

Завдання 0: Реалізувати Віртуальну Машину за допомогою Python.

- [Завдання 0: Реалізувати Віртуальну Машину за допомогою Python.](#)
 - [ЧАСТИНА 0 - структура модуля.](#)
 - [!!! Рішення повинно включати в себе згенерований пакет \(wheel та sdist версії\).](#)
 - [!!!](#)
 - [Парсер](#)
 - [ЧАСТИНА 1 - математичний рушій](#)
 - [Бінарні операції](#)
 - [Унарні операції](#)
 - [Операції з пам'яттю](#)
 - [Префікс ОР](#)
 - [Float та int](#)
 - [ЧАСТИНА 2 - I/O](#)
 - [I/O](#)
 - [ЧАСТИНА 3 - Серіалізація](#)
 - [ЧАСТИНА 4 - Дебагер.](#)
 - [ЧАСТИНА 5 - Конструкції управління](#)
 - [Операції для управління потоком виконання](#)
 - [ЧАСТИНА 6 - Функції](#)
 - [Виклик функції](#)
 - [Оновлення для дебагера:](#)
 - [ЧАСТИНА 7 - Тестування дебагера та VM](#)
 - [ЧАСТИНА 8 - Пишемо програму](#)

В цьому завданні, вам необхідно реалізувати просту VM, здатну виконувати потік спеціальних інструкцій.

Архітектура VM має включати **стек** - структуру даних, що використовується для тимчасового збереження об'єктів для подальшої обробки. Також для реалізації змінних, потрібно буде додати інші структури даних, які в сукупності називаються **пам'яттю**. Реалізація може бути довільною і включати одну або декілька структур даних.

Завдання складається з 5 частин. Кожна включає в себе певний додатковий функціонал, що потрібно реалізувати поверх існуючого. Частина 1 реалізується з нуля.

Для виконання роботи, зовнішні бібліотеки не потрібні (окрім `pytest`, див. нижче). Ваш код буде протестований в звичайному середовищі Python 3.10, але інші версії також повинні підходити, якщо ви не використовуватимете нових функцій з Python 3.11+.

Будь ласка, встановіть модуль `pytest` щоб тестувати ваше рішення.

```
pip install pytest
# або pip3 якщо ви використовуєте глобальне середовище.
```

Тепер ви можете запускати тести:

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

ЧАСТИНА 0 - структура модуля.

Рішення повинно бути у вигляді модуля, який має наступну структуру:

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

Всі тести зберігаються в папці `tests/`. Припускається, що `xvm.vm` можна імпортувати.

!!! Рішення повинно включати в себе згенерований пакет (wheel та sdist версії). !!!

Див: <https://packaging.python.org/en/latest/flow/#build-artifacts>

Дуже рекомендую використовувати `pyproject.toml`.

Парсер

Щоб запускати тести, спочатку імплементуйте простий парсер тексту байткоду. Див.

```
xvm/vm.py:parse_string
```

 та тести.

ЧАСТИНА 1 - математичний рушій

Спершу, імплементуйте просту ВМ для математичних операцій.

Специфікація байткоду:

Бінарні операції

Забрати два об'єкти зі стеку, обчислити бінарний оператор, покласти результат на стек.
OP_ADD – додавання ($a + b$)
OP_SUB – віднімання ($a - b$)
OP_MUL – множення ($a * b$)
OP_DIV – ділення (a / b)
OP_DIV – ділення (a / b для float, $a // b$ для int)

Зауважте, що операції (на відміну від функцій) читають аргументи зі стеку по порядку:
Стек: `[b, a]` -> $a - b$

Унарні операції

Забрати один об'єкт зі стеку, обчислити унарний оператор, покласти результат на стек.
OP_SQRT – квадратний корінь (`math.sqrt(a)`)
OP_NEG – унарний мінус ($-a$)
OP_EXP – експонента (`math.exp(a)`)

Операції з пам'яттю

Завантажити значення змінної ``variable_name`` і покласти його на стек.
`OP_LOAD_VAR <variable_name>`

Завантажити константу ``value`` та покласти його на стек.
`OP_LOAD_CONST <value>`

Забрати об'єкт зі стеку, присвоїти змінній `<variable_name>` значення цього об'єкту.
`OP_STORE_VAR <variable_name>`

Префікс OP_

В тестах ви можете знайти приклади коду для VM. Зверніть увагу, що там опкоди записуються без префіксу `OP_`. Ця деталь не надто ускладнює імплементацію, але робить синтаксис більш реалістичним.

Float та int

Єдина відмінність такої VM від лекційної полягає в тому, що вона має підтримувати цілі числа та дійсні числа і не сплутувати їх. Конвертацію реалізовувати не потрібно.

ЧАСТИНА 2 - I/O

В Python, ви можете передавати функції як змінні/аргументи будь-куди. Наприкла, ви можете написати функцію

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

і потім передавати її як аргумент в іншу функцію:

```
def use_foo(foo, args):  
    foo(args)  
  
use_foo(my_func, 3)  
# Output: 6
```

Також пам'ятайте, що ви можете написати функцію будь-де, навіть в середині іншої функції.

```
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
```

Вашим завданням буде реалізація інструкцій пов'язаних із I/O. Щоб зробити реалізацію VM більш гнучкою, VM повинен приймати в конструктор відповідні функції на вхід. Тобто:

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

В даному випадку, коли `OP_INPUT_*` буде виконуватись, вбудовану функцію `input` буде викликано, а отриманий результат оброблено та покладений на стек. Ця функція дає користувачу можливість ввести в терміналі число або функцію. Така сама ситуація і з `OP_PRINT` (`print` буде викликано).

I/O

I/O в нашій VM буде реалізовано наступним чином: Клас VM отримує параметри:

- "print_fn" - функція, що приймає на вхід об'єкт для виводу.
- "input_fn" - функція, що повертає об'єкт із вхідних даних.

Забирає об'єкт зі стеку та передає його значення в "print_fn".
`OP_PRINT`

Читати об'єкт (рядок або число) та покласти його на стек.
`OP_INPUT_STRING`
`OP_INPUT_NUMBER`

ЧАСТИНА 3 - Серіалізація

Далі, реалізуйте функції серіалізації та десериалізації:

- Метод `vm.run_code_from_json(json_filename)` . Див тести в якості прикладу використання.
- Методи `vm.dump_memory(filename)` та `vm.dump_stack(filename)` мають серіалізувати пам'ять та стек в pickle файл.
- Методи `vm.load_memory(filename)` та `vm.load_stack(filename)` мають десериалізовувати пам'ять та стек із pickle файлу, та замінюють поточні пам'ять та стек VM.

ЧАСТИНА 4 - Дебагер.

Тепер, давайте імплементуємо інструмент командного рядка. Дебагер! Це одна з найбільш важливих програм для розробників - вона дозволяє вам відлагоджувати вашу VM, а також код, який ви пишете для цієї VM.

Не забудьте додати його в перелік програм у `pyproject.toml` , щоб можна було запускати дебагер командою

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

Спочатку додайте інструкцію `OP_BREAKPOINT` . При звичайному виконанні (без дебагера) вона нічого не робить - NO-OP.

Наступні команди дебагера повинні підтримуватись:

- `stack arg1` - вивести `arg1` верхніх елементів зі стеку. Якщо `arg1` не вказано (`stack`), то вивести увесь стек.
- `memory` - вивести усі змінні в пам'яті VM.
- `print arg1` - вивести значення змінної з назвою `arg1` . Після імплементації викликів функцій, `arg1` треба шукати тільки у локальному фреймі.
- `load arg1` - завантажити програму (код) з файлу зі шляхом `arg1`
- `run` - запустити виконання завантаженого коду, зупинитись тільки при виконанні `OP_BREAKPOINT` інструкції.
- `step` - виконати наступну інструкцію із завантаженого коду.
- `list` - вивести до 5 інструкцій перед поточною і до 5 після неї.
- `exec arg1 arg2...argN` - виконати інструкцію з оп-кодом `arg1` та аргументами `arg2, ..., argN`

ЧАСТИНА 5 - Конструкції управління

В цій частині ви реалізуєте інструкції для конструкцій управління. Вони дозволять представляти `if`, `for/while` та `in`.

Операції для управління потоком виконання

Забрати два об'єкта зі стеку та порівняти їх.

Якщо (`a <OP> b`), покласти 1 на стек, інакше покласти 0 на стек.

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

Стрибки та мітки.

В нашому байткодi, мітка може бути поставлена спеціальною інструкцією `OP_LABEL`.

Рекомендовано парсити всі мітки з байткоду до його виконання, адже операції стрибків можуть іти раніше міток.

Мітка позначає спеціальне місце з іменем, куди можна здійснити стрибок. Якщо `OP_JMP <label_name>` виконується, то далі виконання буде продовжено з інструкції, що іде після `OP_LABEL <label_name>`.

```
OP_LABEL <label_name>
```

Забрати об'єкт (число) зі стеку. Якщо це 1, здійснити стрибок до мітки, інакше продовжити виконання далі.

```
OP_CJMP <label_name>
```

Просто здійснити стрибок до мітки.

```
OP_JMP <label_name>
```

ЧАСТИНА 6 - Функції

Тепер, реалізуйте функції та їх виклик. Очікується, що вони слідуватимуть правилам `scope'u`, тобто змінні із однієї функції не можна побачити в середині іншої:

```
# 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.
```

Виклик функції

Перед викликом потрібно покласти аргументи цієї функції на її стек в зворотньому порядку, а також покласти на стек її ім'я.
Наприклад, щоб викликати `foo(a, b)`, покладіть `a`, покладіть `b`, покладіть `"foo"`, а потім викликайте `OP_CALL`.

`OP_CALL`

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

Повернутись із функції. Якщо функція щось повертає, це щось буде лежати на горі стеку.
`OP_RET`

Код для кожної функції має зберігатись окрему. Тому, код який ви передаєте в `vm.run_code` буде словником, з ключами - назвами функцій, а значеннями - списками інструкцій. Глобальний код (вхідна точка) є функцією з спеціальною назвою `"$entrypoint$"`.

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

Оновлення для дебагера:

Додайте підтримку наступних команд:

- `frame` - вивести усі змінні із поточного фрейму.
- `next` - запустити наступну інструкцію із завантаженого коду. На відміну від `step`, при виконанні інструкції виклику функції почекати виконання цієї функції і переступити її.

ЧАСТИНА 7 - Тестування дебагера та VM

В реальному світі, написання коду - це ще не все. Нам потрібно придумати способи його протестувати, знайти проблеми, і здобути впевненість у стабільності нашої програми.

Придумайте тестовий сценарій для дебагера.

- Опція 1 (проста): написати перелік інструкцій (англійською чи українською) для уявного QA тестувальника (мене), який буде тестувати ваш код.
- Опція 2 (автоматизація - +10 балів та моя повага :-))): написати скрипт, який автоматично протестує дебагер за вашим сценарієм. Буде потрібно підкласти `stdin` для `xvm` програми.

ЧАСТИНА 8 - Пишемо програму

Нарешті, остання частина. Ви ж не просто так писали це все? Так от давайте використаємо вашу VM і напишемо трохи коду для неї!

Оберіть одну опцію і реалізуйте її на нашому діалекті XVM. Як ви вважаєте, яких функцій не вистачає нашому XVM щоб реалізувати дійсно складні програми, як от текстові ігри чи просунуті алгоритми?

- Опція 1. Найбільший Спільний Дільник (Euclidean Algorithm) Input: a, b. While b != 0, set a, b = b, a % b. Output: a. Підказка: Реалізуйте `OP_MOD` який робить `a % b`.
- Опція 2. Підняття у ступінь через повторне множення. Input: x, n. Обчисліть x^n у циклі. Більш цікавий варіант: реалізуйте підняття у ступінь через рекурсію та divide-and-conquer підхід.
- Опція 3. Перевірка на простоту. Input: n. Loop i from 2 to \sqrt{n} . If `n % i == 0` → print "Not prime", else "Prime".
- Опція 4. Сума цифр числа. Input: n. While `n > 0`: `digit = n % 10`; `sum += digit`; `n = n // 10`. Output sum.
- Опція 5. Перевірка числа на паліндром Input: n. Пройдіть з кінця по цифрах, порівняйте з числом.