ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 4

Мова Ассеблера

Весна 2021

Python - це об’єктно-орієнтована, інтерпретована мова. Внутрішньо інтерпретатор Python програму-Python перетворює в байт-код і надалі інтерпретується за допомогою віртуальної машини. Більшість сучасних мов програмування підтримують абстракції високого рівня, тоді як інструкції віртуальної машини наближаються до машинної мови інструкцій, які підтримуються апаратною архітектурою, що полегшує інтерпретацію байт-коду, ніж інтерпретацію вихідної програми.

Перевага реалізацій віртуальних машин полягає в розділенні відображення від абстракцій високого рівня до машинних інструкцій низького рівня на дві частини: абстракції високого рівня до байт-коду та байт-коду до машинних інструкцій.
Хоча байт-код є абстракцією вищого рівня, ніж машинна мова, це не так. Як програмісти, якщо ми розуміємо, як основна машина виконує наші програми, нам краще підготуватися до правильного вибору того, як ми програмуємо. Так само важливо, що розуміння того, як виконуються програми, може допомогти ми діагностуємо проблеми, коли щось піде не так.

Ця лекція представляє програмування мови асемблера на мові байт-кодів віртуальної машини Python. Віртуальна машина Python є внутрішнім компонентом інтерпретатора Python і не доступна для безпосереднього використання. Натомість був розроблений інтерпретатор байт-кодів під назвою JCoCo, який імітує підмножину поведінки віртуальної машини Python 3.2. Замість того, щоб безпосередньо писати файли байт-кодів, JCoCo підтримує мову збірки віртуальних машин Python.

Вивчаючи мову асемблерів, ми обмежимось лише підмножиною Python. JCoCo підтримує булеві значення, цілі числа, рядки, плаваючі символи, кортежі, списки та словники. Він підтримує визначення класів та функцій та виклики функцій. Він також підтримує більшість інструкцій віртуальної машини Python, включаючи підтримку умовного виконання, ітерації та обробки винятків. Він не підтримує імпорт модулів або коду рівня модуля. JCoCo відрізняється від Python тим, що вимагає основної функції, де починається виконання програми, зібраної у JCoCo.

Щоб запустити програму мови асемблера, її потрібно спочатку зібрати, потім її можна виконати. Віртуальна машина JCoCo включає асемблер, тому збірка не є окремим кроком. Програміст мови асемблерів пише програму у форматі мови асемблера JCoCo, надаючи її JCoCo, який потім збирає та інтерпретує програму.

Основною відмінністю мови збірки JCoCo від байт-коду є наявність міток у форматі мови збірки. Мітки - це цілі інструкцій, які змінюють звичайну послідовність виконання інструкцій.

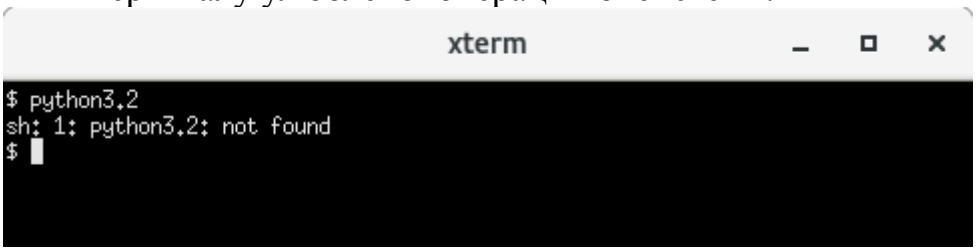
Інструкції, такі як інструкції з розгалуження та переходу, набагато легше розшифрувати, якщо там сказано "перейти до циклу1", а не "перейти до адреси 63". Звичайно, інструкції байт-коду кодуються як самі цифри, тому асемблер перекладає "перейти до циклу1" приблизно до "48 63", що, звичайно, потребує посібника для розшифровки.

Навчитися програмувати в збірці не так вже й складно, коли ти дізнаєшся, як такі конструкції, як цикли while, для циклів, операторів if-then, визначень функцій та викликів функцій реалізовані мовою асемблера. Маніпулювання рядками та списками - ще одна навичка, яка допомагає, якщо у вас є приклади для наслідування.

Десемблер - це інструмент, який приймає програму машинної мови та виробляє її асемблерну версію. Python включає модуль під назвою dis, який включає десемблер. Коли ви пишете програму Python, вона аналізується і перетворюється в байт-код при читанні інтерпретатором. Розбірник модулів dis створює програму мови збірки з цього байт-коду. JCoCo включає власний розбірник, який використовує модуль Python dis і видає вихідні дані, придатні для віртуальної машини JcoCo.

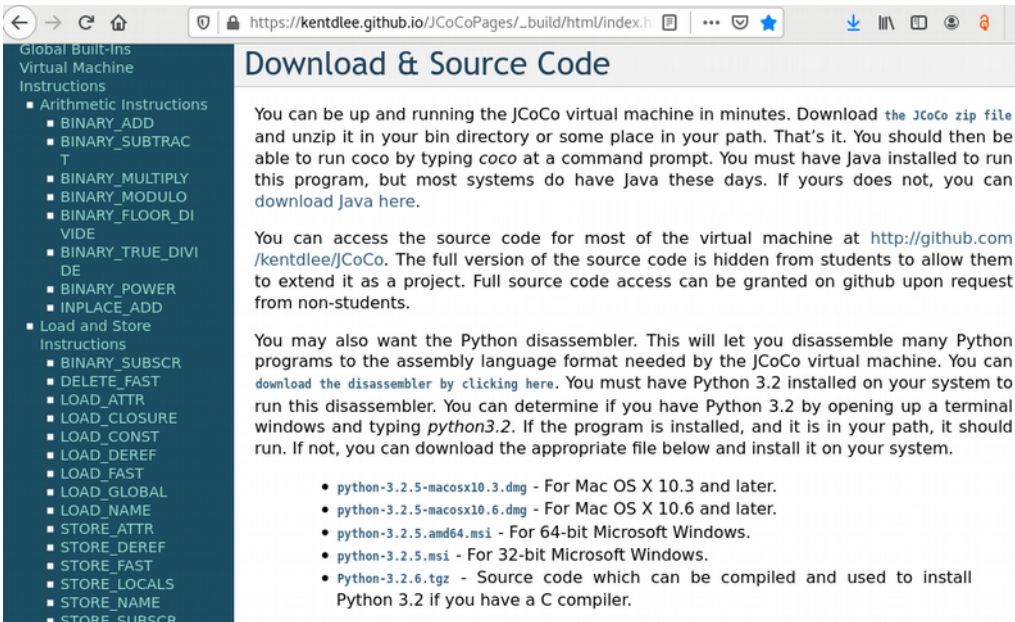
Існування розбірника для JCoCo означає, що вивчення мови асемблерів настільки ж просто, як написання програми Python та запуск її через розбірник, щоб побачити, як вона реалізована на мові асемблера. Це означає, що ви можете дізнатися, як реалізований Python, вивчаючи мову асемблерів! Оскільки віртуальна машина Python не гарантована зворотно сумісною, ви повинні використовувати Python 3.2 під час розбирання програм, тому переконайтеся, що у вашій системі встановлена версія 3.2.

Щоб перевірити це, ви можете спробувати набрати “python3.2” у вікні терміналу улюбленої операційної системи.



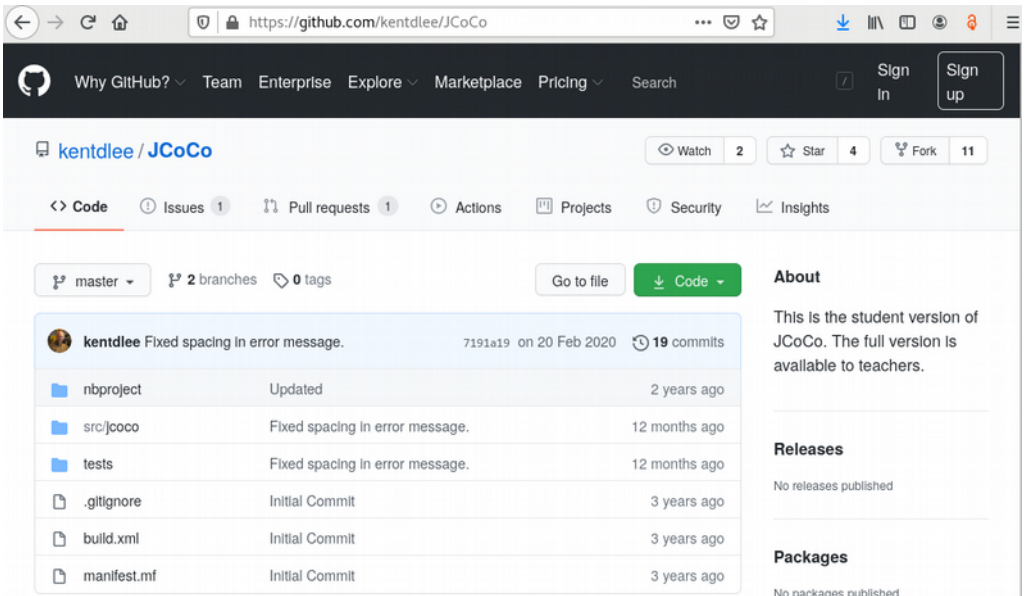
Якщо там написано, що команду не знайдено, швидше за все, у вас не встановлений Python 3.2.

У цьому випадку ви можете завантажити його з http://python.org або безпосередньо з веб-сайту JCoCo за адресою http://cs.luther.edu/~leekent/JCoCo . Решта цієї лекції і наступні знайомлять вас з програмуванням на мові асемблера за допомогою віртуальної машини JCoCo.


https://kentdlee.github.io/JCoCoPages/_build/html/index.h
Global Built-Ins
Virtual Machine Instructions
Arithmetic Instructions
BINARY_ADD
BINARY_SUBTRACT
BINARY_MULTIPLY
BINARY_MODULO
BINARY_FLOOR_DIVIDE
BINARY_TRUE_DIVIDE
BINARY_POWER
INPLACE_ADD
Load and Store Instructions
BINARY_SUBSCR
DELETE_FAST
LOAD_ATTR
LOAD_CLOSURE
LOAD_CONST
LOAD_DEREF
LOAD_FAST
LOAD_GLOBAL
LOAD_NAME
STORE_ATTR
STORE_DEREF
STORE_FAST
STORE_LOCALS
STORE_NAME
STORE_SUBSCR
Download & Source Code
You can be up and running the JCoCo virtual machine in minutes. Download the JCoCo zip file and unzip it in your bin directory or some place in your path. That's it. You should then be able to run coco by typing coco at a command prompt. You must have Java installed to run this program, but most systems do have Java these days. If yours does not, you can download Java here.
You can access the source code for most of the virtual machine at http://github.com/kentdlee/JCoCo. The full version of the source code is hidden from students to allow them to extend it as a project. Full source code access can be granted on github upon request from non-students.
You may also want the Python disassembler. This will let you disassemble many Python programs to the assembly language format needed by the JCoCo virtual machine. You can download the disassembler by clicking here. You must have Python 3.2 installed on your system to run this disassembler. You can determine if you have Python 3.2 by opening up a terminal windows and typing python3.2. If the program is installed, and it is in your path, it should run. If not, you can download the appropriate file below and install it on your system.
python-3.2.5-macosx10.3.dmg - For Mac OS X 10.3 and later.
python-3.2.5-macosx10.6.dmg - For Mac OS X 10.6 and later.
python-3.2.5.amd64.msi - For 64-bit Microsoft Windows.
python-3.2.5.msi - For 32-bit Microsoft Windows.
Python-3.2.6.tgz - Source code which can be compiled and used to install Python 3.2 if you have a C compiler.

Ви можете завантажити повну двійкову реалізацію віртуальної машини JCoCo, перейшовши за адресою http://cs.luther.edu/~leekent/JCoCo . Завантажте zip-файл, що містить скрипт кокосової оболонки, який запускає віртуальну машину Java, у файлі jar JCoCo.

Ви також можете перейти до github і отримати вихідний код для проекту JCoCo зі зниженою функціональністю за адресою http://github.com/kentdlee/JCoCo .


https://github.com/kentdlee/JCoCo
Why GitHub?
Team
Enterprise
Explore
Marketplace
Pricing
Search
Sign In
Sign up
kentdlee / JCoCo
Watch 2
Star 4
Fork 11
Code
Issues 1
Pull requests 1
Actions
Projects
Security
Insights
master
2 branches
0 tags
Go to file
Code
kentdlee Fixed spacing in error message.
7191a19 on 20 Feb 2020
19 commits
nbproject Updated 2 years ago
src/jcoco Fixed spacing in error message. 12 months ago
tests Fixed spacing in error message. 12 months ago
.gitignore Initial Commit 3 years ago
build.xml Initial Commit 3 years ago
manifest.mf Initial Commit 3 years ago
About
This is the student version of JCoCo. The full version is available to teachers.
Releases
No releases published
Packages
No packages published

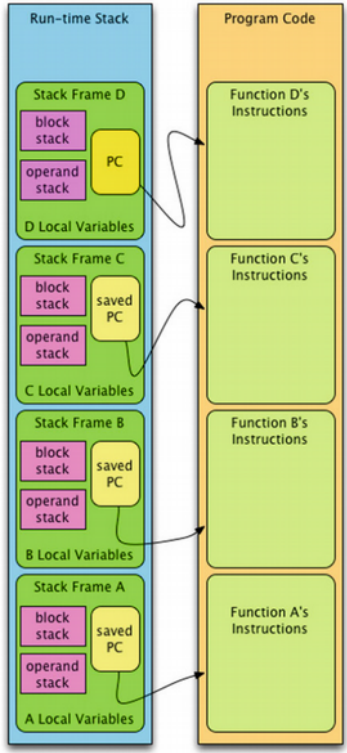
Огляд віртуальної машини JCoCo

JCoCo, як і Python, - це віртуальна машина або інтерпретатор для інструкцій байт-коду. JCoCo написано на Java із використанням об'єктно-орієнтованих принципів і не зберігає своїх інструкцій у фактичному форматі байт-коду. Натомість він читає файл мови асемблера і збирає його, будуючи внутрішнє представлення програми як послідовність функцій, кожна з яких має власну послідовність інструкцій байт-коду. CoCo - ще одна реалізація цієї віртуальної машини, реалізована на C ++.

Програма JCoCo, як програми на інших мовах програмування, використовує стек часу виконання для зберігання інформації про кожну функцію, що викликається під час виконання програми. Кожен виклик функції в програмі JCoCo призводить до створення нового об'єкта кадру стека, який створюється і переміщується в стек часу виконання.

Коли функція повертається, відповідний кадр стека вискакує із стека часу виконання та відкидається. На малюнку 4.1 зображено чотири активні виклики функцій. Функція A називається функцією B, яка викликає функція C, яка викликала функцію D перед поверненням будь-якої з функцій. Верх стека знаходиться у верхній частині рис. 4.1. Кожен кадр стека містить усі локальні змінні, які визначені у функції. Кожен кадр стека також містить два додаткові стеки, стек операндів і стек блоків.

Рис.4.1
Віртуальна машина JCoCo



JCoCo, як і віртуальна машина Python, є своєю архітектурою на основі стеку. Це означає, що операнди для інструкцій переміщуються в стек операндів. Інструкції віртуальної машини виводять свої операнди зі стеку операндів, виконують заплановані операції та переносять результати в стек операндів. Більшість процесорів не базуються на стеках. Натомість вони мають регістри загального призначення для проміжних результатів. Архітектури на основі стеку керують набором проміжних результатів як стеком, а не змушують програміста відстежувати, які регістри містять які результати.

Абстракція стеку трохи полегшує життя програміста асемблерної мови. Стек операндів використовується віртуальною машиною для зберігання всіх проміжних результатів виконання команд. Цей стиль обчислень використовується довгий час, від головних комп'ютерів Hewlett Packard 1960-х до 1980-х років до калькуляторів, зроблених Hewlett Packard сьогодні. Віртуальна машина Java, або JVM, є ще одним прикладом машини стека.

“Лихе горе — початок”.

До складу JCoCo входить розбірник, який працює з Python 3.2 для розбирання програм Python на програми асемблера JCoCo, що забезпечує чудовий спосіб вивчити програмування мови асемблера за допомогою віртуальної машини JCoCo. Розглянемо наступну програму Python, яка додає 5 і 6 разом і друкує суму на екран.

```
1  from disassembler import *
2  import sys
3
4  def main():
5      x=5
6      y=6
7      z=x+y
8      print(z)
9
10 if len(sys.argv) == 1:
11     main()
12 else:
13     disassemble(main)
```

Запуск програми **`addtwo.py`** з *пайтоном* дає наступний результат. Зверніть увагу, що аргумент 1 необхідний для отримання вихідних даних збірки через код у рядках 10-13 програми Python.

```
MyComputer> python3.2 addtwo.py 1
Function: main/0
Constants: None, 5, 6
Locals: x, y, z
Globals: print
BEGIN
                  LOAD_CONST 1
                  STORE_FAST 0
                  LOAD_CONST 2
                  STORE_FAST 1
                  LOAD_FAST 0
                  LOAD_FAST 1
                  BINARY_ADD
                  STORE_FAST 2
                  LOAD_GLOBAL 0
                  LOAD_FAST 2
                  CALL_FUNCTION 1
                  POP_TOP
                  LOAD_CONST 0
                  RETURN_VALUE
END
MyComputer> python3.2 addtwo.py  1 > addtwo.casm
```

Розбирач друкує програму мови збірки до стандартного виводу, яким зазвичай є екран. Другий запуск програми **`addtwo.py`** перенаправляє стандартний вивід у файл, який називається **`addtwo.casm`**. **Casm** - це розширення, вибране для мовних файлів асемблера JCoCo, і розшифровується як CoCo Assembly. Цей файл **`casm`** містить усі рядки між двома підказками **`MyComputer`** вище. Для запуску цієї програми ви можете викликати віртуальну машину JCoCo, як показано тут.

```
MyComputer> coco -v addtwo.casm
Function: main/0
Constants: None, 5, 6
Locals: x, y, z
Globals: print
BEGIN
          LOAD_CONST                    1
          STORE_FAST                    0
          LOAD_CONST                    2
          STORE_FAST                    1
          LOAD_FAST                     0
          LOAD_FAST                     1
          BINARY_ADD
          STORE_FAST                    2
          LOAD_GLOBAL                   0
          LOAD_FAST                     2
          CALL_FUNCTION                 1
          POP_TOP
```

```
            LOAD_CONST                          0
            RETURN_VALUE
END

11
MyComputer> coco addtwo.casm
11
MyComputer>
```

Перший запуск викликає коко, який збирає програму, що виробляє зібраний висновок, а потім запускає програму, що виробляє **11**, яка з'являється під зібраним висновком. Зібраний висновок показано, оскільки при виклику кокосу використовувався параметр **-v**. Зібраний вихід виводиться на потік, який називається стандартною помилкою, який відокремлений від стандартного вихідного потоку, де друкується **11**. Щоб надрукувати лише точний результат програми, параметр **-v** можна опустити.

У цій програмі JCoCo є одна функція, яка називається main. Збірка вказує, що **main** має 0 формальних параметрів.

Константи, які використовуються в коді, включають `None`, `5` і `6`. У функції є три локальні змінні: `x`, `y` та `z`. Викликається глобальна функція друку, як і у списку глобалів. Кожна функція в JCoCo має ці категорії ідентифікаторів та значень у межах кожної визначеної функції. Іноді одна або кілька з цих категорій можуть бути порожніми, і в цьому випадку їх можна опустити.

Інструкції слідують за ключовим словом **`begin`** і перед кінцевим ключовим словом. **`LOAD_CONST`** завантажує значення константи за своїм індексом (на основі нуля та **`1`** у цьому випадку) у константи у стек операндів. JCoCo - це машина стека, і тому всі операції виконуються з операндами, висунутими та вискоченими зі стеку операндів.

Інструкція **`STORE_FAST`** зберігає значення в списку місцевих жителів, у цьому випадку зі зміщенням **`0`**, розташування **`x`**. **`LOAD_FAST`** робить протилежність **`STORE_FAST`**, висуваючи значення в стеці операндів зі списку змінних локальних систем. **`BINARY_ADD`** виймає два операнди зі стеку та додає їх разом, підштовхуючи результат. **`CALL_FUNCTION`** виводить кількість аргументів, зазначених в інструкції (**`1`** - у цьому випадку), а потім виводить функцію зі стеку. Нарешті, він викликає функцію **`popped`** із аргументами **`popped`**.

Результат виклику функції залишається у верхній частині стеку операндів. У разі функції друку **None** повертається та залишається у стосі.
Інструкція **POP_TOP** виймає із стека значення **None** та відкидає його лише для того, щоб основна функція натискала **None** у стеці безпосередньо перед поверненням. **RETURN_VALUE** вискакує верхній аргумент зі стеку операндів і повертає це значення у функцію, що викликає. Оскільки main була єдиною викликаною функцією, повернення з неї закінчує інтерпретацію кокосової програми.

Щоб запустити цей код, ви повинні мати виконуваний файл **coco** десь на вашому шляху. Тоді ви можете виконати наступний код, щоб спробувати.

```
MyComputer> python3.2 addtwo.py 1 > addtwo.casm
MyComputer> coco addtwo.casm
```

Введення-виведення

JCoCo надає одну вбудовану функцію зчитування вводу з клавіатури та кілька функцій для запису виводу на цей екран або стандартний вихід. Наступна програма демонструє отримання введеної інформації з клавіатури та друк на стандартний вихід.

```
1  import disassembler
2  def main():
3      name = input("Enter your name: ")
4      age = int(input("Enter your age: "))
5      print(name + ", a year from now you will be", age+1, "years old.")
6  #main()
7  disassembler.disassemble(main)
```

У коді Python тут викликається функція введення. Виклик введення вимагає рядкового запиту і повертає рядок введеного вводу. Виклик функції **`int`** у рядку, як це робиться у рядку, який отримує вік від користувача, повертає цілочисельне представлення значення рядка. Нарешті, функція друку приймає випадкову кількість аргументів, перетворює кожен у рядок за допомогою магічного методу **`_ _str_ _`** і друкує кожен рядок, розділений пробілами. Перший аргумент для друку в коді. **`3.3`** - це результат об'єднання імені та рядка ", через рік ти будеш". Використовувалось об'єднання рядків, оскільки між значенням імені та комою не повинно бути пробілу.

Мова асемблера, що реалізує програму, наведена на рис. 4.2. Зверніть увагу, що вбудовані функції, такі як **`input`**, **`int`** та **`print`**, декларуються під глобальний список. **`name`** та **`age`** змінні — локальні.

Рис. 4.2
JCoCo I/O

```
Function: main/0
Constants: None, "Enter your name:
    "Enter your age: ",
    ", a year from now you will be
    1, "years old."
Locals: name, age
Globals: input, int, print
BEGIN
          LOAD_GLOBAL          0
          LOAD_CONST           1
          CALL_FUNCTION        1
          STORE_FAST           0
          LOAD_GLOBAL          1
          LOAD_GLOBAL          0
          LOAD_CONST           2
          CALL_FUNCTION        1
          CALL_FUNCTION        1
          STORE_FAST           1
          LOAD_GLOBAL          2
          LOAD_FAST            0
          LOAD_CONST           3
          BINARY_ADD
          LOAD_FAST            1
          LOAD_CONST           4
          BINARY_ADD
          LOAD_CONST           5
          CALL_FUNCTION        3
          POP_TOP
          LOAD_CONST           0
          RETURN_VALUE
END
```

Рядок 9 висуває функцію введення на стек операндів.
Рядок 10 штовхає рядок із запитом на введення.
Рядок 11 викликає функцію введення з одним дозволеним аргументом, який їй надано.
1 у рядку 11 - це кількість аргументів. Коли функція введення повертається, вона залишає рядок, введений користувачем у стек операндів.
Рядок 12 зберігає цю назву в локальних.

Рядок 13 готується перетворити наступний вхід у ціле число, спочатку натиснувши функцію **`int`** на стек операндів. Потім рядок 14 завантажує функцію введення.
Рядок 15 завантажує підказку, як рядок 10 раніше.
Рядок 16 викликає функцію введення. Результат негайно передається функції **`int`**, викликаючи її на рядок 17. Функція int залишає ціле число у верхній частині стеку операндів, а рядок 18 зберігає це в розташуванні змінної **`age`**.

Наступна частина програми друкує вихідні дані. Щоб підготуватися до виклику функції друку, спочатку слід оцінити аргументи, потім можна викликати друк.
Рядок 19 висуває функцію друку на стек, але не викликає друк. Існує три аргументи функції друку. Перший аргумент - це результат об’єднання двох рядків разом.
Рядок 20 висуває значення змінної імені у стек.
Рядок 21 штовхає рядок ”, через рік ти будеш” на стек.
Рядок 22 викликає магічний метод `__add__` для об'єднання двох рядків. `BINARY_ADD` інструкція витягує два операнди зі стеку, викликає метод `__add__` для першого об'єкта, що з'являється з другим об'єктом як аргументи.

Рядки 23–25 складають вік та 1, щоб отримати правильне значення віку, яке потрібно передати для друку.
Рядок 26 штовхає останню константу рядка в стеку операндів, а рядок 27 нарешті викликає функцію друку, залишаючи згодом **`None`** у стеці операндів.
У рядку 28 вискакує значення **`None`**, і негайно **`None`** повертається назад у стек у рядку 29, оскільки головна функція повертає **`None`** у цьому випадку, який повертається у рядок 30, закінчуючи виконання програми **`iotest.casm`**.

Кілька важливих речей, які слід запам’ятати з цієї лекції:

- Отримання введення та виведення результату покладаються на вбудовані функції введення та друку.
- Перш ніж функцію можна викликати, її потрібно натиснути на стек операндів. Усі необхідні аргументи функції також повинні бути перенесені у стек поверх функції, яку потрібно викликати.
- Нарешті, коли функція повертається, вона залишає своє значення повернення в стеку операндів.

На наступній лекції розглянемо методи розгалудження обчислювального процесу.