    -s "src"        produce human-readable bytecode from normal string

compare -format src [-format src]+
                    produce bytecode comparison for giving sources
                    (supported formats -py, -pyc, -s)

$ ./bc.py compare -py src1.py -py src2.py -py src3.py
INSTRUCTION | src1.py     | src2.py     | src3.py
LOAD_FAST     15            8             3
POP_TOP       0             12            0
CALL_FUNCTION 9             0             0
LOAD_NAME     9             3             9
RETURN_VALUE  3             3             3
...
```

The table columns must be fixed. Make them at least 11-13 characters long. If the source name jumps out of the column – truncate. In the example above, the sum of opcodes per row has no interest, we rather concentrated on "peaks" among the files. So we want order them by that *peaks*. Table's output is fabricated and do not reflect real data for the examples given at the beginning. It will be your achievement to see what happened there.

Important: Attach the resulting table as a separate file (just as that `./bc.py args > result.out`). That file will be evaluated!

Resources

1. https://docs.python.org/3/library/dis.html
2. https://docs.python.org/3/library/py_compile.html
3. https://docs.python.org/3/library/inspect.html
4. http://www.goldsborough.me/python/low-level/2016/10/04/00-31-30-disassembling_python_bytecode/