Python. ДЗ 1
Интерпретатор байткода

Выполнение кода в интерпретаторе CPython происходит в две фазы: сначала синтаксис языка компилируется в более простой, называемый *байткодом*, а после этот байткод выполняется в виртуальной машине.
Процесс можно провести и собственноручно. Для этого надо последовательно вызвать в интерпретаторе функции `compile` и `exec`. Изучить то, как устроен байткод, можно вызвав ваш скрипт с ключиком `-m dis` или воспользовавшись функцией `dis.dis`.

В данном задании вам предлагается написать интерпретатор байткода питона на питоне (пользоваться exec и eval в качестве реализации, ясное дело, запрещено).

1. **Интерфейс**

   Ваш интерпретатор должен быть модулем из которого можно заимпортировать класс VirtualMachine, который будет содержать метод run_code. Этот метод должен работать аналогично встроенной функции exec, то есть уметь принимать в себя как `codeobject`, возвращаемый функцией `compile`, так и просто строку с кодом на питоне внутри (тогда нужно вызывать `compile` внутри метода).

2. **О том, как устроен байткод и интерпретатор**

   Интерпретатор питона - это *виртуальная машина*, или если более точно, *стековая машина*. Стековая машина оперирует несколькими стеками, а команды байткода задают какие-то операции над этими стеками. Интерпретатор принимает на вход `codeobjects`, в которых содержится набор инструкций (собственно, байткод) и какая-то вспомогательная информация для их работы.

   Проще всего понять принцип на примере. Давайте рассмотрим такое выражение: `print(1 + 2 * 3)`

   Его байткод можно достать например так:

   ```
   code = list(compile('print(1 + 2 * 3)', '<test>', 'exec').co_code)
   ```

   code – это codeobject, именно такие мы должны интерпретировать

   ```
   <code object <module> at 0x10a496780, file "<test>", line 1>
   ```

   байткод внутри доступен так

   ```
   list(code.co_code)
   ```

   Результат - список операций, которые надо выполнять последовательно

   ```
   [101, 0, 0, 100, 5, 0, 131, 1, 0, 1, 100, 3, 0, 83]
   ```

   С тем какая цифра какую операцию задает – тоже поможет `dis`

   ```
   dis.opname[131] == 'CALL_FUNCTION'
   ```

   Наглядно увидеть весь байткод функции можно через `dis.dis`

```
In [13]: a = compile('x = 2; y = 2; print(x + y)', '<test>', 'exec')

In [14]: dis.dis(a)
  1           0 LOAD_CONST               0 (2)
              3 STORE_NAME               0 (x)
              6 LOAD_CONST               0 (2)
              9 STORE_NAME               1 (y)
             12 LOAD_NAME                2 (print)
             15 LOAD_NAME                0 (x)
             18 LOAD_NAME                1 (y)
             21 BINARY_ADD
             22 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             25 POP_TOP
             26 LOAD_CONST               1 (None)
             29 RETURN_VALUE
```

Как уже говорилось выше, инструкции байткода взаимодействуют со стеком.

Операция LOAD_CONST кладёт константу на вершину стека, STORE_NAME сохраняет верхнее значение со стека в текущие локальные переменные. LOAD_NAME работает ровно наоборот, достаёт переменную из словарика локальных или глобальных переменных и кладёт на стек. BINARY_ADD достаёт два объекта с вершины стека и кладёт назад их сумму. CALL_FUNCTION вызывает функцию с какими-то аргументами (в данном случае print с одним аргументом).

С тем, как устроены более сложные структуры байткода вам предлагается разобраться самим, поисследовав тривиальные примеры.

3. **Как начать и что сделать**

Настоятельно рекомендуется начать с того, что бы настроить себе тестирующий скрипт и тесты. Для этого напишите десяток очень простых скриптов, использующих только какие-то базовые операции, сохраните их и напишите простую обертку, которая будет вызывать ваш интерпретатор и функцию exec, сравнивать их результаты и считать расхождения. Добейтесь отсутствия расхождений, добавьте тестов. Повторите.

Приблизительно в таком порядке стоит разбирать и реализовывать механизмы интерпретатора, каждое следующее реализовать сложнее чем предыдущее:

- Функция print и арифметика, скобки, переменные арифметические и логические типы
- Условия и циклы
- Строки, форматирование
- Списки, словари, кортежи, генераторы списков, распаковка, слайсы
- Функции и фреймы, замыкания, (*) декораторы
- (*)Классы

И не забудьте для начала зучить документацию по модулю dis - `https://docs.python.org/3/library/dis.html`.

Там описаны все существующие операции байткода.

Hint: в Python3.6 у dis появился метод get_instructions, который позволяет просто получить все вызовы операций вместе с их аргументами.

4. **Формат тестов**

Решение сдаваемое в энитаск должно иметь такую структуру:

```
from utils import run_vm  # вызов функции тестирующего фреймворка

class VirtualMachine(object):
    def run_code(self, code):
        """
        :type: code_obj
        """
        pass

if __name__ == "__main__":
    vm = VirtualMachine()
    run_vm(vm)
```

Здесь определен класс интерпретатора VirtualMachine с методом run_code, объект которого передается в тестирующую функцию run_vm. Тестирующая функция для каждого теста вызовет на сервере метод run_code, выведет в файл результат работы метода и сравнит с эталоном.

5. **Оценка**

Частичное решение тоже является решением и будет оценено. Баллы за каждый тест будут даваться пропорционально количеству операций в них и сложности их реализации. Итоговая оценка = взвешенная сумма количества баллов за пройденные тесты.

6. **Коментарии**

- Задание намеренно не содержит полной спецификации работы виртуальной машины. Разобраться в том как работают, например, функции – ваша задача, не просите семинаристов вам это объяснить.
- Не стесняйтесь гуглить, читать и обсуждать между собой какие-то вопросы которые вам непонятны. В интернете много детальных разборов этой темы. (Но помните, что заимствование кода, как из сети, так и у однокурсников карается полным незачетом задания всем причастным).
- Часть тем, затронутых в условии и необходимых для выполнения задания (например модули и импорт), пока не рассказана на лекциях, но будет затронута на одной из ближайших.

7. **Успехов!**

8. **Advanced**

   Ссылки для более глубокого изучения:

   - Академический проект интерпретатора для PY27 и PY33, снабженный множеством комментариев, но не лишенный проблем:
     `https://github.com/nedbat/byterun`
     Его детальное обсуждение в блоге:
     `http://www.aosabook.org/en/500L/a-python-interpreter-written-in-python.html`
   - Исходный код родного интерпретатора PY36 - поможет разобраться с тонкостями `https://github.com/python/cpython/blob/master/Python/ceval.c`
   - `https://habrahabr.ru/company/buruki/blog/189972`
   - `http://akaptur.com/blog/2013/11/17/introduction-to-the-python-interpreter-3`
   - `https://www.quora.com/What-is-Python-byte-code`
   - `http://security.coverity.com/blog/2014/Nov/understanding-python-bytecode.html`
   - `http://stackoverflow.com/questions/2220699/whats-the-difference-between-eval-exec-and-compile-in-python`
   - `http://multigrad.blogspot.ru/2014/06/fun-with-python-bytecode.html`