ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 6

Мова Ассеблера (закінчення)

Весна 2021

Діапазонні об’єкти і ледачі оцінки

Індексація в послідовності - це ще один спосіб ітерації в програмі. Коли ви індексуєте список, ви використовуєте індекс для отримання елемента списку. Як правило, індекси базуються на нулі. Отже, перший елемент послідовності має індекс 0, другий - індекс 1 тощо.

Сьогодні використовується дві версії Python. Версія 2, хоча стара версія все ще широко використовується, оскільки існує багато програм Python, написаних з її допомогою, і їх перетворення на використання Python 3 вимагає великих витрат. Python 3 був створений, щоб нові функції могли додати, що може бути несумісним зі старою версією. Одна різниця полягала у функції діапазону. У Python 2 функція діапазону генерувала список цілих чисел зазначеного розміру та значень. Це неефективно, оскільки деякі діапазони можуть складатися з мільйонів цілих чисел. Мільйон цілих чисел займає багато місця в пам'яті і займає певний час для його створення.

Крім того, залежно від того, як пишеться код, можуть знадобитися не всі цілі числа в діапазоні. Ці проблеми є результатом пильної оцінки функції діапазону. Ретельне оцінювання - це коли вся послідовність генерується до того, як фактично буде використаний будь-який елемент послідовності. У Python 2 весь список цілих чисел створюється, як тільки викликається функція діапазону, хоча код може використовувати лише одне ціле число за раз.

Python 3 вирішив нетерплячу оцінку функції діапазону, визначивши об'єкт діапазону, який ліниво оцінюється. Це означає, що коли ви викликаєте функцію діапазону для генерації мільйона цілих чисел, ви не отримуєте жодного з них відразу. Натомість ви отримуєте об’єкт діапазону. З об'єкта діапазону ви можете отримати доступ до ітератора. Коли на ітераторі викликається _ _next_ _, ви отримуєте наступний елемент у послідовності. Коли _ _next_ _ викликається на ітераторі об'єкта діапазону, ви отримуєте наступне ціле число в послідовності діапазону.

Ледача оцінка - це коли наступне значення в послідовності генерується лише тоді, коли воно готове до використання, а не раніше. Цей код створює *об'єкт діапазону*. Об’єкт діапазону призначений для надання лінивої оцінки цілочисельних послідовностей.

```
1  from disassembler import *
2  def main():
3      x = input("Enter list: ")
4      lst = x.split()
5      for i in range(len(lst)-1,-1,-1):
6          print(lst[i])
7  disassemble(main)
```

Цей код Python використовує індекси для ітерації списку назад. У цьому випадку ітератор над об'єктом діапазону видає спадний список цілих чисел, які є індексами до списку значень, введених користувачем.

Якщо використання вводить чотири значення, розділені пробілом, тоді об’єкт діапазону видасть послідовність [3, 2, 1, 0]. Перший аргумент діапазону - це початкове значення, другий - значення, яке минуло стоп-значення, а третій аргумент - збільшення. Отже, послідовність у коді Python на слайді 7 - низхідна послідовність, яка зменшується на одне ціле число за раз від довжини списку мінус один до нуля.

```
Function: main/0
Constants: None,"Enter list: ",1,-1,-1
Locals: x, lst, i
Globals: input,split,range,len,print
BEGIN
            LOAD_GLOBAL         0
            LOAD_CONST          1
            CALL_FUNCTION       1
            STORE_FAST          0
            LOAD_FAST           0
            LOAD_ATTR           1
            CALL_FUNCTION       0
            STORE_FAST          1
            SETUP_LOOP label02
            LOAD_GLOBAL         2
            LOAD_GLOBAL         3
            LOAD_FAST           1
            CALL_FUNCTION       1
            LOAD_CONST          2
            BINARY_SUBTRACT
            LOAD_CONST          3
            LOAD_CONST          4
            CALL_FUNCTION       3
            GET_ITER
label00:    FOR_ITER label01
            STORE_FAST          2
            LOAD_GLOBAL         4
            LOAD_FAST           1
            LOAD_FAST           2
            BINARY_SUBSCR
            CALL_FUNCTION       1
            POP_TOP
            JUMP_ABSOLUTE label00
label01:    POP_BLOCK
label02:    LOAD_CONST          0
            RETURN_VALUE
END
```

Рис.6.1. Збірка діапазону

Код збірки JCoCo на рис. 6.1 реалізує цю саму програму. Рядки 15–23, налаштовані для виклику функції діапазону з трьома цілими значеннями. Рядки 15–20 викликають функцію **`len`**, щоб отримати довжину списку і відняти одиницю. Рядки 21 і 22 ставлять два **`-1`** значення у стек операндів. Рядок 23 викликає функцію діапазону, яка створює і підштовхує об'єкт діапазону до стеку операндів як його результат.

Рядок 24 створює ітератор для об'єкта діапазону. Як описано в останньому розділі, інструкція **`FOR_ITER`** викликає магічний метод **`_ _next_ _`** на ітераторі, щоб отримати наступне ціле число в послідовності діапазону. Ледаче оцінювання відбувається тому, що ітератор відстежує, яке ціле число є наступним значенням у послідовності. У рядку 26 зберігається наступне ціле число у змінній **`i`**.

Інструкція **`BINARY_SUBSCR`** - це інструкція, яка ще не зустрічалась у цьому розділі. Рядок 28 завантажує список, що називається **`lst`**, у стек операндів. Рядок 29 завантажує значення **`i`** у стек операндів. Інструкція **`BINARY_SUBSCR`** індексується в **`lst`** в позиції **`i`** і висуває значення, знайдене в цій позиції, у стек операндів. Це значення друкується за допомогою виклику функції друку в рядку 31 програми.

Ледаче оцінювання - важливе поняття мови програмування. Якщо ви коли-небудь стикнетеся з написанням коду, який повинен генерувати передбачувану послідовність значень, ви, ймовірно, захочете сформувати цю послідовність ліниво. Ітератори, як і ітератори діапазону, - це засоби, за допомогою яких ми можемо ліниво отримувати доступ до послідовності значень, а об’єкти діапазону визначають послідовність цілих чисел, не бажаючи генерувати їх усіх.

Функції і замикання

До цього моменту в останніх лекціях всі приклади програм були визначені *в основній функції*. JCoCo підтримує визначення багатьох функцій і навіть вкладених функцій. Ось програма Python, яка демонструє, як писати вкладені функції на мові програмування Python. Основна функція викликає функцію з іменем **`f`**, яка повертає функцію **`g`**, вкладену всередину функції **`f`**. Функція **`g`** повертає **`x`**. Ця програма демонструє вкладені функції в JCoCo, а також способи побудови Замикання.

```
1 import disassembler
2 def main():
3   x = 10
4   def f(x):
5     def g():
6       return x
7     return g
8   print(f(3)())
9 disassembler.disassemble(main)
```

Зверніть увагу, що код Python у слайді 15 викликає функцію дізасемблювання на функції верхнього рівня **`main`**. Це не викликається на **`f`** або **`g`**, оскільки вони вкладені всередину **`main`**, і розбирач автоматично розбирає будь-які вкладені функції розібраної функції.

Варто також зазначити формат відповідної програми JCoCo на рис. 6.2. Основна функція верхнього рівня визначена вздовж лівої сторони. Відступ не впливає на JCoCo, але візуально ви бачите, що **`f`** вкладено всередину **`main`**. Функція **`g`** вкладена всередину **`f`**, оскільки вона з'являється відразу після першого рядка визначення **`f`** у рядку 3. Решта визначення **`f`** починається знову з рядка 10 і продовжується до рядка 21. Визначення **`g`** починається з рядка 3 і поширюється на рядок 9.

```
Function: main/0
    Function: f/1
        Function: g/0
        Constants: None
        FreeVars: x
        BEGIN
                    LOAD_DEREF      0
                    RETURN_VALUE
        END
    Constants: None, code(g)
    Locals: x, g
    CellVars: x
    BEGIN
              LOAD_CLOSURE      0
              BUILD_TUPLE       1
              LOAD_CONST        1
              MAKE_CLOSURE      0
              STORE_FAST        1
              LOAD_FAST         1
              RETURN_VALUE
    END
```

```
Constants: None, 10, code(f), 3
Locals: x, f
Globals: print
BEGIN
          LOAD_CONST            1
          STORE_FAST            0
          LOAD_CONST            2
          MAKE_FUNCTION         0
          STORE_FAST            1
          LOAD_GLOBAL           0
          LOAD_FAST             1
          LOAD_CONST            3
          CALL_FUNCTION         1
          CALL_FUNCTION         0
          CALL_FUNCTION         1
          POP_TOP
          LOAD_CONST            0
          RETURN_VALUE
END
```

Рис.6.2.

Кількість аргументів для кожної функції задається цілим числом після скісної риски. **`f/1`** означає, що **`f`** очікує одного аргументу. Основна та **`g`**-функції очікують нульових аргументів. Ці значення використовуються під час виклику функції для перевірки того, що функція викликається з необхідною кількістю аргументів.

Уважно вивчіть код Python на слайді 15. Основна функція викликає функцію **f**, яка повертає функцію **g**. Зверніть увагу, що **f** повертає **g**, воно не викликає **g**. У операторі **print main** викликається функція **f**, передаючи 3 функції, яка повертає **g**. Додатковий набір пар після виклику функції **f** (3) **g**. Це дійсно

Програма Python, але не поширена. Питання: Що друкує програма? Здається, є два можливі варіанти: або **10**, або **3**. Що здається більш імовірним?

З одного боку, **g** викликається з основної функції, де **x** дорівнює **10**. Якщо програма надрукувала **10**, ми сказали б, що Python - це мова з динамічним масштабом, що означає, що функція виконується в середовищі, в якому вона викликається . Оскільки **g** викликається з **main**, значення **x** дорівнює **10**, і мовою з динамічним масштабом буде надруковано **10**. Слово динамічний використовується, оскільки якщо **g** було викликано в іншому середовищі, це може повернути щось зовсім інше. Ми можемо визначити лише те, що поверне **g**, простеживши виконання програми до точки, де викликається **g**.

З іншого боку, `g` було визначено у діапазоні `x`, значення якого було `3`. У цьому випадку середовищем, в якому виконується `g`, є середовище, передбачене `f`. Якщо друкується `3`, то Python - це мова зі статичним масштабом, що означає, що нам потрібно лише зрозуміти, що містило середовище, коли було визначено `g`, а не коли його викликали. У мові зі статичним масштабом цей конкретний екземпляр `g` повертатиме одне й те саме значення кожного разу, коли його викликають, неважливо, куди його викликають у програмі.

Значення **x** визначається, коли визначається **g**. Мови з динамічним масштабом зустрічаються рідко. Коли його вперше було визначено, його динамічно масштабували. Маккарті швидко виправив це і зробив Ліспа мовою зі статичним масштабом. Цікаво відзначити, що Emacs Lisp має динамічний масштаб.

Python має статичний обсяг, як і більшість сучасних мов програмування. Для виконання функцій мовою зі статичним масштабом потрібні дві частини, коли функція може повернути іншу функцію. Для виконання **`g`** потрібен не тільки код **`g`**, але й середовище, в якому було визначено цей екземпляр **`g`**. Утворюється Замикання. Замикання - це середовище, в якому визначена функція, та код самої функції. Це Замикання називається тим, коли функція **`g`** нарешті викликається в **`main`**.

У програмуванні, **замиканням** (англ. closure) називають підпрограму, що виконується в середовищі, яке містить одну або більше зв'язаних змінних. Підпрограма має доступ до цих змінних під час виконання.

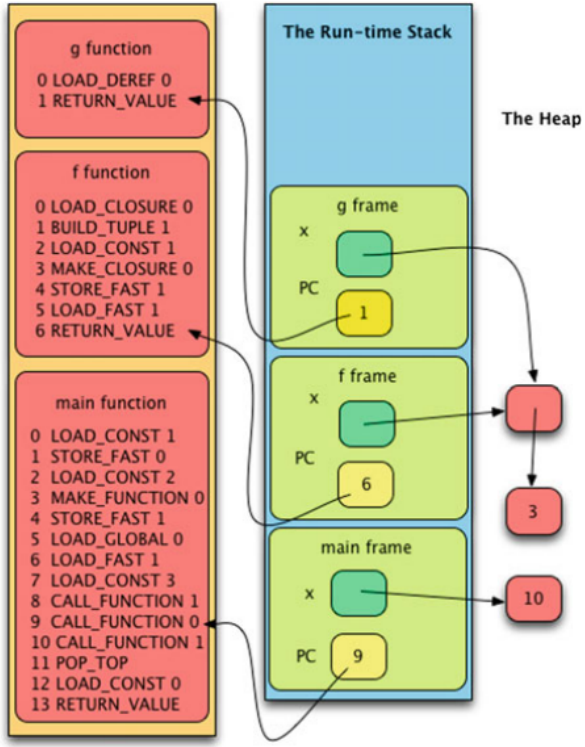
Застосування замикань асоціюється з функціональним програмуванням. У функціональному програмуванні за допомогою замикань можуть моделюватись такі конструкції, як об'єкти в інших мовах програмування.

Погляньте код JCoCo для цієї програми на рис. 6.2. Рядок 14 починає створення нового об'єкта Замикання в тілі функції **`f`** шляхом завантаження змінної комірки з іменем **`x`** на стек. Змінна комірки - це непряме посилання на значення. На рисунку 6.3 показано, що відбувається в програмі безпосередньо перед тим, як **`x`** повертається у функції **`g`**. Змінна в Python, як Java та багато інших мов, насправді є посиланням, яке вказує на значення. Значення існують у купі та створюються динамічно під час виконання програми. Коли змінної присвоюється нове значення, робиться посилання на змінні, щоб вказати на нове значення в купі.

Простір для значень у купі, які більше не потрібні, відновлюється збирачем сміття, який звільняє простір у купі, щоб його можна було використовувати повторно. На рис. 6.3 у купі є три значення, 10, 3 і ще одне значення, яке називається комірка в JCoCo та віртуальна машина Python.

Оскільки функція **`g`** потребує доступу до змінної **`x`** поза функцією **`f`**, на **`3`** опосередковано посилається через змінну комірки. Інструкція **`LOAD_CLOSURE`** штовхає цю змінну комірки у стек, який буде використаний у закритті. Оскільки для середовища потрібно лише одне значення, наступна інструкція в рядку 15 будує кортеж усіх значень, необхідних для середовища. Рядок 16 завантажує код для **`g`** у стек. Рядок 17 формує Замикання, вискакуючи функцію та середовище зі стеку та будуючи об’єкт Замикання.

Рис.3.3 Виконання **nested.casm**



Оскільки функція **`g`** потребує доступу до змінної **`x`** поза функцією **`f`**, на **`3`** опосередковано посилається через змінну комірки. Інструкція **`LOAD_CLOSURE`** штовхає цю змінну комірки у стек, який буде використаний у закритті. Оскільки для середовища потрібно лише одне значення, наступна інструкція в рядку 15 будує кортеж усіх значень, необхідних для середовища. Рядок 16 завантажує код для **`g`** у стек. Рядок 17 формує Замикання, вискакуючи функцію та середовище зі стеку та будуючи об’єкт Замикання.

Змінна **`x`** є локальною змінною для функції **`f`**. Але оскільки **`x`** посилається на **`g`**, а **`g`** вкладено всередину **`f`**, змінна **`x`** також вказана як змінна комірки в **`f`**. Змінна комірки - це непряме посилання на значення. Це означає, що є один додатковий крок до пошуку значення, на яке посилається **`x`**. Ми повинні пройти камеру, щоб дістатися до **`3`**.

Інструкція **LOAD_DEREF** у рядку 7 є новою. **LOAD_DEREF** завантажує значення, на яке посилається посилання, на яке вказано у списку комірок. Отже, ці інструкції висувають **3** на стек операндів. Нарешті, рядок 35 викликає Замикання, що складається з функції та її даних. У функції g фрівари посилаються на набір посилань у закритті, яке було щойно викликане, тому перша інструкція, **LOAD_DEREF**, завантажує **3** в стек операндів. Малюнок 6.3 зображує цей стан безпосередньо перед тим, як є страченою інструкція **RETURN_VALUE**.

Для завершення виконання цієї програми `3` повертається із виклику до `g`, а її кадр вискакує із стеку під час виконання. Елемент керування повертається до основного, де друкується `3`. Після повернення з основного його кадр також вискакує зі стеку під час виконання, який закінчує програму.

Рекурсія

Функції в JCoCo можуть викликати себе. Функція, яка викликає себе, є рекурсивною функцією. Рекурсивні функції докладно будуть розглянуті в лекціях імплементації функціонального програмування. Навчитися добре писати рекурсивні функції не складно, якщо дотримуватись деяких основних правил. Механіка написання рекурсивної функції включає надання базового випадку, який стоїть на першому місці у функції. Потім вирішення проблеми, яку ви вирішуєте, має бути вирішене шляхом виклику тієї самої функції на деякому меншому фрагменті даних, використовуючи цей результат для побудови рішення більшої проблеми.

Розглянемо факторіальне визначення. Факториал нуля, записаний `0!`, визначається рівним `1`. Це базовий випадок. Для цілого числа `n` більше `0`, `n! = n * (n - 1)!`. Це рекурсивне визначення, оскільки факторіал визначається як сам по собі. Він називається на чомусь меншому, тобто `n - 1`, що ближче до базового випадку, і результат використовується при обчисленні `n!`. Ось програма Python, яка обчислює `5!`.

```
1  import disassembler
2  def factorial(n):
3      if n==0:
4          return 1
5      return n*factorial(n-1)
6  def main():
7      print(factorial(5))
8
9  disassembler.disassemble(factorial)
10 disassembler.disassemble(main)
```

Реалізація цієї програми JCoCo наведена на рис. 6.4. Програма починається головним чином із завантаження `5` у стек операндів та виклику факторіальної функції. Результат друкується на екрані за допомогою функції друку.

```
Function: factorial/1
Constants: None, 0, 1
Locals: n
Globals: factorial
BEGIN
            LOAD_FAST           0
            LOAD_CONST          1
            COMPARE_OP          2
            POP_JUMP_IF_FALSE label00
            LOAD_CONST          2
            RETURN_VALUE
label00:    LOAD_FAST           0
            LOAD_GLOBAL         0
            LOAD_FAST           0
            LOAD_CONST          2
            BINARY_SUBTRACT
            CALL_FUNCTION       1
            BINARY_MULTIPLY
            RETURN_VALUE
END
Function: main/0
Constants: None, 5
Globals: print, factorial
BEGIN
            LOAD_GLOBAL         0
            LOAD_GLOBAL         1
            LOAD_CONST          1
            CALL_FUNCTION       1
            CALL_FUNCTION       1
            POP_TOP
            LOAD_CONST          0
            RETURN_VALUE
END
```

Рис.6.4 Асемблювання рекурсії

Реалізація цієї програми JCoCo наведена на рис. 6.4. Програма починається головним чином із завантаження `5` у стек операндів та виклику факторіальної функції. Результат друкується на екрані за допомогою функції друку.

Виклик факторіального переходу до першої інструкції функції, де **n** завантажується в стек операндів, що в цьому випадку дорівнює **5**. Рядки 7–8 порівнюють **n** з **0**, і якщо два значення рівні, повертається **1**. Зверніть увагу, що в цьому випадку інструкція **RETURN_VALUE** відображається посередині факторіальної функції. Інструкція повернення не повинна з'являтися в кінці функції. Він може з’являтися де завгодно, і в цьому випадку має сенс якомога швидше повернутися з базової справи.

Код із **`label00 forward`** - це рекурсивний випадок, оскільки в іншому випадку ми б уже повернулися. Код віднімає одиницю з **`n`** і викликає факторіал із новим, меншим значенням. Зверніть увагу, що рекурсивний виклик функції ідентичний будь-якому іншому виклику функції. Нарешті, після виклику функції результат виклику знаходиться на операнді стека, і він множиться на **`n`**, щоб отримати **`n!`** який повертається.

Оскільки це рекурсивна функція, попередні два абзаци повторюються ще 5 разів, кожен раз зменшуючи **`n`** на **`1`**. Програма продовжує відлік поки **`1`** не повертається для факторіалу **`0`**. Найглибшими є 7 кадрів стека для стеку часу виконання для цієї програми: один для основної функції та ще шість для рекурсивних викликів факторіальних функцій. При виконанні базового випадку стек часу виконання зростає до 7 кадрів стека, а потім знову стискається, коли рекурсія розгортається. Нарешті, коли програма повертається до основної, **`120`** виводиться на екран.

Підтримка класів та об’єктів

У Python визначення класу складається з оголошення класу. Оголошення класу містить магічний метод **`_ _init_ _`**, який відповідає за ініціалізацію будь-якого створеного об'єкта. У рядку 14 цієї програми Python створюється екземпляр об'єкта. Python автоматично викликає конструктор для ініціалізації простору, виділеного Python для об'єкта.

```
1  import disassembler
2  class Dog:
3      def __init__(self):
4          self.food = 0
5      def eat(self):
6          self.food = self.food + 1
7      def speak(self):
8          if self.food > 2:
9              print("I am happy!")
10         else:
11             print("I am hungry!!!")
12         self.food=self.food - 1
13 def main():
14     mesa  = Dog()
15     mesa.eat()
16     mesa.speak()
17     mesa.eat()
18     mesa.eat()
19     mesa.speak()
20 disassembler.disassemble(Dog)
21 disassembler.disassemble(main)
```

Клас - це сукупність функцій, які всі функціонують на певній групі даних. Наприклад, клас **`Dog`** містить функцію, яка називається *їсти*, а інша називається *говорити*. Функції *їсти* та *говорити* діють на об'єкти типу **`Dog`**.

Функції визначення класу часто називають методами, щоб відрізнити їх від самостійних функцій. Методи отримують посилання на поточний об’єкт, який є колекцією даних, над якими працює метод. Посилання **`self`** використовується для посилання на поточний об'єкт і завжди є першим параметром методу в Python. Цей опис методів не зовсім точний при розгляді Віртуальної машини Python.

У віртуальній машині Python методи створюються з функцій класу при екземплярі об’єкта. Розглянемо програму збірки на рис. 6.5, яка демонструє створення класу під назвою **Dog**. У рядку 36 основної функції викликається клас **Dog**. Виклик класу в Python означає виконання коду, який виділяє об'єкт **Dog** у пам'ять. Цією роботою займається віртуальна машина.
Програма мови асемблер не виділяє безпосередньо простір для об'єкта за допомогою будь-якої інструкції. Це здійснюється за допомогою виклику класу.
Усі об'єкти в Python складаються зі словника, що зберігає атрибути об'єкта.

Коли викликається клас **Dog**, словник в об'єкті **Dog** ініціалізується за допомогою об'єктів **Dog**. Методи об'єкта **Dog** по суті є функціями класу **Dog**. Різниця між методом і функцією полягає в самопараметрі.

Метод забезпечує аргумент **self** для своєї функції, надаючи посилання на поточний об'єкт як перший параметр. Метод є обгорткою для функції. Після збереження методів у словнику об’єкта викликається метод **_ _init_ _** для подальшої ініціалізації об’єкта.

Щоб краще зрозуміти різницю між методами та функціями, розглянемо код, який викликає метод **`eat`**. Коли викликається метод eat для об'єкта **`Dog`**, метод надає посилання на поточний об'єкт **`Dog`** як перший параметр перед викликом функції їжі класу **`Dog`**. Це виглядає, починаючи з рядка 38 на рис. 6.5. Рядок 38 завантажує посилання для мези на стек операндів.

Потім рядок 39 шукає метод eat у словнику об’єкта, залишаючи метод поверх стеку операндів, але не посилання на об’єкт **`Dog`**. Також зверніть увагу, що поверх методу не завантажуються аргументи. Проте функція їсти має один параметр, **`self`**. Коли метод "**`eat`**" викликається на рядку 40, віртуальна машина завантажує параметр **`self`** перед викликом функції **`eat`**. Розрізнення між методами та функціями ще раз переглянуто в наступному розділі.

Рис.6.5
Клас Dog

```
Class: Dog
BEGIN
    Function: eat/1
    Constants: None, 1
    Locals: self
    Globals: food
    BEGIN
                LOAD_FAST      0
                LOAD_ATTR      0
                LOAD_CONST     1
                BINARY_ADD
                LOAD_FAST      0
                STORE_ATTR     0
                LOAD_CONST     0
                RETURN_VALUE
    END
    Function: __init__/1
    Constants: None, 0
    Locals: self
    Globals: food
    BEGIN
                LOAD_CONST   1
                LOAD_FAST    0
                STORE_ATTR   0
                LOAD_CONST   0
                RETURN_VALUE
    END
    # speak function omitted
END
Function: main/0
Constants: None
Locals: mesa
Globals: Dog, eat, speak
BEGIN
            LOAD_GLOBAL    0
            CALL_FUNCTION  0
            STORE_FAST     0
            LOAD_FAST      0
            LOAD_ATTR      1
            CALL_FUNCTION  0
            POP_TOP
            ...
            RETURN_VALUE
END
```

Успадкування

В об'єктно-орієнтованому програмуванні успадкування починає діяти, коли один клас успадковує від іншого. Спадщина корисна для поліморфізму та повторного використання коду. При програмуванні за допомогою Python поліморфізм відбувається без успадкування, оскільки Python - це динамічно набрана мова, тобто всі виклики методів розглядаються під час виконання, як це було показано в останньому розділі, коли виконувалася інструкція **`LOAD_ATTR`**. Інструкція **`LOAD_ATTR`** шукає метод за іменем у словнику об’єкта.

Пошук під час виконання методів за назвою створює поліморфну поведінку Python. Єдина мета успадкування в Python - повторне використання коду. У наступному розділі буде докладніше про поліморфізм та те, як це застосовується до програмування на Java, де успадкування потрібно для реалізації поліморфізму.

Розглянемо програму Python нижче. Знову є клас Dog, який цього разу успадковується від класу Animal. Клас Animal визначає метод їжі, який повторно використовується класом Dog. Конструктор Animal також містить код, який повторно використовується класом Dog. Але клас Dog визначає власний метод говоріння, замінюючи метод говоріння у класі Animal.

Спадщина Animal, що сприяє класу Dog, позначається написанням Dog (Animal) у рядку 12. Виклик super у рядку 14 повертає екземпляр об'єкта супер-класу, який може бути використаний для посилання на супер-клас, у цьому випадку Клас Animal. Програми Python можуть використовувати багаторазове успадкування. У JCoCo це неправда. Наразі підтримується лише одиночне успадкування.

```
1  import disassembler
2
3  class Animal:
4      def __init__(self,name):
5          self.name = name
6          self.food = 0
7      def eat(self):
8          self.food = self.food + 1
9      def speak(self):
10         print(self.name, "is an animal")
11
12 class Dog(Animal):
13     def __init__(self,name):
14         super().__init__(name)
15     def speak(self):
16         print(self.name, "says woof!")
17
18 def main():
19     mesa = Dog("Mesa")
20     mesa.eat()
21     mesa.speak()
22
23 disassembler.disassemble(Animal)
24 disassembler.disassemble(Dog)
25 disassembler.disassemble(main)
```

```
Class: Animal
BEGIN
    Function: __init__/2
    Constants: None, 0
    Locals: self, name
    Globals: name, food
    BEGIN
                LOAD_FAST    1
                LOAD_FAST    0
                STORE_ATTR   0
                LOAD_CONST   1
                LOAD_FAST    0
                STORE_ATTR   1
                LOAD_CONST   0
                RETURN_VALUE
    END
    Function: eat/1
    Constants: None, 1
    Locals: self
    Globals: food
    BEGIN
                LOAD_FAST  0
                ...
                RETURN_VALUE
    END
    Function: speak/1
    Constants: None, "is an animal"
    Locals: self
    Globals: print, name
    BEGIN
                LOAD_GLOBAL  0
                ...
                RETURN_VALUE
    END
END
```

Рис.6.6 Наслідування в JcoCo (1)

Рис.6.7 Наслідування в JcoCo (2)

```
Class: Dog(Animal)
BEGIN
    Function: __init__/2
    Constants: None
    Locals: self, name
    FreeVars: __class__
    Globals: super, __init__
    BEGIN
                LOAD_GLOBAL       0
                CALL_FUNCTION   0
                LOAD_ATTR   1
                LOAD_FAST     1
                CALL_FUNCTION     1
                POP_TOP
                LOAD_CONST 0
                RETURN_VALUE
    END
    Function: speak/1
    Constants: None, "says woof!"
    Locals: self
    Globals: print, name
    BEGIN
                LOAD_GLOBAL    0
                ...
                RETURN_VALUE
    END
END
Function: main/0
Constants: None, "Mesa"
Locals: mesa
Globals: Dog, eat, speak
BEGIN
            LOAD_GLOBAL   0
            LOAD_CONST    1
            CALL_FUNCTION   1
            STORE_FAST    0
            ...
            RETURN_VALUE
END
```

Динамічно створювані класична

Попередній розділ демонструє оголошення класу та створення об’єктів мовою асемблера JCoCo. Також можна створити клас динамічно, під час виконання. Це також дозволяє за бажанням визначати змінні класу. Змінна класу є змінною присвоюється класу замість екземплярів класу (тобто об'єктів). Розглянемо цю програму Python, де `dogNumber` - це змінна класу, яка може використовуватися для підрахунку кількості екземплярів створеного об'єкта. Створення цього класу вимагає виконання додаткового коду під час побудови класу. Змінна класу повинна бути ініціалізована значенням `0`.

Код збірки для цієї програми виглядає дещо інакше, ніж у попередньому розділі. Замість того, щоб бачити клас Dog, існує функція Dog. Функція Dog стає класом Dog в результаті її виконання. Щоб пояснити, що відбувається, почнемо з рис. 6.9 який містить код основної функції. Рядок 55 цього коду містить інструкцію LOAD_BUILD_CLASS. Ця інструкція завантажує вбудовану функцію, яка будує клас із двох аргументів. Замикання - один з аргументів. Як було зазначено в розд. 3.11, Замикання - це код та середовище. Ми ще раз відвідаємо це в розділах 4 і 6.

Замикання - це і код, і середовище (тобто колекція змінних), в якому код повинен виконуватися. Замикання в цьому прикладі відповідає за створення вмісту класу, включаючи його змінну класу та методи класу, які, якщо згадати з попереднього розділу цього розділу, насправді є функціями, доки об’єкт не буде створений.

Іншим аргументом вбудованої функції конструктора класів є назва класу. Рядок 56 створює Замикання. Лінія 57 створює кортеж із Замикання. У рядку 58 завантажується код для ініціалізації класу (тобто код для функції Dog). Рядок 59 будує Замикання за допомогою кортежу та коду. Рядок 60 завантажує ім'я класу в стек операндів. Потім рядок 61 викликає вбудовану функцію конструктора класів, передаючи їй Замикання та ім'я класу.

Потім вбудована функція конструктора класів виконує певне ведення домашнього господарства, створюючи екземпляр класу, називаючи його Dog, оскільки це було передано як ім'я класу, і викликає функцію Dog для завершення створення екземпляра класу. Це переходить до коду в рядку 33 на рис. 6.8.

Рис.6.8
Динамічно створювані класи (1)

```
Function: main/0
    Function: Dog/1
        Function: __init__/2
        Constants: None, 1
        Locals: self, name
        FreeVars: Dog
        Globals: name, dogNumber, id
        BEGIN
                    LOAD_FAST                   1
                    LOAD_FAST                   0
                    STORE_ATTR                  0
                    LOAD_DEREF                  0
                    LOAD_ATTR                   1
                    LOAD_FAST                   0
                    STORE_ATTR                  2
                    ...
                    RETURN_VALUE
        END
        Function: speak/1
        Constants: None, "Dog number: "
        Locals: self
        Globals: print, id
        BEGIN
                    LOAD_GLOBAL                 0
                    ...
                    RETURN_VALUE
        END
    Constants: 0, code(__init__), code(speak), None
    Locals: __locals__
    FreeVars: Dog
    Globals: __name__, __module__, dogNumber, __init__, speak
    BEGIN
              LOAD_FAST                     0
              STORE_LOCALS
              LOAD_NAME                     0
              STORE_NAME                    1
              LOAD_CONST                    0
              STORE_NAME                    2
              LOAD_CLOSURE                  0
              BUILD_TUPLE                   1
              LOAD_CONST                    1
              MAKE_CLOSURE                  0
              STORE_NAME                    3
              LOAD_CONST                    2
              MAKE_FUNCTION                 0
              STORE_NAME                    4
              LOAD_CONST                    3
              RETURN_VALUE
    END
```

Рис.6.8
Динамічно створювані класи (2)

```
50  Constants: None, code(Dog), "Dog", "Mesa", "Sequoia"
51  Locals: x, y
52  CellVars: Dog
53  Globals: speak
54  BEGIN
55              LOAD_BUILD_CLASS
56              LOAD_CLOSURE                    0
57              BUILD_TUPLE                     1
58              LOAD_CONST                      1
59              MAKE_CLOSURE                    0
60              LOAD_CONST                      2
61              CALL_FUNCTION                   2
62              STORE_DEREF                     0
63              LOAD_DEREF                      0
64              LOAD_CONST                      3
65              CALL_FUNCTION                   1
66              STORE_FAST                      0
67              ...
68              RETURN_VALUE
69  END
```

Інструкція STORE_LOCALS у рядку 34 заслуговує певного пояснення. Локальні змінні функції Dog - це словник або карта зі рядків (тобто імен змінних) до їх значень. Коли вбудована функція конструктора класів викликає функцію Dog для завершення побудови класу, вона передає у функцію словник класу. Цей словник є словником класу Dog, і, виконуючи інструкцію STORE_LOCALS, словник також стає словником місцевих жителів для функції Dog.

Отже, все, що зберігається в локальних змінних, потім буде зберігатися в екземплярі класу. Це спільне використання словника локальних змінних та словника класів спрощує побудову класу, роблячи будь-які змінні, що зберігаються у функції Dog, також з іменованими змінними, включаючи іменовані методи, в екземплярі класу.

У рядку 35 на рис. 6.8 зберігається Dog як назва модуля. Вбудована функція побудови класу атрибуту _ _name_ _ вже встановлена на Dog. Отже, рядок 36 дає таку ж назву атрибуту _ _module_ _. Рядки 37 та 38 ініціалізують змінну класу dogNumber до 0. Рядок 39 починає роботу з додавання функцій до екземпляра класу, який створюватиметься до методів при створенні екземпляра Dog. Першою функцією, яку слід зберегти, є конструктор _ _init_ _, який відбувається в рядках 39–43. Рядки 44–46 зберігають функцію мовлення у класі.

Причиною того, що конструктор займає трохи більше роботи, є те, що він посилається і збільшує змінну класу dogNumber, і тому конструктору потрібні як код, так і середовище для правильного виконання. Метод Speak не посилається на будь-які змінні класу, тому середовище не потрібно. Інструкція MAKE_FUNCTION створює Замикання з порожнім середовищем.

Хоча класи можуть бути побудовані як динамічно (тобто під час виконання), так і статично, використовуючи синтаксис мови асемблера, розбірник буде використовувати динамічне розподіл, коли середовище використовується в одному з методів екземпляра класу. Використання середовища вимагає Замикання, і Замикання можуть бути побудовані під час динамічного розподілу класу.

Висновки щодо JCoCo

Розуміння мови асемблера є ключовим для вивчення того, як працюють мови програмування вищого рівня. Цей розділ представив програмування на мові асемблера на ряді прикладів, проводячи паралелі між Python та Python віртуальною машиною або інструкціями JCoCo. Використання розбірника було ключовим для отримання цього розуміння і є чудовим інструментом для використання з будь-якою платформою.

Більшість ключових конструкцій мов програмування були представлені як програми Python, так і програми JCoCo. Розділ, завершений висвітленням занять, успадкування та створення динамічного класу.
Мова асамблера, що висвітлена в останніх лекціях, буде використовуватися і в подальших лекціях.

JCoCo - це асемблер / віртуальна машина для інструкцій віртуальної машини Python. Звичайно, є й інші мови збірки. MIPS - це архітектура центрального процесора, яка має широку підтримку написання програм асемблерної мови, включаючи симулятор MIPS, який називається SPIM. Насправді асемблери доступні практично для будь-якої комбінації апаратного забезпечення та операційної системи, яка використовується сьогодні. Intel / Linux, Intel / Windows, Intel / Mac OS X всі підтримують програмування на мові асемблера. Віртуальну машину Java можна запрограмувати за вказівками JVM за допомогою Java-асемблера під назвою Jasmin.

Мова асамблера є основною мовою, яку використовують усі мови програмування вищого рівня у своїх реалізаціях.

На наступній лекції почнемо розгляд імплементації об’єкто-орієнтованого програмування.