Assembly Language

3

Python is an object-oriented, interpreted language. Internally to the Python interpreter, a Python program is converted to bytecode and interpreted using a virtual machine. Most modern programming languages have support for high-level abstractions while the instructions of a virtual machine are closer to the machine language instructions supported by hardware architectures, making the interpretation of bytecode easier than interpretation of the original source program. The advantage of virtual machine implementations results from dividing the mapping from high-level abstractions to low-level machine instructions into two parts: high-level abstractions to bytecode and bytecode to machine instructions.

While bytecode is a higher level abstraction than machine language, it is not greatly so. As programmers, if we understand how the underlying machine executes our programs, we better equip ourselves to make good choices about how we program. Just as importantly, having an understanding of how programs are executed can help us diagnose problems when things go wrong.

This chapter introduces assembly language programming in the bytecode language of the Python virtual machine. The Python virtual machine is an internal component of the Python interpreter and is not available to use directly. Instead, a bytecode interpreter called JCoCo has been developed that mimics a subset of the behavior of the Python 3.2 virtual machine. Instead of writing bytecode files directly, JCoCo supports a Python virtual machine assembly language.

While learning assembly language, we'll limit ourselves to a subset of Python. JCoCo supports boolean values, integers, strings, floats, tuples, lists, and dictionaries. It supports class and function definitions and function calls. It also supports most of the instructions of the Python virtual machine including support for conditional execution, iteration, and exception handling. It does not support importing modules or module level code. JCoCo differs from Python by requiring a main function where execution of a JCoCo assembled program begins.

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_3

To run an assembly language program it must first be assembled, then it can be executed. The JCoCo virtual machine includes the assembler so assembly isn't a separate step. An assembly language programmer writes a program in the JCoCo assembly language format, providing it to JCoCo, which then assembles and interprets the program.

The main difference between JCoCo assembly language and bytecode is the presence of labels in the assembly language format. Labels are the targets of instructions that change the normal sequence of execution of instructions. Instructions like branch and jump instructions are much easier to decipher if it says "jump to loop1" rather than "jump to address 63". Of course, bytecode instructions are encoded as numbers themselves, so the assembler translates "jump to loop1" to something like "48 63" which of course would require a manual to decipher.

Learning to program in assembly isn't all that hard once you learn how constructs like while loops, for loops, if-then statements, function definitions, and function calls are implemented in assembly language. String and list manipulation is another skill that helps if you have examples to follow. A *disassembler* is a tool that will take a machine language program and produce an assembly language version of it. Python includes a module called *dis* that includes a disassembler. When you write a Python program it is parsed and converted to bytecode when read by the interpreter. The *dis* module disassembler produces an assembly language program from this bytecode. JCoCo includes its own disassembler which uses the Python *dis* module and produces output suitable for the JCoCo virtual machine.

The existence of the disassembler for JCoCo means that learning assembly language is as easy as writing a Python program and running it through the disassembler to see how it is implemented in assembly language. That means you can discover how Python is implemented while learning assembly language! Because Python's virtual machine is not guaranteed to be backwards compatible, you must use Python 3.2 when disassembling programs so make sure that version 3.2 is installed on your system. To test this you can try typing "python3.2" in a terminal window in your favorite operating system. If it says command not found, you likely don't have Python 3.2 installed. In that case you can download it from http://python.org or directly from the JCoCo website at http://cs.luther.edu/~leekent/JCoCo. The rest of this chapter introduces you to assembly language programming using the JCoCo virtual machine.

You can download the full binary implementation of the JCoCo virtual machine by going to http://cs.luther.edu/~leekent/JCoCo. Download the zip file containing a coco shell script which runs the Java Virtual Machine on the JCoCo jar file. You can also go to github and get the source code for the reduced functionality JCoCo project at http://github.com/kentdlee/JCoCo.

3.1 Overview of the JCoCo VM

JCoCo, like Python, is a virtual machine, or interpreter, for bytecode instructions. JCoCo is written in Java using object-oriented principles and does not store its instructions in actual bytecode format. Instead, it reads an assembly language file and assembles it building an internal representation of the program as a sequence of functions each with their own sequence of bytecode instructions. CoCo is another implementation of this virtual machine, implemented in C++. You can find documentation on the C++ version at http://cs.luther.edu/~leekent/CoCo. JCoCo is backwards compatible with CoCo, but JCoCo does provide some additional functionality including the ability to define classes, create objects, and utilize single inheritance which are not used extensively in this text. Additionally, JCoCo provides an interactive command-line debugger that can be used for debugging JCoCo assembly language programs.

Most of the material presented in this chapter is true of either JCoCo or CoCo. Chap. 6 again revisits JCoCo as a target language for *Small*, but either JCoCo or CoCo will work as the *Small* compiler target.

A JCoCo program, like programs in other programming languages, utilizes a *run-time stack* to store information about each function called while the program is executing. Each function call in a JCoCo program results in a new stack frame object being created and pushed onto the run-time stack. When a function returns, its corresponding stack frame is popped from the run-time stack and discarded. Figure 3.1 depicts four active function calls. Function A called function B, which called function C, which called function D before any of the functions returned. The top of the stack is at the top of Fig. 3.1. Each stack frame contains all local variables that are defined in the function. Each stack frame also contains two additional stacks, an *operand stack* and a *block stack*.

JCoCo, like the Python virtual machine, is a stack based architecture. This means that operands for instructions are pushed onto an *operand stack*. Virtual machine instructions pop their operands from the operand stack, do their intended operation, and push their results onto the operand stack. Most CPUs are not stack based. Instead they have general purpose registers for holding intermediate results. Stack based architectures manage the set of intermediate results as a stack rather than forcing the programmer to keep track of which registers hold which results. The stack abstraction makes the life of an assembly language programmer a little easier. The operand stack is used by the virtual machine to store all intermediate results of instruction execution. This style of computation has been in use a long time, from Hewlett Packard mainframe computers of the 1960's through the 1980's to calculators still made by Hewlett Packard today. The Java Virtual Machine, or JVM, is another example of a stack machine.

The other stack utilized by JCoCo is a *block stack*. The block stack keeps track of exit points for blocks of code within a JCoCo function. When a loop is entered, the exit address of the loop is pushed onto the block stack. The instructions of each function are at zero-based offsets from the beginning of the function, so we can think of each function having its own instruction address space starting at 0. By storing each loop's exit point address on the block stack, if a break instruction is executed inside a loop, the exit point of the loop can be found and the execution of the break instruction will jump to that address. Exception handlers also push the address of the handler onto the block stack. If an exception occurs, execution jumps to the exception

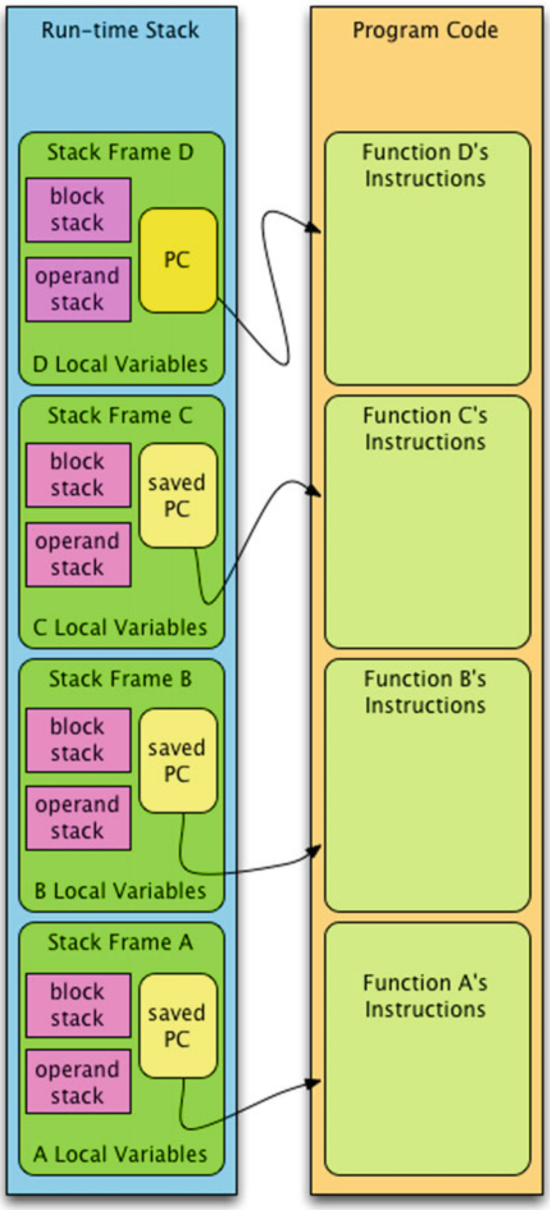

Fig. 3.1 The JCoCo virtual machine

handler by popping the address from the block stack. When a loop or try block is exited, the corresponding block stack address is popped from the block stack.

A *program counter*, or PC, is responsible for holding the address of the next instruction to be executed. The machine proceeds by fetching an instruction from the code, incrementing the PC, and executing the fetched instruction. Execution proceeds this way until a RETURN_VALUE instruction is executed or an exception occurs. When a function call is executed, the current program counter is stored in the stack frame until the called function returns, when the PC is restored to the next instruction in the current stack frame. This is depicted in Fig. 3.1 with the arrows from the stack frames to the code of their corresponding functions.

When an exception occurs, if no matching exception handler is found, execution of the function terminates and control is passed to the previously called function where the exception continues to propagate back until a matching exception handler is found. If no matching handler is found, the complete traceback of the exception is printed. If no exception occurs during the running of a program, execution terminates when the main function executes the RETURN_VALUE instruction.

The specification for JCoCo, including all instructions, global functions, and the complete assembly language BNF supported by JCoCo can be found in *Appendix A*. The rest of this chapter examines various Python language constructs and the corresponding assembly language that implement these constructs. JCoCo assembly language can be learned by examining Python code and learning how it is implemented in assembly language. The rest of this chapter proceeds in this fashion.

3.2 Getting Started

JCoCo includes a disassembler that works with Python 3.2 to disassemble Python programs into JCoCo assembly language programs, providing a great way to learn assembly language programming using the JCoCo virtual machine. Consider the following Python program that adds 5 and 6 together and prints the sum to the screen.

```python
from disassembler import *
import sys

def main():
    x=5
    y=6
    z=x+y
    print(z)

if len(sys.argv) == 1:
    main()
else:
    disassemble(main)
```

Running this with python 3.2 as follows produces this output. Note that the 1 argument is required to get assembly output because of the code on lines 10-13 of the Python program.

```
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
```

The disassembler prints the assembly language program to standard output, which is usually the screen. The second run of the addtwo.py program redirects the standard output to a file called addtwo.casm. The *casm* is the extension chosen for JCoCo assembly language files and stands for CoCo Assembly. This CASM file holds all the lines between the two MyComputer prompts above. To run this program you can invoke the JCoCo virtual machine as shown here.

```
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>
```

The first run invokes *coco* which assembles the program producing the assembled output and then runs the program producing the 11 that appears below the assembled output. The assembled output is shown because the *-v* option was used when invoking *coco*. The assembled output is printed to a stream called *standard error* which is separate from the *standard output* stream where the 11 is printed. To only print the exact output of the program, the *-v* option can be omitted.

In this JCoCo program there is one function called main. The assembly indicates main has 0 formal parameters. Constants that are used in the code include None, 5, and 6. There are three local variables in the function: *x*, *y*, and *z*. The global *print* function is called and so is in the list of globals. Every function in JCoCo has these categories of identifiers and values within each defined function. Sometimes one or more of these categories may be empty and can be omitted in that case.

The instructions follow the *begin* keyword and preceed the *end* keyword. LOAD_CONST loads the constant value at its index (zero based and 1 in this case) into the constants onto the operand stack. JCoCo is a stack machine and therefore all operations are performed with operands pushed and popped from the operand stack.

The STORE_FAST instruction stores a value in the locals list, in this case at offset 0, the location of x. LOAD_FAST does the opposite of STORE_FAST, pushing a value on the operand stack from the locals list of variables. BINARY_ADD pops two operands from the stack and adds them together, pushing the result. CALL_FUNCTION pops the number of arguments specified in the instruction (1 in this case) and then pops the function from the stack. Finally, it calls the popped function with the popped arguments. The result of the function call is left on the top of the operand stack. In the case of the print function, *None* is returned and left on the stack. The POP_TOP instruction pops the *None* from the stack and discards it only to have the main function push a None on the stack just before returning. RETURN_VALUE pops the top argument from the operand stack and returns that value to the calling function. Since main was the only function called, returning from it ends the coco interpretation of the program.

To run this code, you must have the coco executable somewhere in your path. Then you can execute the following code to try it out.

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

3.3 Input/Output

JCoCo provides one built-in function to read input from the keyboard and several functions for writing output to this screen or standard output. The following program demonstrates getting input from the keyboard and printing to standard output.

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

In the Python code in Sect. 3.3, the *input* function is called. Calling input requires a string prompt and returns a string of the input that was entered. Calling the *int* function on a string, as is done in the line that gets the *age* from the user, returns the integer representation of the string's value. Finally, the *print* function takes a random number of arguments, converts each to a string using the __str__ magic method, and prints each string separated by spaces. The first argument to print in the code of Sect. 3.3 is the result of concatenating *name* and the string", a year from now you will be". String concatenation was used because there shouldn't be a space between the *name* value and the comma.

The assembly language that implements the program in Sect. 3.3 is given in Fig. 3.2. Notice that built-in functions like *input*, *int*, and *print* are declared under the *Globals* list. The *name* and *age* variables are the locals.

Line 9 pushes the *input* function onto the operand stack. Line 10 pushes the string prompt for *input*. Line 11 calls the *input* function with the one allowed argument given to it. The *1* in line 11 is the number of arguments. When the *input* function returns it leaves string entered by the user on the operand stack. Line 12 stores that string in the *name* location in the locals.

Line 13 prepares to convert the next input to an integer by first pushing the *int* function on the operand stack. Then line 14 loads the *input* function. Line 15 loads the prompt like line 10 did previously. Line 16 calls the input function. The result is immediately passed to the *int* function by calling it on line 17. The *int* function leaves an integer on the top of the operand stack and line 18 stores that in the *age* variable location.

The next part of the program prints the output. To prepare for calling the *print* function, the arguments must be evaluated first, then *print* can be called. Line 19 pushes the *print* function onto the stack but does not call *print*. There are three arguments to the *print* function. The first argument is the result of concatenating two strings together. Line 20 pushes the *name* variable's value on the stack. Line 21 pushes the string", a year from now you will be" onto the stack. Line 22 calls the __add__ magic method to concatenate the two strings. The BINARY_ADD instruction pops two operands from the stack, calls the __add__ method on the first object popped with the second object as the argument which is described in more detail in *Appendix A*.

Lines 23–25 add together *age* and *1* to get the correct age value to pass to *print*. Line 26 pushes the last string constant on the operand stack and line 27 finally calls

```
1  Function: main/0
2  Constants: None, "Enter your name: ",
3      "Enter your age: ",
4      ", a year from now you will be",
5      1, "years old."
6  Locals: name, age
7  Globals: input, int, print
8  BEGIN
9            LOAD_GLOBAL         0
10           LOAD_CONST          1
11           CALL_FUNCTION       1
12           STORE_FAST          0
13           LOAD_GLOBAL         1
14           LOAD_GLOBAL         0
15           LOAD_CONST          2
16           CALL_FUNCTION       1
17           CALL_FUNCTION       1
18           STORE_FAST          1
19           LOAD_GLOBAL         2
20           LOAD_FAST           0
21           LOAD_CONST          3
22           BINARY_ADD
23           LOAD_FAST           1
24           LOAD_CONST          4
25           BINARY_ADD
26           LOAD_CONST          5
27           CALL_FUNCTION       3
28           POP_TOP
29           LOAD_CONST          0
30           RETURN_VALUE
31 END
```

Fig. 3.2 JCoCo I/O

the *print* function leaving *None* on the operand stack afterwards. Line 28 pops the *None* value and immediately *None* is pushed back on the stack in line 29 because the *main* function returns *None* in this case, which is returned in line 30, ending the iotest.casm program's execution.

A few important things to learn from this section:

- Getting input and producing output rely on the built-in functions *input* and *print*.
- Before a function can be called, it must be pushed on the operand stack. All required arguments to the function must also be pushed onto the stack on top of the function to be called.
- Finally, when a function returns, it leaves its return value on the operand stack.

Practice 3.1 The code in Fig. 3.2 is a bit wasteful which often happens when compiling a program written in a higher level language. Optimize the code in Fig. 3.2 so it contains fewer instructions.

You can check your answer(s) in Section 3.17.1.

3.4 If-Then-Else Statements

Programming languages must be able to execute code based on conditions, either externally provided via input or computed from other values as the program executes. If-then statements are one means of executing code conditionally. The code provided here isolates just an if-then statement to show how it is implemented in JCoCo assembly.

```
import disassembler
def main():
    x=5
    y=6
    if x > y:
        z=x
    else:
        z=y
    print(z)
disassembler.disassemble(main)
```

Disassembling this Python code results in the code in Fig. 3.3. There are new instructions in Fig. 3.3 that haven't been encountered until now, but just as importantly, there are labels in this code. A label provides a symbolic target to jump to in the code. Labels, like *label00* and *label01*, are defined by writing them before an instruction and are terminated with a colon. A label to the right of an instruction is a target for that instruction. Labels are a convenience in all assembly languages. They let assembly language programmers think of jumping to a target in a program, rather than changing the contents of the PC register, which is what actually happens. When a program is executed using JCoCo the labels disappear because JCoCo assembles the code, replacing the labels with the actual PC target addresses. The JCoCo code in Fig. 3.4 shows the JCoCo code after it has been assembled. The assembled code is printed by *coco* when the program is executed.

The first instruction, the *LOAD_CONST*, is at offset 0 in the code. The instructions of each function are at zero-based offsets from the beginning of the function, so we can think of each function as having its own address space starting at zero. In the code in Figs. 3.3 and 3.4 the line number of the first instruction is 6, so 6 can be subtracted from the line numbers to determine any instruction's address within the function and 6 can be added to any target to determine the line number of the target

```
 1  Function: main/0
 2  Constants: None, 5, 6
 3  Locals: x, y, z
 4  Globals: print
 5  BEGIN
 6            LOAD_CONST          1
 7            STORE_FAST          0
 8            LOAD_CONST          2
 9            STORE_FAST          1
10            LOAD_FAST           0
11            LOAD_FAST           1
12            COMPARE_OP          4
13            POP_JUMP_IF_FALSE label00
14            LOAD_FAST           0
15            STORE_FAST          2
16            JUMP_FORWARD label01
17  label00:  LOAD_FAST           1
18            STORE_FAST          2
19  label01:  LOAD_GLOBAL         0
20            LOAD_FAST           2
21            CALL_FUNCTION       1
22            POP_TOP
23            LOAD_CONST          0
24            RETURN_VALUE
25  END
```

Fig. 3.3 If-Then-Else assembly

location. In Fig. 3.4 the target of line 13 is 11 which corresponds to line 17. Looking at Fig. 3.3 this corresponds to the line where *label00* is defined. Likewise, the target of the *JUMP_FORWARD* instruction in Fig. 3.4 is *label01* which is defined on line 19. Subtracting 6, we expect to see 13 as the target PC address in the assembled code of Fig. 3.4.

Consulting the JCoCo BNF in *Appendix A*, there can be multiple labels on one instruction. In addition, instruction addresses have nothing to do with which line they are on. That only appears to be the case in Fig. 3.4 because the instructions are on consecutive lines. But, adding blank lines to the program would do nothing to change the instruction addresses. So, we could have a program like this where one instruction has two labels. These three instructions would be at three addresses within the program even though there are four lines in the code.

```
  onelabel:  LOAD_FAST     1
             STORE_FAST    2
  twolabel:
threelabel: LOAD_GLOBAL    0
```

```
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
          COMPARE_OP            4
          POP_JUMP_IF_FALSE     11
          LOAD_FAST             0
          STORE_FAST            2
          JUMP_FORWARD          13
          LOAD_FAST             1
          STORE_FAST            2
          LOAD_GLOBAL           0
          LOAD_FAST             2
          CALL_FUNCTION         1
          POP_TOP
          LOAD_CONST            0
          RETURN_VALUE
END
```

Fig. 3.4 Assembled code

Labels can be composed of any sequence of letters, digits, underscores, or the @ character, but must start with a letter, underscore, or the @ character. They can be any number of characters long.

In Fig. 3.3, lines 6–11 load the two values to be compared on the stack. The *COMPARE_OP* instruction on line 12 has an argument of 4. Consulting the *COMPARE_OP* instruction in *Appendix A* reveals that a 4 corresponds to a *greater than* comparison. The comparison is done by calling the *__gt__* magic method on the second item from the top of the operand stack and passing it the top of the operand stack. The two operands are popped by the *COMPARE_OP* instruction and a boolean value, either *True* or *False*, is pushed on the operand stack as the result.

The next instruction jumps to the target location if the value left on the operand stack was *False*. Either way, the *POP_JUMP_IF_FALSE* instruction pops the top value from the operand stack.

Take note of line 16 in Fig. 3.3. In assembly there is nothing like an if-then-else statement. Line 15 is the end of the code that implements the *then* part of the statement. Without line 16, JCoCo would continue executing and would go right into the *else* part of the statement. The *JUMP_FORWARD* instruction is necessary to jump past the *else* part of the code if the *then* part was executed. Line 17 begins the *else* code and line 18 is the last instruction of the if-then-else statement. The label definition for *label01* is still part of the if-then-else statement, but labels the instruction immediately following the if-then-else statement.

Practice 3.2 Without touching the code that compares the two values, the assembly in Fig. 3.4 can be optimized to remove at least three instructions. Rewrite the code to remove at least three instructions from this code. With a little more work, five instructions could be removed.

You can check your answer(s) in Section 3.17.2.

3.4.1 If-Then Statements

Frequently if-then statements are written without an else clause. For instance, this program prints *x* if *x* is greater than *y*. In either case *y* is printed.

```python
import disassembler

def main():
    x = 5
    y = 6
    if x > y:
        print(x)

    print(y)

disassembler.disassemble(main)
```

Disassembling this code produces the program in Fig. 3.5. The code is very similar to the code presented in Fig. 3.3. Line 13 once again jumps past the *then* part of the program. Lines 14–17 contain the *then* code. Interestingly, line 18 jumps forward to line 19. Comparing this to the code in Fig. 3.3 where the jump forward jumps past the *else* part, the same happens in Fig. 3.5 except that there is no *else* part of the statement.

Some assembly languages do not have an equivalent to *POP_JUMP_IF_FALSE*. Instead, only an equivalent to *POP_JUMP_IF_TRUE* is available. In that case, the opposite of the condition can be tested and the jump will be executed if the opposite is true, skipping over the *then* part. For instance, if testing for *greater than* is the intent of the code, *less than or equal to* can be tested to jump around the *then* part of an if-then-else statement.

```
Function: main/0
Constants: None, 5, 6
Locals: x, y
Globals: print
BEGIN
          LOAD_CONST          1
          STORE_FAST          0
          LOAD_CONST          2
          STORE_FAST          1
          LOAD_FAST           0
          LOAD_FAST           1
          COMPARE_OP          4
          POP_JUMP_IF_FALSE label00
          LOAD_GLOBAL         0
          LOAD_FAST           0
          CALL_FUNCTION       1
          POP_TOP
          JUMP_FORWARD      label00
label00:  LOAD_GLOBAL         0
          LOAD_FAST           1
          CALL_FUNCTION       1
          POP_TOP
          LOAD_CONST          0
          RETURN_VALUE
END
```

Fig. 3.5 If-Then assembly

Whether testing the original condition or the opposite, clearly the *JUMP_FORWARD* is not needed in the code in Fig. 3.5. As was seen in practice 3.1, the Python compiler generated a wasteful instruction. It isn't wrong to jump forward, it's just not needed. The convenience of writing in a language like Python far outweighs the inconvenience of writing in a language like JCoCo assembly language, so an extra instruction now and then is not that big a deal. In this case though, the Python compiler could be written in such a way as to recognize when the extra instruction is not needed.

Practice 3.3 Rewrite the code in Fig. 3.5 so it executes with the same result using *POP_JUMP_IF_TRUE* instead of the jump if false instruction. Be sure to optimize your code when you write it so there are no unnecessary instructions.

You can check your answer(s) in Section 3.17.3.

3.5 While Loops

Consider this code which computes the Fibonacci number for the value stored in the variable *f*. The sequence of Fibonacci numbers are computed by adding the previous two numbers in the sequence together to get the next number. The sequence consists of 1, 1, 2, 3, 5, 8, 13, 21, and so on, the eighth element of the sequence being 21.

```
import disassembler
def main():
  f=8
  i=1
  j=1
  n=1
  while n < f:
    n=n+1
    tmp = j
    j=j+i
    i = tmp
  print("Fib("+str(n)+") is",i)
disassembler.disassemble(main)
```

The JCoCo assembly for this program implements the while loop of the Python program using *JUMP_ABSOLUTE* and *POP_JUMP_IF_FALSE* instructions. Prior to the loop, the *SETUP_LOOP* instruction's purpose is not readily apparent. In Python a loop may be exited using a *break* instruction. Using *break* inside a loop is not a recommended programming style. A *break* is never needed. It is sometimes used as a convenience. To handle the *break* instruction when it is executed there must be some knowledge about where the loop ends. In the code in Fig. 3.6 the first instruction after the loop is on line 33, where *label02* is defined. The *SETUP_LOOP* instruction pushes the address of that instruction on the block stack. If a break instruction is executed, the block stack is popped and the *PC* is set to the popped instruction address.

Lines 15–18 of Fig. 3.6 implement the comparison of $n < f$ similarly to the way if-then-else comparisons are performed. The first line of this code is labeled with *label00* because the end of the loop jumps back there to see if another iteration should be performed. A while loop continues executing until the condition evaluates to *False* so the *POP_JUMP_IF_FALSE* instruction jumps to *label01* when the loop terminates.

The instruction at *label01* labels the *POP_BLOCK* instruction. This instruction is needed if the loop exits normally, not as the result of a break statement. The block stack is popped, removing the loop exit point from it. When exiting as a result of a break, execution jumps to the instruction at line 33, skipping the *POP_BLOCK* instruction since the break statement already popped the block stack.

An important thing to notice is that a while loop and an if-then-else statement are implemented using the same instructions. There is no special *loop* instruction in assembly language. The overall flow of a while loop is a test before the body of the loop corresponding to the while loop condition. If the loop condition is not

```
Function: main/0
Constants: None,8,1,"Fib(",") is"
Locals: f, i, j, n, tmp
Globals: print, str
BEGIN
          LOAD_CONST       1
          STORE_FAST       0
          LOAD_CONST       2
          STORE_FAST       1
          LOAD_CONST       2
          STORE_FAST       2
          LOAD_CONST       2
          STORE_FAST       3
          SETUP_LOOP label02
label00:  LOAD_FAST        3
          LOAD_FAST        0
          COMPARE_OP       0
          POP_JUMP_IF_FALSE label01
          LOAD_FAST        3
          LOAD_CONST       2
          BINARY_ADD
          STORE_FAST       3
          LOAD_FAST        2
          STORE_FAST       4
          LOAD_FAST        2
          LOAD_FAST        1
          BINARY_ADD
          STORE_FAST       2
          LOAD_FAST        4
          STORE_FAST       1
          JUMP_ABSOLUTE label00
label01:  POP_BLOCK
label02:  LOAD_GLOBAL      0
          LOAD_CONST       3
          LOAD_GLOBAL      1
          LOAD_FAST        3
          CALL_FUNCTION    1
          BINARY_ADD
          LOAD_CONST       4
          BINARY_ADD
          LOAD_FAST        1
          CALL_FUNCTION    2
          POP_TOP
          LOAD_CONST       0
          RETURN_VALUE
END
```

Fig. 3.6 While loop assembly

met, execution jumps to the next instruction after the loop. After the body of the loop a jump returns execution to the while loop condition code to check if another iteration of the body will be performed. This idiom, or pattern of instructions, is used to implement loops and similar patterns are used for loops in other assembly languages as well.

Practice 3.4 Write a short program that tests the use of the *BREAK_LOOP* instruction. You don't have to write a while loop to test this. Simply write some code that uses a *BREAK_LOOP* and prints something to the screen to verify that it worked.

You can check your answer(s) in Section 3.17.4.

3.6 Exception Handling

Exception handling occurs in Python within a try-except statement. Statements within the *try block* are executed and if an exception occurs execution jumps to the *except block* of statements. If main were called on the Python program given here, any error condition would send it to the except block which simply prints the exception in this case. The except block is only executed if there is an error in the try block. Errors that could occur in this program would be a conversion error for either of the two floating point number conversions or a division by zero error. The code catches an exception if a zero is entered for the second value.

```
import disassembler
def main():
    try:
        x = float(input("Enter a number: "))
        y = float(input("Enter a number: "))
        z=x/y
        print(x,"/",y,"=",z)
    except Exception as ex:
        print(ex)
disassembler.disassemble(main)
```

Implementing exception handling in JCoCo is similar in some ways to implementing the *BREAK_LOOP* instruction. The difference is that the exception causes the program to jump from one place to the next instead of the *BREAK_LOOP* instruction. Both exception handling and the break instruction make use of the block stack. When a loop is entered, the *SETUP_LOOP* instruction pushes the exit point of the loop onto the block stack; the exit point being an integer referring to the address of the first instruction after the loop.

To distinguish between loop exit points and exception handling, the *SETUP_EXCEPT* instruction pushes the negative of the except handler's address (i.e. –1*address). So a negative number on the block stack refers to an exception handler while a positive value refers to a loop exit point. In the code in Fig. 3.7 the exception handler's code begins at *label00*.

The *try* block code begins on line 7 with the *SETUP_EXCEPT*. This pushes the handler's address for *label00* on the block stack which corresponds to a –27. Execution proceeds by getting input from the user, converting the input to floats, doing the division, and printing the result. The *print* completes on line 24 where *None*, which is returned by *print*, is popped from the operand stack.

If execution makes it to the end of the *try* block, then no exception occurred and line 25 pops the –27 from the block stack, ending the *try* block. Line 26 jumps past the end of the *except* block.

If an exception occurs, three things are pushed onto the operand stack before any handling of the exception occurs. The *traceback* is pushed first. The traceback is a copy of the run-time stack containing each function call and the stored PC of all pending functions including the current function's stack frame and PC. Above the traceback there are two copies of the exception object pushed on the operand stack when an exception occurs.

If an exception occurs in the *try* block, JCoCo consults the block stack and pops values until a negative address is found corresponding to some *except* block. Multiple try-except statements may be nested, so it is possible that the block stack will contain more than one negative address. When a negative address is found, the *PC* is set to its positive value causing execution to jump to the *except* block. In Fig. 3.7, that's line 27. The traceback and two copies of the exception are pushed onto the stack prior to line 27 being executed.

Why are three objects pushed on the operand stack when an exception occurs? Python's *RAISE_VARARGS* instruction describes the contents of the operand stack as TOS2 containing the traceback, TOS1 the parameter, and TOS the exception object. In the JCoCo implementation the parameter to an exception can be retrieved by converting the exception to a string, so the object at TOS1 is simply the exception again. For the sake of compatibility with the Python disassembler JCoCo pushes three operands pushed onto the operand stack when an exception is raised.

Exception handlers in Python may be written to match only certain types of exceptions. For instance, in Python a division by zero exception is different than a float conversion error. The JCoCo virtual machine currently only has one type of exception, called *Exception*. It is possible to extend JCoCo to support other types of exceptions, but currently there is only one type of exception object that can be created. The argument to the exception object can be anything that is desired. The program in Fig. 3.7 is written to catch any type of exception, but it could be written to catch only a certain type of exception. Line 27 duplicates the exception object on the top of the operand stack. Line 35 loads a global *Exception* object onto the stack. The *COMPARE_OP 10* instruction compares the exception using the exception match comparison which calls the *__excmatch__* magic method to see if there is a match between the thrown exception and the specified pattern. If there is not a match, line 30 jumps to the end of the except block. The *END_FINALLY* instruction on line 47 detects if the exception was handled and if not, it re-throws the exception for some outer exception handling block.

```
Function: main/0
Constants: None,
    "Enter a number: ", "/", "="
Locals: x, y, z, ex
Globals: float,input,print,Exception
BEGIN
          SETUP_EXCEPT label00
          LOAD_GLOBAL          0
          LOAD_GLOBAL          1
          LOAD_CONST           1
          CALL_FUNCTION        1
          CALL_FUNCTION        1
          STORE_FAST           0
          ...
          BINARY_TRUE_DIVIDE
          STORE_FAST           2
          LOAD_GLOBAL          2
          LOAD_FAST            0
          LOAD_CONST           2
          LOAD_FAST            1
          LOAD_CONST           3
          LOAD_FAST            2
          CALL_FUNCTION        5
          POP_TOP
          POP_BLOCK
          JUMP_FORWARD label03
label00:  DUP_TOP
          LOAD_GLOBAL          3
          COMPARE_OP          10
          POP_JUMP_IF_FALSE label02
          POP_TOP
          STORE_FAST           3
          POP_TOP
          SETUP_FINALLY label01
          LOAD_GLOBAL          2
          LOAD_FAST            3
          CALL_FUNCTION        1
          POP_TOP
          POP_BLOCK
          POP_EXCEPT
          LOAD_CONST           0
label01:  LOAD_CONST           0
          STORE_FAST           3
          DELETE_FAST          3
          END_FINALLY
          JUMP_FORWARD label03
label02:  END_FINALLY
label03:  LOAD_CONST           0
          RETURN_VALUE
END
```

Fig. 3.7 Exception handling assembly

If the exception was a match, execution of the handler code commences as it does on line 31 of the program. The top of the operand stack contains the extra exception object so it is thrown away by line 31. Line 32 takes the remaining exception object and makes the *ex* reference point to it. Line 33 pops the traceback from the operand stack.

Should an exception occur while executing an exception handler, then JCoCo must clean up from the exception. Line 34 executes the *SETUP_FINALLY* instruction to push another block stack record to keep track of the end of the exception handler. Lines 35–38 print the exception named *ex* in the code.

Line 39 pops the exit address that was pushed by the *SETUP_FINALLY* instruction. The *POP_EXCEPT* instruction on line 40 then pops the block stack address for the exception handler exit address. Line 41 pushes a *None* on the operand stack.

Line 42 is either the next instruction executed or it is jumped to as a result of an exception while executing the handler code for the previous exception. Either way, the *ex* variable is made to refer to *None*. The *DELETE_FAST* instruction doesn't appear to do much in this code. It is generated by the disassembler, but appears to delete *None* which doesn't seem to need to be done.

The last instruction of the handler code, the *END_FINALLY* instruction checks to see if the exception was handled. In this case, it was handled and the instruction does nothing. If execution jumps to line 47 then the exception handler did not match the raised exception and therefore the exception is re-raised. Line 48 wraps up by setting up to return *None* from the *main* function.

Practice 3.5 Write a short program that tests creating an exception, raising it, and printing the handled exception. Write this as a JCoCo program without using the disassembler.

You can check your answer(s) in Section 3.17.5.

3.7 List Constants

Building a compound value like a list is not too hard. To build a list constant using JCoCo you push the elements of the list on the operand stack in the order you want them to appear in the list. Then you call the *BUILD_LIST* instruction. The argument to the instruction specifies the length of the list. This code builds a list and prints it to the screen.

```
import disassembler
def main():
    lst = ["hello","world"]
    print(lst)
disassembler.disassemble(main)
```

```
1  Function: main/0
2  Constants: None, "hello", "world"
3  Locals: lst
4  Globals: print
5  BEGIN
6            LOAD_CONST          1
7            LOAD_CONST          2
8            BUILD_LIST          2
9            STORE_FAST          0
10           LOAD_GLOBAL         0
11           LOAD_FAST           0
12           CALL_FUNCTION       1
13           POP_TOP
14           LOAD_CONST          0
15           RETURN_VALUE
16 END
```

Fig. 3.8 Assembly for building a list

The assembly language program in Fig. 3.8 builds a list with two elements: [‘hello’, ‘world’]. Lines 6 and 7 push the two strings on the operand stack. Line 8 pops the two operands from the stack, builds the list object, and pushes the resulting list on the operand stack. Python defines the *__str__* magic method for built-in type of value, which is called on the list on line 12.

If you run this program using the JCoCo interpreter you will notice that [‘hello’, ‘world’] is not printed to the screen. Instead, [hello, world] is printed. This is because currently the *__str__* method is called on each element of the list to convert it to a string for printing. This is not the correct method to call. Instead, the *__repr__* magic method should be called which returns a printable representation of the value retaining any type information. In the next chapter there will be an opportunity to fix this.

3.8 Calling a Method

Calling functions like *print* and *input* was relatively simple. Push the function name followed by the arguments to the function on the operand stack. Then, call the function with the *CALL_FUNCTION* instruction. But, how about methods? How does a method like *split* get called on a string? Here is a program that demonstrates how to call *split* in Python.

```
 1  Function: main/0
 2  Constants: None,"Enter integers:"
 3  Locals: s, lst
 4  Globals: input, split, print
 5  BEGIN
 6            LOAD_GLOBAL          0
 7            LOAD_CONST           1
 8            CALL_FUNCTION        1
 9            STORE_FAST           0
10            LOAD_FAST            0
11            LOAD_ATTR            1
12            CALL_FUNCTION        0
13            STORE_FAST           1
14            LOAD_GLOBAL          2
15            LOAD_FAST            1
16            CALL_FUNCTION        1
17            POP_TOP
18            LOAD_CONST           0
19            RETURN_VALUE
20  END
```

Fig. 3.9 Assembly for Calling a method

```
import disassembler
def main():
    s = input("Enter integers:")
    lst = s.split()
    print(lst)
disassembler.disassemble(main)
```

Line 6 of the assembly language code in Fig. 3.9 prepares to call the *input* function by loading the name *input* onto the operand stack. Line 7 loads the argument to *input*, the prompt string. Line 8 calls the input function leaving the entered text on the operand stack. Calling *split* is done similarly.

In this Python code the syntax of calling *input* and *split* is quite different. Python sees the difference and uses the *LOAD_ATTR* instruction in the assembly language instructions to get the *split* attribute of the object referred to by *s*. Line 10 loads the object referred to by *s* on the stack. Then line 11 finds the *split* attribute of that object. Each object in JCoCo and Python contains a dictionary of all the object's attributes. This *LOAD_ATTR* instruction examines the dictionary and with the key found in the globals list at the operands index. It then loads that attribute onto the operand stack. The *CALL_FUNCTION* instruction then calls the method that was located with the *LOAD_ATTR* instruction.

The *STORE_ATTR* instruction stores an attribute in an object in much the same way that an attribute is loaded. JCoCo does not presently support the *STORE_ATTR* instruction but could with relatively little effort. The ability to load and store object attributes means that JCoCo could be used to implement an object-oriented language. This makes sense since Python is an object-oriented language.

Practice 3.6 Normally, if you want to add to numbers together in Python, like 5 and 6, you write 5+6. This corresponds to using the *BINARY_ADD* instruction in JCoCo which in turn calls the magic method *__add__* with the method call 5.__add__(6). Write a short JCoCo program where you add two integers together without using the *BINARY_ADD* instruction. Print the result to the screen.

You can check your answer(s) in Section 3.17.6.

3.9 Iterating Over a List

Iterating through a sequence of any sort in JCoCo requires an iterator. There are iterator objects for every type of sequence: lists, tuples, strings, and other types of sequences that have yet to be introduced. Here is a Python program that splits a string into a list of strings and iterates over the list.

```
from disassembler import *
def main():
    x = input("Enter a list: ")
    lst = x.split()
    for b in lst:
        print(b)
disassemble(main)
```

Lines 6–8 of the assembly code in Fig. 3.10 gets an input string from the user, leaving it on the operand stack. Line 9 stores this in the variable *x*. Lines 10–12 call the *split* method on this string, leaving a list object on the top of the operand stack. The list contains the list of space separated strings from the original string in *x*. Line 13 stores this list in the variable *lst*.

Line 14 sets up the exit point of a loop as was covered earlier in this chapter. Line 15 loads the *lst* variable onto the operand stack. The *GET_ITER* instruction creates an iterator with the top of the operand stack. The *lst* is popped from the operand stack during this instruction and the resulting iterator is pushed onto the stack.

An iterator has a *__next__* magic method that is called by the *FOR_ITER* instruction. When *FOR_ITER* executes the iterator is popped from the stack, *__next__* is called on it, and the iterator and the next value from the sequence are pushed onto the operand stack. The iterator is left below the next value in the sequence at TOS1. When *__next__* is called on the iterator and there are no more elements left in the sequence, the PC is set to the label of the *FOR_ITER* instruction, ending the loop.

When the loop is finished the block stack is popped to clean up from the loop. Line 25 loads the *None* on the stack before returning from the *main* function.

```
Function: main/0
Constants: None, "Enter a list: "
Locals: x, lst, b
Globals: input, split, 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_FAST        1
          GET_ITER
label00:  FOR_ITER label01
          STORE_FAST       2
          LOAD_GLOBAL      2
          LOAD_FAST        2
          CALL_FUNCTION    1
          POP_TOP
          JUMP_ABSOLUTE  label00
label01:  POP_BLOCK
label02:  LOAD_CONST       0
          RETURN_VALUE
END
```

Fig. 3.10 List iteration assembly

Practice 3.7 Write a JCoCo program that gets a string from the user and iterates over the characters of the string, printing them to the screen.

You can check your answer(s) in Section 3.17.7.

3.10 Range Objects and Lazy Evaluation

Indexing into a sequence is another way to iterate in a program. When you index into a list, you use a subscript to retrieve an element of the list. Generally, indices are zero-based. So the first element of a sequence is at index 0, the second at index 1, and so on.

There are two versions of Python in use today. Version 2, while older is still widely used because there are many Python programs that were written using it and there is a cost to converting them to use Python 3. Python 3 was created so new features could be added that might be incompatible with the older version. One difference was in the range function. In Python 2, the range function generated a list of integers of the specified size and values. This is inefficient because some ranges might consist of millions of integers. A million integers takes up a lot of space in memory and takes some time to generate. In addition, depending on how code is written, not all the integers in a range may be needed. These problems are a result of *eager evaluation* of the range function. Eager evaluation is when an entire sequence is generated before any element of the sequence will actually be used. In Python 2 the entire list of integers is created as soon as the range function is called even though the code can only use one integer at a time.

Python 3 has dealt with the *eager evaluation* of the range function by defining a range object that is *lazily evaluated*. This means that when you call the range function to generate a million integers, you don't get any of them right away. Instead, you get a *range object*. From the range object you can access an iterator. When *__next__* is called on an iterator you get the next item in the sequence. When *__next__* is called on a range object iterator you get the next integer in the range's sequence. *Lazy evaluation* is when the next value in a sequence is generated only when it is ready to be used and not before. This code creates a range object. The range object is designed to provide lazy evaluation of integer sequences.

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

This Python code uses indices to iterate backwards through a list. In this case an iterator over the range object yields a descending list of integers which are the indices into the list of values entered by the user. If the use enters four space separated values, then the range object will yield the sequence [3, 2, 1, 0]. The first argument to range is the start value, the second is one past the stop value, and the third argument is the increment. So the sequence in the Python code in Sect. 3.10 is a descending sequence that goes down one integer at a time from the length of the list minus one to zero.

The JCoCo assembly code in Fig. 3.11 implements this same program. Lines 15–23 set up for calling the range function with the three integer values. Lines 15–20 call the *len* function to get the length of the list and subtract one. Lines 21 and 22 put

```
 1  Function: main/0
 2  Constants: None,"Enter list: ",1,-1,-1
 3  Locals: x, lst, i
 4  Globals: input,split,range,len,print
 5  BEGIN
 6            LOAD_GLOBAL         0
 7            LOAD_CONST          1
 8            CALL_FUNCTION       1
 9            STORE_FAST          0
10            LOAD_FAST           0
11            LOAD_ATTR           1
12            CALL_FUNCTION       0
13            STORE_FAST          1
14            SETUP_LOOP label02
15            LOAD_GLOBAL         2
16            LOAD_GLOBAL         3
17            LOAD_FAST           1
18            CALL_FUNCTION       1
19            LOAD_CONST          2
20            BINARY_SUBTRACT
21            LOAD_CONST          3
22            LOAD_CONST          4
23            CALL_FUNCTION       3
24            GET_ITER
25  label00:  FOR_ITER label01
26            STORE_FAST          2
27            LOAD_GLOBAL         4
28            LOAD_FAST           1
29            LOAD_FAST           2
30            BINARY_SUBSCR
31            CALL_FUNCTION       1
32            POP_TOP
33            JUMP_ABSOLUTE label00
34  label01:  POP_BLOCK
35  label02:  LOAD_CONST          0
36            RETURN_VALUE
37  END
```

Fig. 3.11 Range assembly

two –1 values on the operand stack. Line 23 calls the range function which creates and pushes a range object onto the operand stack as its result.

Line 24 creates an iterator for the range object. As described in the last section, the *FOR_ITER* instruction calls the *__next__* magic method on the iterator to get the next integer in the range's sequence. The lazy evaluation occurs because an iterator keeps track of which integer is the next value in the sequence. Line 26 stores the next integer in the variable *i*.

The *BINARY_SUBSCR* instruction is an instruction that has not been encountered yet in this chapter. Line 28 loads the list called *lst* onto the operand stack. Line 29 loads the value of *i* onto the operand stack. The *BINARY_SUBSCR* instruction indexes into *lst* at position *i* and pushes the value found at that position onto the operand stack. That value is printed by the print function call on line 31 of the program.

Lazy evaluation is an important programming language concept. If you ever find yourself writing code that must generate a predictable sequence of values you probably want to generate that sequence lazily. Iterators, like range iterators, are the means by which we can lazily access a sequence of values and range objects define a sequence of integers without eagerly generating all of them.

3.11 Functions and Closures

Up to this point in the chapter all the example programs have been defined in a *main* function. JCoCo supports the definition of multiple functions and even nested functions. Here is a Python program that demonstrates how to write nested functions in the Python programming language. The *main* function calls the function named *f* which returns the function *g* nested inside the *f* function. The *g* function returns *x*. This program demonstrates nested functions in JCoCo along with how to build a closure.

```
import disassembler
def main():
  x = 10
  def f(x):
    def g():
      return x
    return g
  print(f(3)())
disassembler.disassemble(main)
```

Notice the Python code in section 3.11 calls the disassembler on the top-level function *main*. It is not called on *f* or *g* because they are nested inside *main* and the disassembler automatically disassembles any nested functions of a disassembled function.

The format of the corresponding JCoCo program in Fig. 3.12 is worth noting as well. The top level *main* function is defined along the left hand side. Indentation has no effect on JCoCo but visually you see that *f* is nested inside *main*. The function *g* is nested inside *f* because it appears immediately after the first line of the definition of *f* on line 3. The rest of the definition of *f* starts again on line 10 and extends to line 21. The definition of *g* starts on line 3 and extends to line 9.

The number of arguments for each function is given by the integer after the slash. The *f/1* indicates that *f* expects one argument. The *main* and *g* functions expect zero arguments. These values are used during a function call to verify that the function is called with the required number of arguments.

```
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
```

Fig. 3.12 Nested functions assembly

Examine the Python code in section 3.11 carefully. The *main* function calls the function *f* which returns the function *g*. Notice that *f returns g*, it does not *call g*. In the print statement of *main* the function *f* is called, passing 3 to the function that returns *g*. The extra set of parens after the function call *f(3)* calls *g*. This is a valid Python program, but not a common one. The question is: What does the program print? There are two possible choices it seems: either 10 or 3. Which seems more likely?

On the one hand, *g* is being called from the *main* function where *x* is equal to 10. If the program printed 10, we would say that Python is a dynamically scoped language, meaning that the function executes in the environment in which it is called. Since *g* is called from *main* the value of *x* is 10 and in a dynamically scoped language 10 would be printed. The word *dynamic* is used because if *g* were called in another environment it may return something completely different. We can only determine what *g* will return by tracing the execution of the program to the point where *g* is called.

On the other hand, *g* was defined in the scope of an *x* whose value was 3. In that case, the environment in which *g* executes is the environment provided by *f*. If 3 is printed then Python is a statically scoped language meaning that we need only understand what the environment contained when *g* was defined, not when it was called. In a statically scoped language this specific instance of *g* will return the same value each and every time it is called, not matter where it is called in the program. The value of *x* is determined when *g* is defined.

Dynamically scoped languages are rare. Lisp, when it was first defined, was dynamically scoped. McCarthy quickly corrected that and made Lisp a statically scoped language. It is interesting to note that Emacs Lisp is dynamically scoped. Python is statically scoped as are most modern programming languages.

To execute functions in a statically scoped language, two pieces are needed when a function may return another function. To execute *g* not only is the code for *g* required, but so also is the environment in which this instance of *g* was defined. A *closure* is formed. A closure is the environment in which a function is defined and the code for the function itself. This closure is what is called when the function *g* is finally called in *main*.

Take a look at the JCoCo code for this program in Fig. 3.12. Line 14 begins creating a new closure object in the body of function *f* by loading the cell variable named *x* onto the stack. A cell variable is an indirect reference to a value. Figure 3.13 depicts what is happening in the program just before the *x* is returned in the function *g*. A variable in Python, like Java and many other languages, is actually a *reference* that points to a *value*. Values exist on the heap and are created dynamically as the program executes. When a variable is assigned to a new value, the variables reference is made to point to a new value on the heap. The space for values on the heap that are no longer needed is reclaimed by a *garbage collector* that frees space on the heap so it can be re-used. In Fig. 3.13 there are three values on the heap, a 10, a 3, and one other value called a cell in JCoCo and the Python virtual machine.

Because the function *g* needs access to the variable *x* outside the function *f*, the *3* is indirectly referenced through a cell variable. The *LOAD_CLOSURE* instruction

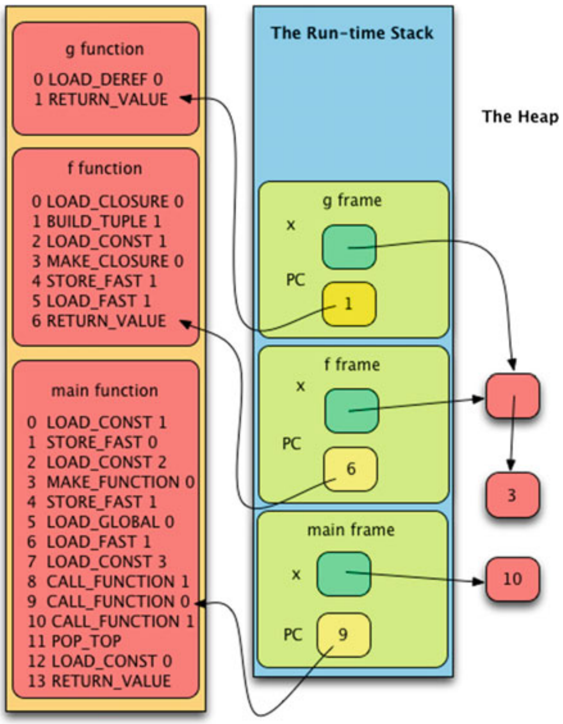

Fig. 3.13 Execution of nested.casm

pushes that cell variable onto the stack to be used in the closure. Since only one value is needed from the environment, the next instruction on line 15 builds a tuple of all the values needed from the environment. Line 16 loads the code for *g* onto the stack. Line 17 forms the closure by popping the function and the environment from the stack and building a *closure* object.

The variable *x* is a local variable for the function *f*. But, because *x* is referenced in *g* and *g* is nested inside *f*, the variable *x* is also listed as a cell variable in *f*. A cell variable is an indirect reference to a value. This means there is one extra step to finding the value that *x* refers to. We must go through the cell to get to the 3.

The *LOAD_DEREF* instruction on line 7 is new. A *LOAD_DEREF* loads the value that is referenced by the reference pointed to in the list of cellvars. So, this instructions pushes the 3 onto the operand stack. Finally, line 35 calls the closure consisting of the function and its data.

In the function *g* the freevars refer to the tuple of references in the closure that was just called, so the first instruction, the *LOAD_DEREF*, loads the 3 onto the operand stack. Figure 3.13 depicts this state right before the *RETURN_VALUE* instruction is executed.

To finish up the execution of this program a 3 is returned from the call to *g* and its frame is popped from the run-time stack. Control returns to *main* where the 3 is printed. After returning from *main* its frame is also popped from the run-time stack which ends the program.

Practice 3.8 The program in Fig. 3.12 would work just fine without the cell. The variable *x* could refer directly to the 3 in both the *f* and *g* functions without any ramifications. Yet, a cell variable is needed in some circumstances. Can you come up with an example where a cell variable is absolutely needed?

You can check your answer(s) in Section 3.17.8.

3.12 Recursion

Functions in JCoCo can call themselves. A function that calls itself is a recursive function. Recursive functions are studied in some detail in Chap. 5 of this text. Learning to write recursive functions well is not hard if you follow some basic rules. The mechanics of writing a recursive function include providing a base case that comes first in the function. Then, the solution to the problem you are solving must be solved by calling the same function on some smaller piece of data while using that result to construct a solution to the bigger problem.

Consider the factorial definition. Factorial of zero, written 0!, is defined to be 1. This is the base case. For integer *n* greater than 0, *n! = n*(n–1)!*. This is a recursive definition because factorial is defined in terms of itself. It is called on something smaller, meaning *n–1* which is closer to the base case, and the result is used in computing *n!*. Here is a Python program that computes *5!*.

```
import disassembler
def factorial(n):
    if n==0:
        return 1
    return n*factorial(n-1)
def main():
    print(factorial(5))

disassembler.disassemble(factorial)
disassembler.disassemble(main)
```

The JCoCo implementation of this program is given in Fig. 3.14. The program begins in main by loading 5 on the operand stack and calling the *factorial* function. The result is printed to the screen with the print function.

Calling *factorial* jumps to the first instruction of the function where *n* is loaded onto the operand stack, which in this case is 5. Lines 7–8 compare *n* to 0 and if the two values are equal, 1 is returned. Notice that the *RETURN_VALUE* instruction appears in the middle of the factorial function in this case. A return instruction doesn't have

```
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
```

Fig. 3.14 Recursion assembly

to appear at the end of a function. It can appear anywhere it makes sense and in this case, it makes sense to return from the base case as soon as soon as possible.

The code from *label00* forward is the recursive case since otherwise we would have returned already. The code subtracts one from *n* and calls factorial with that new, smaller value. Notice that the recursive function call is identical to any other function call. Finally, after the function call the result of the call is on the operand stack and it is multiplied by *n* to get *n!* which is returned.

Because this is a recursive function, the preceding two paragraphs are repeated 5 more times, each time reducing *n* by 1. The program continues to count down until 1 is returned for the factorial of 0. At its deepest, there are 7 stack frames on the run-time stack for this program: one for the *main* function, and six more for the recursive *factorial* function calls. The run-time stack grows to 7 stack frames deep when the base case is executed and then shrinks again as the recursion unwinds. Finally, when the program returns to *main*, 120 is printed to the screen.

Practice 3.9 Draw a picture of the run-time stack just before the instruction on line 11 of Fig. 3.14 is executed. Use Fig. 3.13 as a guide to how you draw this picture. Be sure to include the code, the values of *n*, and the *PC* values.

You can check your answer(s) in Section 3.17.9.

3.13 Support for Classes and Objects

In Python a class definition consists of a class declaration. The class declaration contains a magic method named *__init__* which is responsible for initializing any created object. On line 14 of this Python program an object is instantiated. Python automatically calls the constructor to initialize the space allocated by Python for the object.

```
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)
```

A class is a collection of functions that all operate on some given grouping of data. For instance, the class *Dog* contains a function called *eat* and another called *speak*. Both the *eat* and *speak* functions operate on objects of type *Dog*.

The functions of a class definition are often called methods to differentiate them from stand-alone functions. Methods are provided a reference to the *current object* which is the collection of data on which a method operates. The reference *self* is used to reference the current object and is always the first parameter to a method in Python. This description of methods isn't entirely accurate when considering the Python virtual machine.

In the Python virtual machine *methods* are created from the class' *functions* when an object is instantiated. Consider the assembly program in Fig. 3.15 which demonstrates the instantiation of a class called *Dog*. In line 36 of the main function the class called *Dog* is *called*. Calling a class in Python means executing the code that allocates a *Dog* object in memory. This work is handled by the virtual machine. The assembly language program does not directly allocate the space for the object through any instruction. It is accomplished by calling the class.

All objects in Python consist of a dictionary that stores the attributes of the object. When the *Dog* class is called, the dictionary in the *Dog* object is initialized with the methods *Dog* object. The methods of the *Dog* object are essentially the functions of the *Dog* class. The difference between a method and a function is the *self* parameter. A method provides the *self* argument to its function by providing a reference to the current object as the first parameter. The *method* is a wrapper for the *function*. After the methods are stored in the object's dictionary, the *__init__* method is called to perform any further initialization of the object.

To get a better understanding of the difference between methods and functions, consider the code that calls the *eat* method. When the *eat* method is called on the *Dog* object, the method provides a reference to the current *Dog* object *mesa* as the first parameter before calling the *Dog* class' *eat* function. This is appearent starting on line 38 in Fig. 3.15. Line 38 loads the the reference for *mesa* onto the operand stack. Then line 39 looks up the *eat* method in the object's dictionary, leaving the method on top of the operand stack, but not the reference to the Dog object. Also, notice that no arguments are loaded on top of the method. Yet, the *eat* function has one parameter, self. When the *eat* method is called on line 40 the virtual machine loads the self parameter before calling the *eat* function. The distinction between methods and functions is revisted again in the next chapter.

Practice 3.10 In this section it was stated that every object consists of a dictionary which holds the attributes of the object. What is stored in the dictionary of the object that *mesa* refers to in this section?

You can check your answer(s) in Section 3.17.10.

```
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
```

Fig. 3.15 The dog class

3.13.1 Inheritance

In object-oriented programming, inheritance comes into play when one class inherits from another. Inheritance is useful for polymorphism and code re-use. When programming using Python, polymorphism happens without inheritance because Python is a dynamically typed language, meaning that all method calls are looked up at run-time as was seen in the last section when the *LOAD_ATTR* instruction was executed. The *LOAD_ATTR* instruction looks up a method by name in the object's dictionary. The run-time look up of methods by name creates the polymorphic behavior of Python. The only purpose of inheritance in Python is code re-use. The next chapter will have more on polymorphism and how it applies to Java programming where inheritance is needed to implement polymorphism.

Consider the Python program in this section. Again there is a *Dog* class that this time inherits from an *Animal* class. The *Animal* class defines an *eat* method which is re-used by the *Dog* class. The *Animal* constructor also contains code that is re-used by the *Dog* class. But, the *Dog* class defines its own *speak* method, overriding the *speak* method in the *Animal* class.

The inheritance of *Animal* contributing to the *Dog* class is indicated by writing *Dog(Animal)* on line 12. The call to *super* on line 14 returns an instance of a super class object which can be used to reference the super class, in this case the *Animal* class. Python programs can use multiple inheritance. This is not true in JCoCo. Only single inheritance is currently supported.

```
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
```

Fig. 3.16 Inheritance in JCoCo - part 1

Practice 3.11 Code was omitted in Figs. 3.16 and 3.17 for brevity in the chapter. Pick a method and complete the assembly code according to the original Python code from which it is derived.

You can check your answer(s) in Section 3.17.11.

```
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
```

Fig. 3.17 Inheritance in JCoCo - part 2

3.13.2 Dynamically Created Classes

The previous section demonstrates declaring a class and creating objects in the JCoCo assembly language. It is also possible to create a class dynamically, at run-time. This also makes it possible to define class variables if desired. A class variable is a variable assigned to the class instead of instances of the class (i.e. objects). Consider this Python program where dogNumber is a class variable that can be used to count the number of instances of an object that have been created. Building this class requires that extra code be executed when building the class. The class variable must be initialized to 0.

```python
import disassembler

def main():

    class Dog:
        dogNumber = 0

        def __init__(self,name):
            self.name = name
            self.id = Dog.dogNumber
            Dog.dogNumber += 1

        def speak(self):
            print("Dog number: ", self.id)

    x = Dog("Mesa")
    y = Dog("Sequoia")

    x.speak()
    y.speak()

disassembler.disassemble(main)
```

The assembly code for this program looks a bit different than the previous section. Instead of seeing a *Dog* class, there is a *Dog* function. The *Dog* function becomes the *Dog* class as a result of its execution. To explain what happens, let's start with Fig. 3.19 which contains the code for the main function. Line 55 of this code contains the *LOAD_BUILD_CLASS* instruction. This instruction loads a built-in function which builds a class from two arguments. A closure is one argument. As was stated in Sect. 3.11, a closure is code and an environment. We will visit this again in chapters 4 and 6. A closure is both code and the environment (i.e. the collection of variables) in which the code should be executed. The closure in this example is responsible for building the class' contents including its class variable and the methods of the class, which if you recall from earlier in this section are really functions until an object is instantiated.

The other argument to the built-in class builder function is the name of the class. Line 56 creates a closure. Line 57 buildes a tuple from the closure. Line 58 loads the code for the class initialization (i.e. the code for the *Dog* function). Line 59 builds the closure with the tuple and the code. Line 60 loads the name of the class onto the operand stack. Line 61 then calls the built-in class builder function passing to it the closure and the name of the class.

The built-in class builder function then does some housekeeping by creating a class instance, naming it *Dog* since that was passed as the name of the class, and calls the *Dog* function to complete the class instantiation. This takes us to the code on line 33 in Fig. 3.18.

```
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
```

Fig. 3.18 Dynamically created class - part 1

```
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
```

Fig. 3.19 Dynamically created class - part 2

The *STORE_LOCALS* instruction on line 34 deserves some explanation. The local variables of the function *Dog* are a dictionary or map from strings (i.e. the names of the variables) to their values. When the built-in class builder function calls the *Dog* function to complete the construction of the class, it passes the dictionary for the class into the function. This dictionary is the dictionary of the class *Dog* and by executing the *STORE_LOCALS* instruction the dictionary also becomes the dictionary of locals for the *Dog* function. So, anything that is stored in the local variables will then be stored in the class instance. This sharing of the local variables dictionary and the class dictionary simplifies the class construction by making any variables stored in the *Dog* function also named variables, including named methods, in the class instance.

Line 35 of Fig. 3.18 stores *Dog* as the module name. The *__name__* attribute is already set to *Dog* by the built-in class builder function. So line 36 gives the same name to the *__module__* attribute.

Lines 37 and 38 initialize the class variable *dogNumber* to 0. Line 39 begins the work of adding the functions into the class instance, which will be instantiated to methods when a *Dog* instance is created. The first function to be stored is the *__init__* constructor which happens on lines 39–43. Lines 44–46 store the *speak* function in the class. The reason the constructor takes a bit more work is because it references and increments the class variable *dogNumber* and because of this the constructor needs both the code and the environment to executed correctly. The *speak* method does not reference any class variables so no environment is needed. The *MAKE_FUNCTION* instruction builds a closure with an empty environment.

While classes can be built either dynamically (i.e. at run-time) or statically using the assembly language syntax, the disassembler will use dynamic allocation when the environment is used in one of the instance methods of the class. Using the environment requires a closure and closures can be constructed during dynamic allocation of the class.

Practice 3.12 In some detail, describe the contents of the operand stack before and after the built-in class builder function is called to create a class instance.
You can check your answer(s) in Section 3.17.12.

3.14 Chapter Summary

An understanding of assembly language is key to learning how higher level programming languages work. This chapter introduced assembly language programming through a series of examples, drawing parallels between Python and Python virtual machine or JCoCo instructions. The use of a disassembler was key to gaining this insight and is a great tool to be able to use with any platform.

Most of the key constructs of programming languages were presented as both Python programs and JCoCo programs. The chapter concluded by covering classes, inheritance, and dynamic class creation.

The assembly language covered in this chapter comes up again in Chaps. 4 and 6. Chapter 4 covers the implementation of the JCoCo virtual machine and Chap. 6 implements a high-level functional language compiler that produces JCoCo assembly language programs.

JCoCo is an assembler/virtual machine for Python virtual machine instructions. Of course, there are other assembly languages. MIPS is a CPU architecture that has wide support for writing assembly language programs including a MIPS simulator called SPIM. In fact, assemblers are available for pretty much any hardware/operating system combination in use today. Intel/Linux, Intel/Windows, Intel/Mac OS X all support assembly language programming. The Java Virtual Machine can be programmed with the instructions of the JVM using a java assembler called Jasmin. Assembly language is the fundamental language that all higher level programming languages use in their implementations.

3.15 Review Questions

1. How do the Python virtual machine and JCoCo differ? Name three differences between the two implementations.

2. What is a disassembler?
3. What is an assembler?
4. What is a stack frame? Where are they stored? What goes inside a stack frame?
5. What is the purpose of the block stack and where is it stored?
6. What is the purpose of the Program Counter?
7. Name an instruction that is responsible for creating a list object and describe how it works.
8. Describe the execution of the *STORE_FAST* and *LOAD_FAST* instructions.
9. How can JCoCo read a line of input from the keyboard?
10. What is the difference between a disassembled Python program and an assembled JCoCo program? Provide a short example and point out the differences.
11. When a Python while loop is implemented in JCoCo, what is the last instruction of the loop and what is its purpose?
12. What do exception handling and loops have in common in the JCoCo implementation?
13. What is lazy evaluation and why is it important to Python and JCoCo?
14. What is a closure and why are closures needed?
15. How do you create an instance of a class in JCoCo? What instructions must be executed to create objects?
16. Write a class, using JCoCo, and create some instances of the class.

3.16 Exercises

1. Consulting the JCoCo assembly language program in *the solution to exercise 3.2*, provide the contents of the operand stack after each instruction is executed.
2. Write a JCoCo program which reads an integer from the user and then creates a list of all the even numbers from 0 up to and including that integer. The program should conclude printing the list to the screen. Test your program with JCoCo to be sure it works. Do this with as few instructions as possible.
3. With as few instructions as possible add some exception handling to the previous exercise to print “You didn’t enter an integer!” if the user fails to enter an integer in their program.
4. In as few instructions as possible write a JCoCo program that computes the sum of the first *n* integers where the non-negative *n* is read from the user.
5. Write a recursive JCoCo program that adds up the first *n* integers where *n* is read from the user. Remember, there must be a base case that comes first in this function and the recursive case must be called on something smaller which is used in computing the solution to the whole problem.
6. Write a Rational class that can be used to add and multiply fractions together. A Rational number has an integer numerator and denominator. The *__add__* method is needed to add together Rationals. The *__mul__* method is for multiplication. To get fractions in reduced format you may want to find the greatest common divisor of the numerator and the denominator when creating a Rational

number. Write this code in Python first, then disassemble it to get started with this assignment.
You may wish to write the greatest common divisor function *gcd* as part of the class although no self parameter is needed for this function. The greatest common divisor of two integers, x and y, can be defined recursively. If y is zero then x is the greatest common divisor. Otherwise, the greatest common divisor of x and y is equal to the greatest common divisor of y and the remainder x divided by y. Write a recursive function called *gcd* to determine the greatest common divisor of x and y.
Disassemble the program and then look for ways of shortening up the JCoCo assembly language program. Use the following main program in your code.

```
import disassembler

def main():
    x = Rational(1,2)
    y = Rational(2,3)
    print(x+y)
    print(x*y)
disassembler.disassemble(Rational)
disassembler.disassemble(main)
```

From this code you should get the following output which matches the output you should get had this been a Python program. Remember to use Python 3.2 when disassembling your program. And, remember to turn in as short a program as possible while getting this output below from the main program given above.

```
7/6
1/3
```

3.17 Solutions to Practice Problems

These are solutions to the practice problems. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

3.17.1 Solution to Practice Problem 3.1

The assembly code in Fig. 3.2 blindly pops the *None* at the end and then pushes *None* again before returning from main. This can be eliminated resulting in two fewer instructions. This would also mean that *None* is not needed in the constants, but this was not eliminated below.

```
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
          RETURN_VALUE
END
```

3.17.2 Solution to Practice Problem 3.2

As in practice 3.1 the *POP_TOP* and *LOAD_CONST* from the end can be eliminated. In the if-then-else code both the *then* part and the *else* part execute exactly the same *STORE_FAST* instruction. That can be moved after the if-then-else code and written just once, resulting in one less instruction and three less overall. Furthermore, if we move the *LOAD_GLOBAL* for the call to *print* before the if-then-else statement, we can avoid storing the maximum value in *z* at all and just leave the result on the top of the operand stack: either *x* or *y*. By leaving the bigger of *x* or *y* on the top of the stack, the call to *print* will print the correct value. This eliminates five instructions from the original code.

```
Function: main/0
Constants: None, 5, 6
Locals: x, y
Globals: print
BEGIN
          LOAD_CONST                     1
          STORE_FAST                     0
          LOAD_CONST                     2
```

```
                STORE_FAST                   1
                LOAD_GLOBAL                  0
                LOAD_FAST                    0
                LOAD_FAST                    1
                COMPARE_OP                   4
                POP_JUMP_IF_FALSE      label00
                LOAD_FAST                    0
                JUMP_FORWARD           label01
label00:        LOAD_FAST                    1
label01:        CALL_FUNCTION                1
                RETURN_VALUE
END
```

It is worth noting that the code above is exactly the disassembled code from this Python program.

```python
import disassembler
def main():
    x=5
    y=6
    print(x if x > y else y)

disassembler.disassemble(main)
```

When main is called, this code prints the result of a *conditional expression*. The if-then-else expression inside the print statement is different than an if-then-else statement. An if-then-else statement updates a variable or has some other side-effect. An if-then-else expression, or *conditional expression* as it is called in Python documentation, yields a value: either the *then* value or the *else* value. In the assembly language code we see that the yielded value is passed to the print function as its argument.

3.17.3 Solution to Practice Problem 3.3

```
Function: main/0
Constants: None, 5, 6
Locals: x, y
Globals: print
BEGIN
                LOAD_CONST                   1
                STORE_FAST                   0
                LOAD_CONST                   2
                STORE_FAST                   1
                LOAD_FAST                    0
                LOAD_FAST                    1
                COMPARE_OP                   1
                POP_JUMP_IF_TRUE       label00
                LOAD_GLOBAL                  0
                LOAD_FAST                    0
                CALL_FUNCTION                1
```

```
            POP_TOP
label00:    LOAD_GLOBAL                 0
            LOAD_FAST                   1
            CALL_FUNCTION               1
            RETURN_VALUE
END
```

3.17.4 Solution to Practice Problem 3.4

The following code behaves differently if the *BREAK_LOOP* instruction is removed from the program.

```
Function: main/0
Constants: None, 7, 6
Locals: x, y
Globals: print
BEGIN
            SETUP_LOOP            label01
            LOAD_CONST                  1
            STORE_FAST                  0
            LOAD_CONST                  2
            STORE_FAST                  1
            LOAD_FAST                   0
            LOAD_FAST                   1
            COMPARE_OP                  1
            POP_JUMP_IF_TRUE      label00
            BREAK_LOOP
            LOAD_GLOBAL                 0
            LOAD_FAST                   0
            CALL_FUNCTION               1
            POP_TOP
label00:    POP_BLOCK
label01:    LOAD_GLOBAL                 0
            LOAD_FAST                   1
            CALL_FUNCTION               1
            RETURN_VALUE
END
```

3.17.5 Solution to Practice Problem 3.5

This is the hello world program with exception handling used to raise and catch an exception. This solution does not include code for *finally* handling in case an exception happened while handling the exception. It also assumes the exception will match when thrown since JCoCo only supports one type of exception.

```
Function: main/0
Constants: None, "Hello World!"
Locals: ex
```

```
Globals: Exception, print
BEGIN
            SETUP_EXCEPT             label00
            LOAD_GLOBAL                    0
            LOAD_CONST                     1
            CALL_FUNCTION                  1
            RAISE_VARARGS                  1
            POP_BLOCK
            JUMP_FORWARD             label01
label00:    LOAD_GLOBAL                    1
            ROT_TWO
            CALL_FUNCTION                  1
            POP_TOP
            POP_EXCEPT
label01:    LOAD_CONST                     0
            RETURN_VALUE
END
```

3.17.6 Solution to Practice Problem 3.6

This program adds 5 and 6 together using the *__add__* magic method associated with integer objects. First 5 is loaded onto the operand stack. Then *LOAD_ATTR* is used to load the *__add__* of the 5 object onto the stack. This is the function. The argument to *__add__* is loaded next which is the 6. The 6 is loaded by the *LOAD_CONST* instruction. Then *__add__* is called with one argument. The 11 is left on the operand stack after the function call. It is stored in *x*, the *print* is loaded, *x* is loaded onto the operand stack, and *print* is called to print the value. Since *print* leaves *None* on the stack, that value is returned from the main function.

```
Function: main/0
Constants: None, 5, 6
Locals: x
Globals: __add__, print
BEGIN

            LOAD_CONST                     1
            LOAD_ATTR                      0
            LOAD_CONST                     2
            CALL_FUNCTION                  1
            STORE_FAST                     0
            LOAD_GLOBAL                    1
            LOAD_FAST                      0
            CALL_FUNCTION                  1
            RETURN_VALUE
END
```

3.17.7 Solution to Practice Problem 3.7

```
Function: main/0
Constants: None, "Enter a string: "
Locals: x, a
Globals: input, print
BEGIN
          LOAD_GLOBAL                0
          LOAD_CONST                 1
          CALL_FUNCTION              1
          STORE_FAST                 0
          SETUP_LOOP           label02
          LOAD_FAST                  0
          GET_ITER
label00:  FOR_ITER             label01
          STORE_FAST                 1
          LOAD_GLOBAL                1
          LOAD_FAST                  1
          CALL_FUNCTION              1
          POP_TOP
          JUMP_ABSOLUTE        label00
label01:  POP_BLOCK
label02:  LOAD_CONST                 0
          RETURN_VALUE
END
```

3.17.8 Solution to Practice Problem 3.8

A cell variable is needed if an inner function makes a modification to a variable that is located in the outer function. Consider the JCoCo program below. Without the cell the program below would print 10 to the screen and with the cell it prints 11. Why is that? Draw the run-time stack both ways to see what happens with and without the cell variable.

```
Function: f/1
    Function: g/1
    Constants: None, 1
    Locals: y
    FreeVars: x
    BEGIN
              LOAD_DEREF             0
              LOAD_CONST             1
              BINARY_ADD
              STORE_DEREF            0
              LOAD_DEREF             0
              LOAD_FAST              0
              BINARY_ADD
              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
          LOAD_DEREF                       0
          CALL_FUNCTION                    1
          LOAD_DEREF                       0
          BINARY_ADD
          RETURN_VALUE
END
Function: main/0
Constants: None, 3
Globals: print, f
BEGIN
          LOAD_GLOBAL                      0
          LOAD_GLOBAL                      1
          LOAD_CONST                       1
          CALL_FUNCTION                    1
          CALL_FUNCTION                    1
          POP_TOP
          LOAD_CONST                       0
          RETURN_VALUE
END
```

Interestingly, this program cannot be written in Python. The closest Python equivalent of this program is given below. However, it is not the equivalent of the program written above. In fact, the program below won't even execute. There is an error on the line *x = x + 1*. The problem is that as soon as Python sees *x* = in the function *g*, it decides there is another *x* that is a local variable in *g*. But, then *x = x + 1* results in an error because *x* in *g* has not yet been assigned a value.

```python
def f(x):
    def g(y):
        x=x+1
        return x + y
    return g(x) + x
def main():
    print(f(3))
main()
```

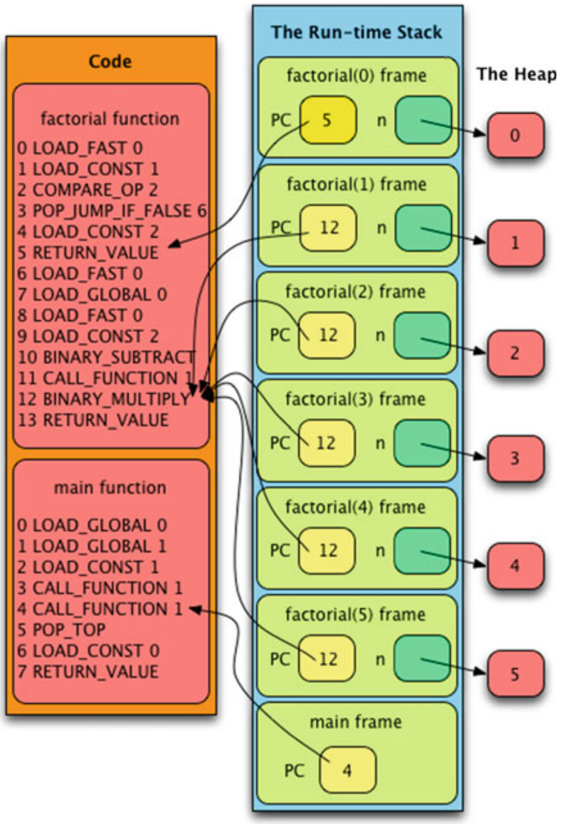

Fig. 3.20 Execution of fact.casm

3.17.9 Solution to Practice Problem 3.9

A couple things to notice in Fig. 3.20. The run-time stack contains one stack frame for every function call to factorial. Each of the stack frames, except the one for the *main* function, point at the *factorial* code. While there is only one copy of each function's code, there may be multiple stack frames executing the code. This happens when a function is recursive. There also multiple *n* values, one for each stack frame. Again this is expected in a recursive function.

3.17.10 Solution to Practice Problem 3.10

Python is a very transparent language. It turns out there is function called *dir* that can be used to print the attributes of an object which are the keys of its dictionary.

The dictionary maps names (i.e. strings) to the attributes of the object. The following strings map to their indicated values.

- *__init__* is mapped to the constructor code.
- *eat* is mapped to the eat method.
- *speak* is mapped to the speak method.
- *food* is mapped to an integer.
- This is all that is mapped by JCoCo. However, if you try this in Python you will discover that a number of other methods are mapped to default implementations of magic methods in Python including a hash method, comparison methods like equality (i.e. *__eq__*), a repr method, a str method, and a number of others.

3.17.11 Solution to Practice Problem 3.11

```
Class: Animal
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__/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: speak/1
    Constants: None, "is an animal"
    Locals: self
    Globals: print, name
    BEGIN
              LOAD_GLOBAL                   0
              LOAD_FAST                     0
              LOAD_ATTR                     1
              LOAD_CONST                    1
              CALL_FUNCTION                 2
              POP_TOP
              LOAD_CONST                    0
              RETURN_VALUE
    END
END
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
              LOAD_FAST                     0
              LOAD_ATTR                     1
              LOAD_CONST                    1
              CALL_FUNCTION                 2
              POP_TOP
              LOAD_CONST                    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
              LOAD_FAST                        0
              LOAD_ATTR                        1
              CALL_FUNCTION                    0
              POP_TOP
              LOAD_FAST                        0
              LOAD_ATTR                        2
              CALL_FUNCTION                    0
              POP_TOP
              LOAD_CONST                       0
              RETURN_VALUE
END
```

3.17.12 Solution to Practice Problem 3.12

To get ready to execute the built-in class builder function the stack must contain the following in order from the top of the stack down: The name of the class is on the top of the operand stack. Below the name is the closure of the class initializing function and its environment. Below that is the built-in class builder function itself. The *CALL_FUNCTION* instruction is executed with two arguments indicated to call the class builder.

Upon its return, the two arguments and the class builder function have been popped from the stack and the instance of the class is left on the operand stack. This class instance may then be stored as a reference from some known location, likely by a *STORE_FAST* instruction.