Task 0: Implement a Virtual Machine with Python.

In this task you have to implement a simple VM capable of running a stream of specific bytecode.

The VM architecture must include the **stack** - a data structure, used for temporary storage of objects for further processing. Also, to implement variables, you have to add a few data structures which together are called **memory**. Memory can be implemented arbitrary (up to you) and may involve one or several data structures.

The task includes 5 parts. In each you have to implement certain functionality on top of the existing one. In part 1 you implement the VM from scratch.

Please note, that you won't need any external modules to finish the task (except `pytest` see below). Your code shall be tested in a raw Python 3.10 environment, but should be okay if you do not use features from Python 3.11 and later.

Please install pytest module to run tests.

```
pip install pytest
# or pip3 if you use your global environment.
```

Now you can run tests by running:

```
pytest tests/test_part?.py
```

PART 0 - module structure.

The module is assumed to be structured as:

```
xvm/__init__.py
xvm/vm.py
xvm/run.py
xvm/other_files_if_necessary.py
pyproject.toml
```

The tests are stored in `tests/` and must assume `xvm.vm` is importable.

!!! The completed task must be distributed as a package (both wheel and sdist). !!!

See: https://packaging.python.org/en/latest/flow/#build-artifacts

I highly recommend using `pyproject.toml`.

Parser

To run tests, first implement a simple line-by-line parser of textual bytecode. See `xvm/vm.py:parse_string` and tests.

PART 1 - math engine

First, implement a simple math engine VM.

Bytecode specification:

Binary

```
Pop two objects off the stack, compute a binary operator, push the result on the
stack.
OP_ADD - addition (a + b)
OP_SUB - subtraction (a - b)
OP_MUL - multiplication (a * b)
OP_DIV - division (a / b for floats, a // b for ints)

Note that operations (unlike functions) read their arguments from stack in order:
Stack: [b, a] -> a - b
```

Unary

```
Pop one object off the stack, compute a unary operator, push the result on the stack.
OP_SQRT - square root (math.sqrt(a))
OP_NEG - negate (-a)
OP_EXP - exponent (math.exp(a))
```

Memory operations

```
Load a variable `variable_name` and push it on the stack.
OP_LOAD_VAR <variable_name>

Load a constatnt `value` and push it on the stack.
```

```
OP_LOAD_CONST <value>

Pop an object off the stack, store it in the variable <variable_name>.
OP_STORE_VAR <variable_name>
```

Prefix OP_

In tests, you may find examples of code for the VM. As you may notice, opcodes are spelled without the prefix `OP_`. This does not make things much harder, but is more realistic.

Float and int

The only difference from the lecture's version of the VM is that you have to support both integers and floats and not mismatch them.

PART 2 - I/O

In Python, you can pass functions as variables everywhere. For instance, you can define a function:

```python
def my_func(a):
    print(a*2)
```

and then pass at as an argument to another function:

```python
def use_foo(foo, args):
    foo(args)

use_foo(my_func, 3)
# Output: 6
```

Also remember, that you can define a function anywhere, even inside of the other function:

```python
def my_func():
    def my_func_inside(a):
        print(a)

    my_func_inside(3)
    return my_func_inside

foo = my_func()
# Output: 3

foo(4)
# Output: 4
```

Now, the task is to implement I/O instructions. To make things more flexible, a VM must be able to take a function, i.e. a callback, as input. E.g.:

```python
VM(input_fn=input, print_fn=print)
```

In this case, when `OP_INPUT_*` is executed, the built-in `input` function will be called, prompting a user to enter the data (string or a number). Same for `OP_PRINT` (`print` function will be called).

I/O

I/O in our VM shall be implemented as follows. A VM has parameters:

- "print_fn" - a function taking a specified object for output.
- "input_fn" - a function that returns an object from the input.

```
Pop the top element off the stack, "print" it.
OP_PRINT

Read element from "input" and push it on the stack.
Note, that an input object can be a number or a string.
OP_INPUT_STRING
OP_INPUT_NUMBER
```

PART 3 - Serialization

Next, implement serialization and deserialization features:

- Implement a method `vm.run_code_from_json(json_filename)`. You can infer the spec from examples.
- Implement methods `vm.dump_memory(filename)` and `vm.dump_stack(filename)`. The methods serialize memory/stack into a pickle file.
- Implement methods `vm.load_memory(filename)` and `vm.load_stack(filename)`. The methods deserialize memory/stack from a pickle file and replace vm's memory/stack with the loaded data.

PART 4 - Command line debugger tool

Now, let's implement a command line tool. The debugger! Being one of the essential programs, it helps you with debugging your VM as well as the code you write for this VM.

Make sure to add it to the `pyproject.toml`, so that it can be invoked using

```
xvm
> ...
> ...
```

The following commands must be supported:

- `stack arg1` - prints `arg1` top elements on stack. If `arg1` is not specified (`stack`), print the entire stack.
- `memory` - prints all variables from memory.
- `print arg1` - print value of a variable `arg1`. When you implement function calls, `arg1` is only selected from the local frame.
- `load arg1` - load code from path `arg1`
- `run` - run loaded code, only stop on OP_BREAKPOINT instructions.
- `step` - run one next line from loaded code
- `list` - print up to 5 lines before the current instruction and up to 5 lines after it.

- `exec arg1 arg2...argN` - execute instruction with opcode `arg1`, and arguments `arg2, ...., argN`

PART 5 - Control flow

In this part you have to implement control flow instructions. These allow to represent ifs, loops and so on.

Control flow

```
Compare two objects from the stack.
If (a <OP> b) push 1 on the stack, else push 0 on the stack.

OP_EQ   ==
OP_NEQ  !=
OP_GT   >
OP_LT   <
OP_GE   >=
OP_LE   <=

Jumps and labels.
In our bytecode, a label is set using a specific OP_LABEL instruction.
It is recommended to parse the labels from the bytecode first, since they can be
defined further in the code. A label sets a possible destination for a jump.
If you OP_JMP <label_name>, the execution is continued from the instruction which
follows the label.
OP_LABEL <label_name>

Pop a number off the stack. If it is 1, jump to label, otherwise continue.
OP_CJMP <label_name>

Just jump to label.
OP_JMP <label_name>
```

PART 6 - Function calling

In this task, you have to implement function calling. It is expected to have rules of scope or stack frames. An example:

```
# Pseudocode:

def foo(a, b):
    c = a + b

a = 3
b = 4
foo(a, b)
print(c)
# Error: c is not defined!

a = 3
b = 4
```

```
c = 0
foo(a, b)
print(c)
# Output: 0. In our bytecode, no globals allowed!!! This simplifies your task.
```

Function calling

```
Call function. Arguments are assumed to be on stack in the reverse order.
The name of the function must be on top of the stack.
I.e. to call foo(a, b), push a, then push b, push "foo", and then call.
OP_CALL

Usage:
    ```
    # foo(a, b)
    OP_LOAD_CONST 3 # a
    OP_LOAD_CONST 5 # b
    OP_LOAD_CONST "foo" # function name
    OP_CALL
    ```

Return from function
OP_RET
```

The code for each function must be stored in a separate object. Now the code you pass to vm must be a dictionary with function names as keys and their bytecode as values. The global code (entry point) must have a key `"$entrypoint$"`.

```
code = {
    "foo": [... opcodes for foo ...],
    "bar": [... opcodes for bar ...],
    "$entrypoint$": [... main entrypoint ...],
}
vm.run_code(code)
```

Debugger updates:

Support new commands:

- `frame` - prints all variables current frame.
- `next` - run one next line from loaded code, if the next function is a function call, do not get inside - execute the function and only then pause.

PART 7 - Testing the debugger

Writing a code is not enough in the real world. We have to come up with ways to test it, find issues, and have a confidence that our system does work.

Come up with a testing scenario for the debugger.

- Option 1 (simple): write a set of instructions (in english or ukrainian) for a QA tester who will test your code.

- Option 2 (automation +10 points and my respect): write a script that tests the debugger automatically. You'd need to override the `stdin` for the `xvm` program

PART 8 - Writing a program

Now, you did not write this program for nothing, did you? Let's now use the VM and write some code for it!

Pick one option and implement it in XVM dialect. Which capabilities do you think our XVM lacks to actually write more impressive programs, like text games or advanced algorithms?

- Option 1. Greatest Common Divisor (Euclidean Algorithm) Input: a, b. While b != 0, set a, b = b, a % b. Output: a. Hint: Implement OP_MOD that does a % b first.
- Option 2. Exponentiation by Repeated Multiplication Input: x, n. Compute x^n with a loop. If you want a challenge: implement fast exponentiation with divide-and-conquer and recursion.
- Option 3. Prime Checker Input: n. Loop i from 2 to sqrt(n). If n % i == 0 → print "Not prime", else "Prime".
- Option 4. Sum of Digits Input: n. While n > 0: digit = n % 10; sum += digit; n = n // 10. Output sum.
- Option 5. Palindrome Number Check Input: n. Reverse it digit by digit, compare with original.