Python Intermediate Programming

Unit 5: Software Development

CHAPTER 2: PYTHON VIRTUAL MACHINE (PVM)

DR. ERIC CHOU

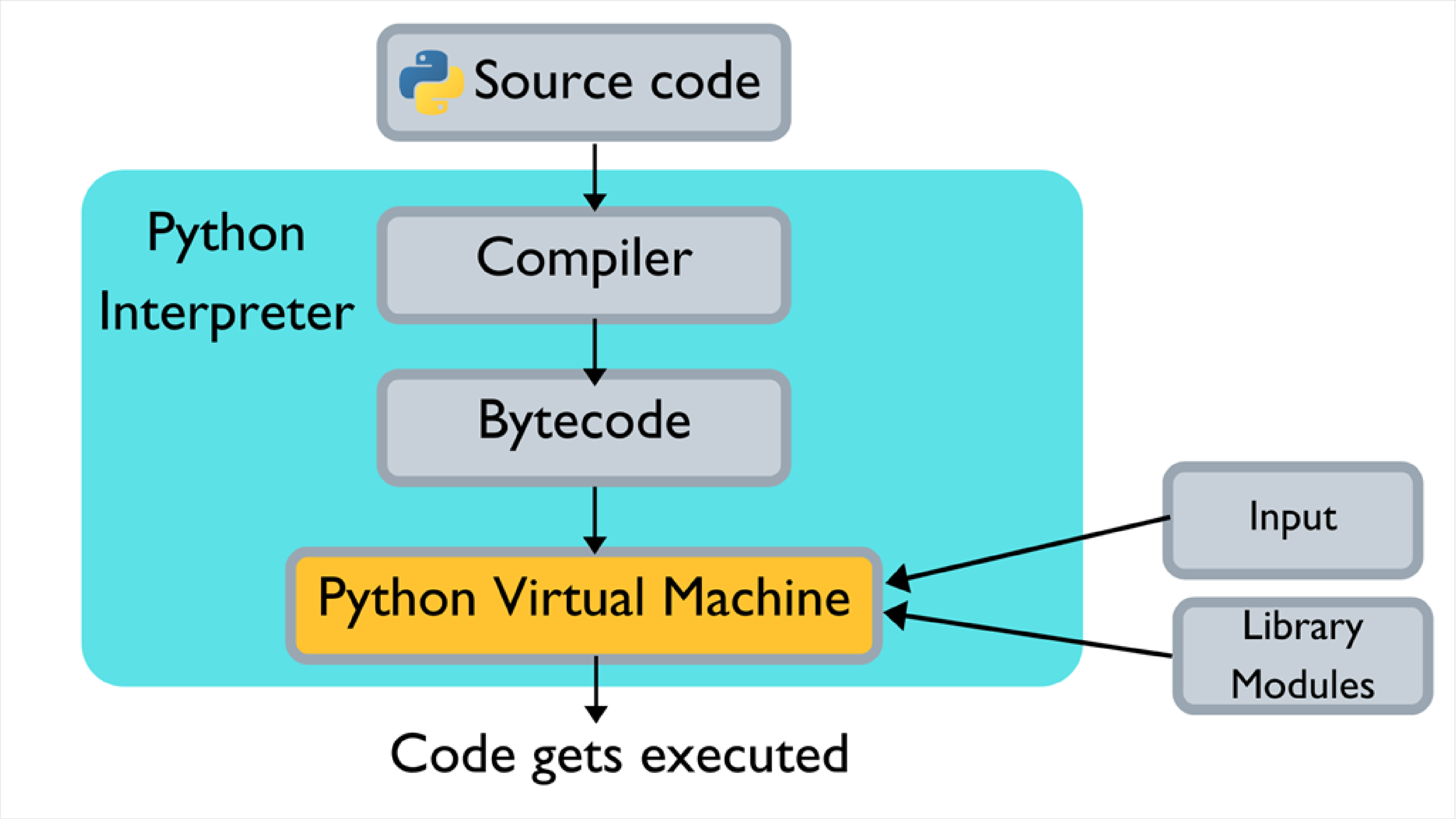
IEEE SENIOR MEMBER

Objectives

- Levels of abstraction are important in computing. We can look at a computer (and the software it runs) at many levels of abstraction: as quantum devices implementing logical components, as collections of these components implementing digital circuits (described by Boolean formulas), as digital circuits combined into an instruction set architecture (which we'll cover in today's lecture) as an architecture capable of running "translated" computer programs written in a high-level language, as large applications written in high-level programming languages, and as distributed/networked applications running on multiple machines communicating over a network.

Objectives

- It is difficult to hold all these levels of abstraction in our minds while we are exploring and creating computing systems/software. We often focus on just one level of abstraction for our jobs, while knowing some -but less- about the levels directly below and above. We have spent most of ICS 31-33 at the level of discussing computing via the high-level programming language Python. In this lecture we will look down one level, at the architecture into which Python programs are translated and run on: the **Python Virtual Machine.**


Source code
Python
Interpreter
Compiler
Bytecode
Python Virtual Machine
Input
Library
Modules
Code gets executed

Objectives

- Some programming languages (e.g., C++) are translated into instructions that run directly on hardware. We speak of compiling programs in that language onto a specific hardware architecture. Other programming languages (e.g., Java and Python) are translated to run on a common architecture: interpreters (or virtual machines) for that architecture can be written in a low-level language (like C, which targets most architectures) or in the machine's hardware language itself for maximal speed.
- Exactly, the same translated programs can run on any architecture that has the virtual machine program implemented on it.

Objectives

- The compiling approach has the advantage of producing programs that execute speedily on real machines. But it is a large undertaking to produce a good compiler for a new architecture, and compiling a program can take a long time.
- The virtual machine approach is much easier to port to new architectures (it is just one smallish program that must be written to run on the new architecture) but the extra layer of software (even as well as we know how to write interpreters) above the machine reduces performance, often by a factor 2-10 or more. So, there is no right way to write language translators: each approach comes with its own advantages and disadvantages, and each can be used/abused in situations.

Objectives

- In fact, hot-spot compilers do a bit of both. They profile programs while interpreting them to find their hots spots (small amounts of code that are executed frequently) and then they compile just those small pieces -all while running the program. Language translation is and has always been an important area in Computer Science. At UCI this topic is covered initially in COMPSCI 142A/B: Language Processor Construction. The computer architectures for real machines (into which compiled programs are compiled) are covered first in ICS-51.

Objectives

- In this lecture, we will discuss the code that Python is translated into and how this code is run on the Python Virtual Machine. There is no way to cover this topic fully in one lecture, so my goal is just to introduce this material and show you an interesting vertical slice through it. We will leverage off Python's **`dis.py`** module, whose **`dis`** function (dis means disassembly) shows us, in a readable form, the Virtual Machine instructions that Python functions, classes, and modules are translated into.
- At the very end of this lecture, we will return to analysis of algorithms by counting the instructions our Python functions are translated into.

Objectives

- Finally, see Section 31.12 in the Python Library documentation for many more details about today's lecture. Once you "get" the big picture, you might find it quite interesting to "dis" a variety of software components. I tried to find information about the Python Virtual Machine on the web, but it is pretty sparse.
- So, I put together this lecture based on general principles and the documentation I could find. There is much more information available for the Java virtual machine.

What do the python file extensions, .pyc .pyd .pyo stand for?

The .py, .pyc, .pyo and .pyd files have their own significance when it comes to executing python programs. The are used for –

- .py: The input source code that you've written.
- .pyc: The compiled bytecode. If you import a module, python will build a *.pyc file that contains the bytecode to make importing it again later easier (and faster).
- .pyo: A *.pyc file that was created while optimizations (-O) was on.
- .pyd: A windows dll file for Python.

http://docs.python.org/faq/windows.html#is-a-pyd-file-the-same-as-a-dll

Basics

LECTURE 1

Basic

- Every function object (what we are mostly concerned with in this lecture) is associated with 3 tuples:

  (1) its local variable names including parameters (in .__code__.co_varnames)

  (2) the global names it uses (in .__code__.co_names)

  (3) the constants it uses (in .__code__.co_consts)

- These tuples are built at the time Python defines the function and they are stored in the __code__ object associated with the function.

Basic

For example, if we define

```
def minimum(alist):
    m = None if len(alist) == 0 else alist[0]
    for v in alist[1:]:
        if v < m:
            m = v
    return m
```

Then,

```
minimum.__code__.co_varnames is ('alist', 'm', 'v')
minimum.__code__.co_names    is ('len', 'None')
minimum.__code__.co_consts   is (None, 0, 1)
```

Basic

- Load operations (e.g., LOAD_FAST, LOAD_GLOBAL, LOAD_CONST, which are all discussed in more detail below) are followed by an integer that indexes these tuples. So for example, in the function addup addup.__code__.co_varnames is the list ('alist', 'sum', 'v'), so the operation LOAD_FAST 0 loads/pushes onto the stack the value the name alist refers to.
- LOAD_FAST 1 loads/pushes onto the stack the value the name sum refers to. LOAD_FAST 2 loads/pushes onto the stack the value the name v refers to.

The Python Virtual Machine (PVM)

LECTURE 1

The Python Virtual Machine (PVM)

- The main data structure in the PVM is the "regular" stack (which is like a restricted a list, allowing only the operations push=append and pop=pop). A stack's primary operations are load/push and store/pop. We load/push a value on the top of an upwardly-growing stack (incrementing the stackp -stack pointer- that indexes the top); we store/pop a values from the top of a stack (decrementing the stackp).

The Python Virtual Machine (PVM)

- There is a secondarily important block stack that is used to store information about nested loops, try, and with statements. For example, a break statement is translated into code that uses the block stack to determine which loop to break out of (and how to continue executing at the first statement outside the loop). As loops, try/except, and with statements are started, information about their blocks are pushed onto the block stack; as they terminate, this information is popped off the bock stack. The block stack is too complicated for today's lecture and we do not needed to understand it: so, when we run across block stack instructions we will point out the fact that we are ignoring them.

The Python Virtual Machine (PVM)

- Here is an example of a simple sequence of stack operations to perform the calculation d = a+b*c, assuming that a, b, c, and d are local variables inside a function: assume co_varnames is ('a', 'b', 'c', 'd') and the actual values for these names are stored in a parallel list [1, 2, 3, None]: e.g., the value for 'a' is 1, the value for 'b' is 2, the value for 'c' is 3, and the value for 'd' is None. Generally the value for a name at index i in the co_varnames tuple is stored in index i in the list of actual values.

The Python Virtual Machine (PVM)

- As we will see in more detail below the meaning of

  LOAD_FAST N

  load/push onto the stack the value stored in co_varnames[N],
  written stackp += 1, stack[stackp] = co_varnames[N]

  STORE_FAST N

  store/pop the value on the top of the stack into co_varnames[N],
  written co_varnames[N] = stack[stackp], and stackp -= 1

  BINARY_MULTIPLY

  load/push onto the stack the * of the two values on the top,
  written stack[stackp-1] = stack[stackp-1] * stack[stackp]; stackp -= 1
  (turns the two values on the top of the stack into their product)

  BINARY_ADD

  load/push onto the stack the + of the two values on the top
  written stack[stackp-1] = stack[stackp-1] + stack[stackp]; stackp -= 1
  (turns the two values on the top of the stack into their sum)

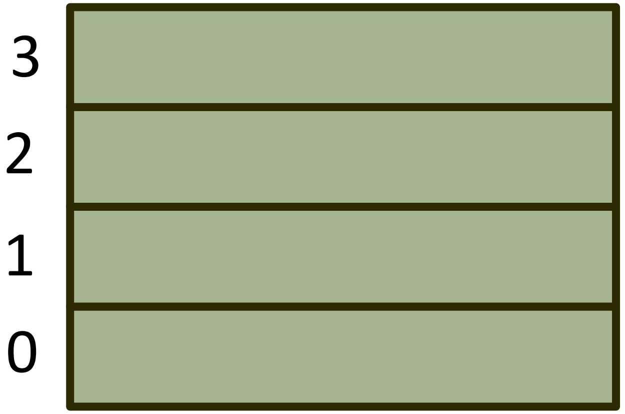
The Python Virtual Machine (PVM)

- The PVM code for `d = a+b*c` is

```
LOAD_FAST 0
LOAD_FAST 1
LOAD_FAST 2
BINARY_MULTIPLY
BINARY_ADD
STORE_FAST 3
```

The Python Virtual Machine (PVM)

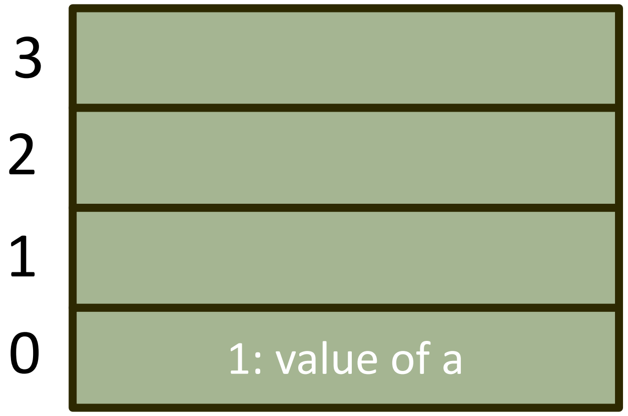
- Here is what happens step by step.
- Initially



stack (with `stackp = -1`, meaning is empty stack)

The Python Virtual Machine (PVM)

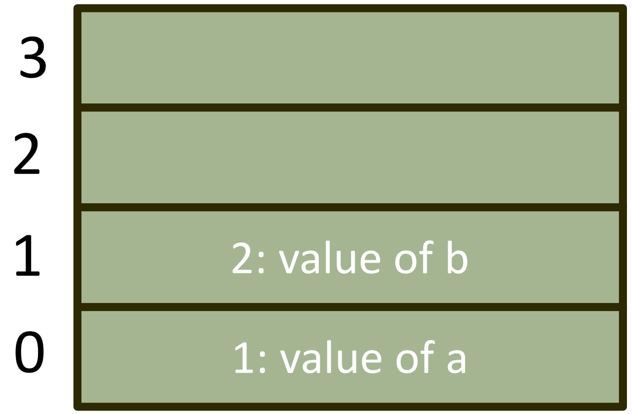
execute LOAD_FAST 0



stack (with **`stackp = 0`**)

The Python Virtual Machine (PVM)

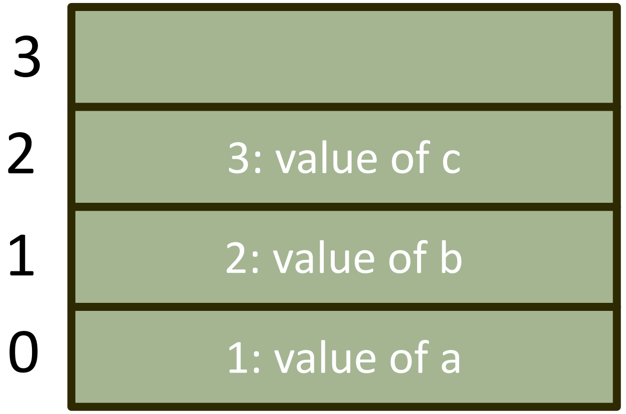
execute LOAD_FAST 1



stack (with **`stackp = 1`**)

The Python Virtual Machine (PVM)

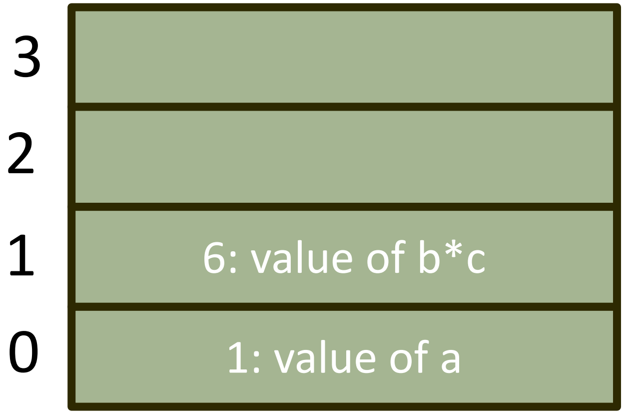
execute LOAD_FAST 2



stack (with **`stackp = 2`**)

The Python Virtual Machine (PVM)

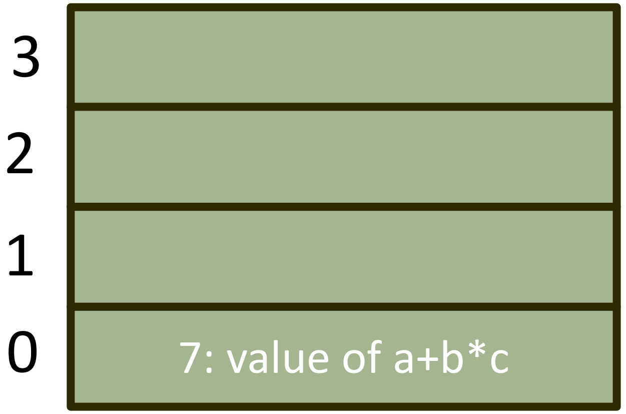
execute BINARY_MULTIPLY



stack (with `stackp = 1`)

The Python Virtual Machine (PVM)

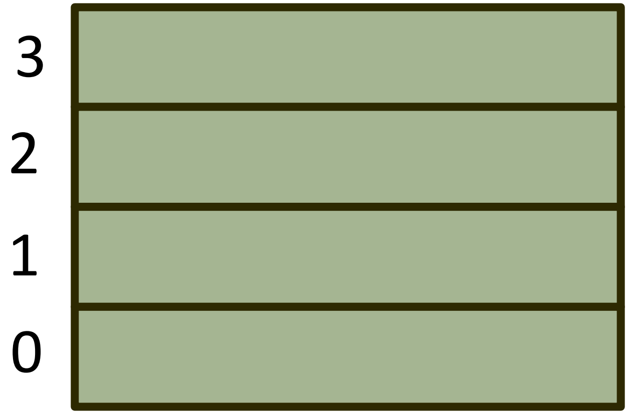
execute BINARY_ADD



stack (with **`stackp = 0`**)

The Python Virtual Machine (PVM)

execute STORE_FAST 3



stack (with **`stackp = -1`**)

The Python Virtual Machine (PVM)

- At this point d's value is 7, the value that was at the top of the stack when STORE_FAST was executed. The actual values for these names are stored in the list [1, 2, 3, 7].

The Python Virtual Machine (PVM)

- Problem: show (by drawing what I drew above) how the following instructions compute `d=(a+b)*c`

```
LOAD_FAST 0
LOAD_FAST 1
BINARY_ADD
LOAD_FAST 2
BINARY_MULTIPLY
STORE_FAST 3
```

- Any expression can be translated into similar code for the PVM to evaluate its value.

Control (the fetch/execute cycle) of the PVM

LECTURE 1

Control of the PVM

Control (the fetch/execute cycle) of the PVM

Each instruction (some of which are shown above) in the PVM consists of 1 or 3 bytes of information (a byte is 8 bits, and can represent numbers from 0 to 255). The first byte is the operation or byte code; the second two bytes are the operand for that byte code (but not all byte codes require operands:

LOAD_FAST does; BINARY_ADD doesn't). Two bytes (16 bits) can represent the unsigned numbers 0 to 65,535: when I first learned this fact, I conjectured that Python function cannot have more than 65,536 different local variable names (it turns out I was wrong; a student showed me a Python function that had more variables, and it still was runnable). Can you think how the student wrote such a large function and tested it?

Control of the PVM

- The instructions are stored in memory: think of memory too as a kind of list named m which stores sequential data in one location after the other. We can illustrate this by writing the instruction sequence above as follows:

Memory Location	Instruction
0	LOAD_FAST 0
3	LOAD_FAST 1
6	LOAD_FAST 2
9	BINARY_MULTIPLY
10	BINARY_ADD
11	STORE_FAST 3

- Just as the virtual machine has the name `stackp` for indexing the stack, it has a name pc (program counter) for indexing memory.

Control of the PVM

- Note that the first instruction is stored in m[0], and each subsequent instruction is stored in a location that is 3 higher (if the instruction has an explicit operand, as the load/store instructions do; most instructions have implicit operands: stack and pc) or 1 higher (if the instruction has no explicit operands, as the binary operator instructions do).
- Technically, the operation name (e.g., LOAD_FAST) represents a one byte value (an integer from 0 to 255). The next two bytes are typically the higher and lower bits in an integer between 0 and 2**16-1: 65,535.

Control of the PVM

- Once a program (instruction sequence) is loaded into memory, the PVM executes it according to the following simple rules. This control/execute cycle is what animates computers, allowing then to execute programs. It is fundamental in Computer Science. Here we assume pc is initially 0 (the index where the program starts).

(1) Fetch the operation and its operand (if present) starting at m[pc]

(2) pc += 3 (if operand is present) or pc += 1 (if no operand is present)

(3) Execute the operation code (mabe change its operand, stack, stackp, or pc)

(4) Go to step 1

- Some operations manipulate the stack/stackp and the lists that store values, others can change the pc (examples of such jump instructions appear in later sections, changing the locus of execution of the code).

Control of the PVM

So, when the pc is initially 0, the PVM executes the code above as follows

1. fetches the operation a m[0] and the operand at m[1] and m[2]
2. increments pc to 3
3. manipulates the stack (see above)
4. goes back to step 1

1. fetches the operation a m[3] and the operand at m[4] and m[5]
2. increments pc to 6
3. manipulates the stack (see above)
4. goes back to step 1

1. fetches the operation a m[6] and the operand at m[7] and m[8]
2. increments pc to 9
3. manipulates the stack (see above)
4. goes back to step 1

Control of the PVM

1. fetches the operation a m[9]: it has no operand
2. increments pc to 10
3. manipulates the stack (see above)
4. goes back to step 1

1. fetches the operation a m[10]: it has no operand
2. increments pc to 11
3. manipulates the stack (see above)
4. goes back to step 1

- At this point there is no more code to execute. In the next example we will see how the PVM executes more complicated code, specified by full Python functions.

The dis function in the dis module

LECTURE 1

The dis function in the dis module

- As described briefly in the introduction, we can print a symbolic/annotated description of any Python function (and module/class too; but in this lecture note we will stick with just functions) by using the dis function in the dis.py module. Here "dis" means "disassemble the code in the function object into a form that is readable by people. The dis function prints information in the console window (a better idea would be for it to return a string that could be printed or processed in other ways).

The dis function in the dis module

- Here is an example function (with line numbers) and the result dis.dis displays for it). All the operation codes and their meanings are covered in detail in the next section; we will look ahead at the relevant ones.

```
1 def addup(alist):
2     sum = 0
3     for v in alist:
4         sum = sum + v
5   return sum
```

The dis function in the dis module

Actually, I wrote the following simple function to show useful information about any function object (its name, its three tuples, and the dis information:) labelled.

```
def func_obj(fo):
    print(fo.__name__)
    print('  co_varnames:',fo.__code__.co_varnames)
    print('  co_names    :',fo.__code__.co_names)
    print('  co_consts   :',fo.__code__.co_consts,'\n')
    print('Source Line m operation/byte-code operand’+ \
          ’ (useful name/number)\n'+69*'-')
    dis.dis(fo)
```

calling func_obj(addup) prints

```
addup
  co_varnames: ('alist', 'sum', 'v')
  co_names   : ()
  co_consts  : (None, 0)
```

Source Line m	op/byte-code	operand (useful name/number)
2	0 LOAD_CONST 3 STORE_FAST	1(0) 2(sum)
3	6 SETUP_LOOP 9 LOAD_FAST 12 GET_ITER >> 13 FOR_ITER 16 STORE_FAST	24(to 33) 0(alist) 16(to 32) 2(v)
4	19 LOAD_FAST 22 LOAD_FAST 25 BINARY_ADD 26 STORE_FAST 29 JUMP_ABSOLUT >> 32 POP_BLOCK	1(sum) 2(v) 1(sum) 13
5	>> 33 LOAD_FAST 36 RETURN_VALUE	1(sum)

Note that any line prefaced by >> means that some other instruction in the function will jump to it (start executing code at it). Jumping in the PVM (by setting pc) is how loops and if statements in Python do their computation.

Here is a high-level description how this function executes. For more details, see the exact description of each instruction in the next section.

Line 2:

m[0]: loads the value 0 (co_consts[1]) on the stack
m[3]: stores the value 0 into sum (co_varnames[1])

Line 3:

m[6]: setup for the loop by pushing the size of the loop onto the block stack
(recall that we won't be doing anything with the block stack)
m[9]: loads the value of alist (co_varnames[0]) on the stack
m[12]: replaces its value on the stack by its iterator (by popping and pushing)
m[13]: loads the next iterator value on the stack (like next(),
jumping to m[32] - the pc + 16- if StopIteration is raised
(code in m[29] jumps back to this location to make the code loop)
m[16]: stores the next value into v (co_varnames[2]), popping it off the stack

Line 4:

m[19]: loads the value of sum (co_varnames[1]) on the stack
m[22]: loads the value of v (co_varnames[2]) on the stack
m[25]: removes from stack/adds two values, pushes the result on the stack
m[26]: stores the value into sum (co_varnames[1]), popping it off the stack
m[29]: sets pc to 13, so the next instruction executed in at m[13]
 (jumps back to a previous location to make the code loop)
m[32]: pops what m[6] pushed onto the block stack
 (recall that we won't be doing anything with the block stack)
 (code in m[13] jumps here, on StopIteration, terminating the loop)

Line 5:

m[33]: load the value of sum (co_varnames[1]) on the stack to return
m[36]: return from the function with the result on the top of the stack

Operation/Byte Codes

LECTURE 1

Operation/Byte Codes

- Below is a list of many **important operations** and how they manipulate the PVM. The complete list is available in section 31.12 of the Python documentation. Recall that many operations manipulate stack, stackp, and pc.

- **Loading/Storing**

  LOAD_CONST N
    stackp += 1, stack[stackp] = co_consts[N]
  LOAD_FAST N
    stackp += 1, stack[stackp] = co_varnames[N]
  LOAD_GLOBAL N
    stackp += 1, stack[stackp] = co_names[N]
  STORE_CONST N
    co_consts[N] = stack[stackp], and stackp -= 1
  STORE_FAST N
    co_varnames[N] = stack[stackp], and stackp -= 1
  STORE_GLOBAL N
    co_names[N] = stack[stackp], and stackp -= 1

Operation/Byte Codes

- There are general Load and Store operations that look up names based on the LEGB rules, if Python is unsure where these names will be found. Generally, it uses the operations above.

Operators

UNARY_POSITIVE

```
stack[stackp] = +stack[stackp]
```

UNARY_NEGATIVE

```
stack[stackp] = -stack[stackp]
```

UNARY_NOT

```
stack[stackp] = not stack[stackp]
```

UNARY_INVERT

```
stack[stackp] = ~stack[stackp]
```

Operators

BINARY_ADD

stack[stackp-1] = stack[stackp-1] + stack[stackp]; stackp -= 1

BINARY_SUBTRACT

stack[stackp-1] = stack[stackp-1] - stack[stackp]; stackp -= 1

BINARY_MULTIPLY

stack[stackp-1] = stack[stackp-1] * stack[stackp]; stackp -= 1

BINARY_TRUE_DIVIDE

stack[stackp-1] = stack[stackp-1] / stack[stackp]; stackp -= 1

BINARY_FLOOR_DIVIDE

stack[stackp-1] = stack[stackp-1] // stack[stackp]; stackp -= 1

BINARY_MODULO

stack[stackp-1] = stack[stackp-1] % stack[stackp]; stackp -= 1

BINARY_POWER

stack[stackp-1] = stack[stackp-1] ** stack[stackp]; stackp -= 1

Operators

BINARY_SUBSCR (indexing)

```
stack[stackp-1] = stack[stackp-1] [ stack[stackp] ]; stackp -= 1
```

BINARY_LSHIFT

```
stack[stackp-1] = stack[stackp-1] << stack[stackp]; stackp -= 1
```

BINARY_RSHIFT

```
stack[stackp-1] = stack[stackp-1] >> stack[stackp]; stackp -= 1
```

BINARY_AND

```
stack[stackp-1] = stack[stackp-1] & stack[stackp]; stackp -= 1
```

BINARY_OR

```
stack[stackp-1] = stack[stackp-1] | stack[stackp]; stackp -= 1
```

BINARY_XOR

```
stack[stackp-1] = stack[stackp-1] ^ stack[stackp]; stackp -= 1
```

There are in-place versions of the binary operators: e.g., x += 1 vs x = x + 1
I will show the meanin of INPLACE_ADD and just list the others here:

```
INPLACE_ADD
  stack[stackp-1] += stack[stackp]; stackp -= 1
```

INPLACE_SUBTRACT, INPLACE_MULTIPLY, INPLACE_FLOOR_DIVIDE, INPLACE_TRUE_DIVIDE, INPLACE_MODULO, INPLACE_POWER, INPLACE_LSHIFT, INPLACE_RSHIFT, INPLACE_AND, INPLACE_XOR, and INPLACE_OR

Iteration:

```
GET_ITER
  stack[stackp] = iter(stack[stackp])
FOR_ITER N
  stackp += 1; stack[stackp] = next(stack[stackp-1])
    but if StopIteration exception is raised in part 2: pc += N
```

Jumping (changing the pc from where the next instruction is fetched):

JUMP_ABSOLUTE N
pc = N
JUMP_FORWARD N
pc += N
POP_JUMP_IF_TRUE N
if stack[stackp] is True, pc = N (always stackp -= 1)
POP_JUMP_IF_FALSE N
if stack[stackp] is False, pc = N (always stackp -= 1)

Calling Functions and Returning/Yielding

CALL_FUNCTION N
The first operand byte is a count of the position arguments: pcount
The second operand byte is a count of the keyword arguments: kcount
There are kcount name indexes (for parameter names) and values on the top of the stack followed by pcount values followed by the function to call
This operation pops all function arguments off the stack to store them into the co_varnames tuple, and pops off the the function itself
The function should leave its answer on the top of the stack

RETURN_VALUE

Return to the location that called this function (its retuned answer on the top of the stack)

YIELD_VALUE

In the Library Reference, they use the notation TOS to mean the location at the top of the stack, and TOSn to mean n down from the top of the stack. So TOS is our stack[stackp] and TOS1 is our stack[stackp-1]

More functions

LECTURE 1

More functions

- It contains both a conditional/if statement (lines 4-5) and a conditional expression (line 6), which translate into a variety of jump instructions. Also note how the tuple assignment in line 2 is translated via the UNPACK_SEQUENCE N instruction (which takes any sequence of values (tuple or list) and pushes N of them onto the stack right to left: so if (0,1) is on the stack, UNPACK_SEQUENCE 2 pops this value off the stack, pushing first 1 then 0 onto the stack (which is why the first value below is stored into sum and the second is stored into count).

```python
def average_positive(alist):
    sum,count = 0,0
    for v in alist:
        if v > 0:
            sum = sum + v
            count += 1
    return sum / (1 if count == 0 else count)
```

average_positive

```
co_varnames: ('alist', 'sum', 'count', 'v')
co_names   : ()
co_consts  : (None, 0, 1, (0, 0))
```

```
Source Line   m  operation/byte-code      operand (useful name/number)
---------------------------------------------------------------------
 2              0 LOAD_CONST                3 ((0, 0))
                3 UNPACK_SEQUENCE           2
                6 STORE_FAST                1 (sum)
                9 STORE_FAST                2 (count)

 3             12 SETUP_LOOP               49 (to 64)
               15 LOAD_FAST                 0 (alist)
               18 GET_ITER
          >>   19 FOR_ITER                 41 (to 63)
               22 STORE_FAST                3 (v)

 4             25 LOAD_FAST                 3 (v)
               28 LOAD_CONST                1 (0)
               31 COMPARE_OP                4 (>)
               34 POP_JUMP_IF_FALSE        19

 5             37 LOAD_FAST                 1 (sum)
               40 LOAD_FAST                 3 (v)
               43 BINARY_ADD
               44 STORE_FAST                1 (sum)
```

```
  6             47 LOAD_FAST                2 (count)
                50 LOAD_CONST               2 (1)
                53 INPLACE_ADD
                54 STORE_FAST               2 (count)
                57 JUMP_ABSOLUTE           19
                60 JUMP_ABSOLUTE           19
           >>   63 POP_BLOCK

  7        >>   64 LOAD_FAST                1 (sum)
                67 LOAD_FAST                2 (count)
                70 LOAD_CONST               1 (0)
                73 COMPARE_OP               2 (==)
                76 POP_JUMP_IF_FALSE       85
                79 LOAD_CONST               2 (1)
                82 JUMP_FORWARD             3 (to 88)
           >>   85 LOAD_FAST                2 (count)
           >>   88 BINARY_TRUE_DIVIDE
                89 RETURN_VALUE
```

More functions

- If there were an else: block, it would appear between 57/60: both jump back to m[19] when their blocks finish executing.
- Please feel free to type in all sorts of small functions to see how they are translated to run on the PVM. The project folder that you can download includes some simple functions and the func_obj function.

More functions

- Note that the two simple functions shown did not call functions on their inside: see addup1 in the project folder, which uses range and len to iterate of the list indexes to compute the sum. The relevant line in the function is

```
for i in range(len(alist)):
```

- The local/global names are

```
co_varnames: ('alist', 'sum', 'i')
co_names    : ('range', 'len')
```

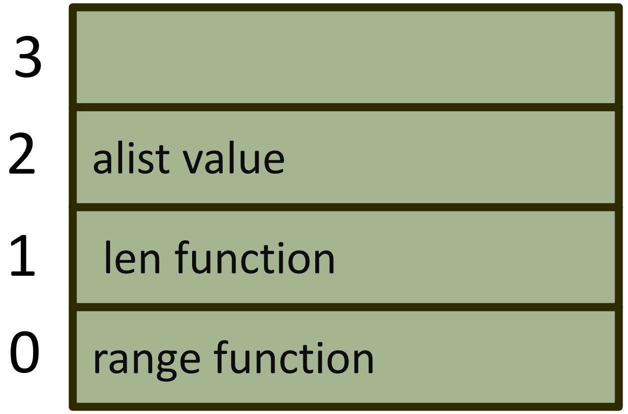
More functions

and sequence of instructions is

```
               9 LOAD_GLOBAL              0 (range)
              12 LOAD_GLOBAL              1 (len)
              15 LOAD_FAST                0 (alist)
              18 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              21 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              24 GET_ITER
         >>   25 FOR_ITER                20 (to 48)
              28 STORE_FAST               2 (i)
```

More Functions

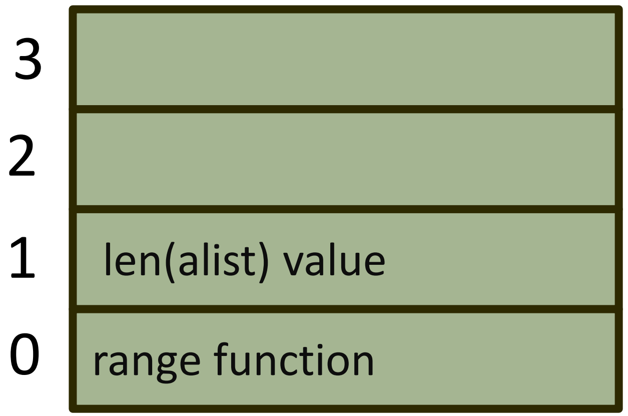
Here the range function, to call last, is loaded/pushed on the stack; then the **`len`** function, to call first, is loaded/pushed on the stack; then the alist variable is pushed on the stack. The stack looks as follows



stack (with **`stackp = 2`**)

More Functions

The first CALL_FUNCTION 1 says using 1 positional argument (stackp=2), call the function specified at stackp-1 (len) on the stack, leaving the answer on the stack. The stack looks as follows



stack (with **`stackp = 1`**)

More Functions

The second CALL_FUNCTION 1 says using 1 positional argument (stackp=1), call the function specified at stackp-1 (range) on the stack, leaving the answer on the stack. The stack looks as follows

`The range iterator`

stack (with **`stackp = 1`**)

Then GET_ITER is called to do the iteration, which uses FOR_ITER to produce the first value (which is stored in local variable i).

Return to Analysis of Algorithms

LECTURE 1

Return to Analysis of Algorithms

- We can use the result of dis to compute the worst-case count of the number of instructions executed in a function. Let's return to (and review) the `addup` function for our analysis.

```
def addup(alist):
    sum = 0
    for v in alist:
        sum = sum + v
    return sum
```

addup

```
co_varnames: ('alist', 'sum', 'v')
co_names    : ()
co_consts   : (None, 0)
```

```
Source Line  m  op/byte-code                 operand (useful name/number)
----------------------------------------------------------------------
  2             0 LOAD_CONST                   1 (0)
                3 STORE_FAST                   1 (sum)

  3             6 SETUP_LOOP                  24 (to 33)
                9 LOAD_FAST                    0 (alist)
               12 GET_ITER
          >>   13 FOR_ITER                    16 (to 32)
               16 STORE_FAST                   2 (v)

  4            19 LOAD_FAST                    1 (sum)
               22 LOAD_FAST                    2 (v)
               25 BINARY_ADD
               26 STORE_FAST                   1 (sum)
               29 JUMP_ABSOLUTE               13
          >>   32 POP_BLOCK

  5       >>   33 LOAD_FAST                    1 (sum)
               36 RETURN_VALUE
```

Return to Analysis of Algorithms

Here is how to account for the number of instructions executed when Python runs this function. As always, we will assume that the len(alist) is N. Here there are no conditional statement so the worst case always executes all the code in the function (and the loop N times).

Instructions done once (not in the loop proper)

2 for line 2, code before the loop: initialize sum

3 for line 3, setup for the loop; not repeated in the loop proper at m[13]

1 for line 4, at m[32], jumped to when `StopIteration` exception is raised

2 for line 5, code after the loop: load/push sum on the stack and return

Instructions done in the loop

2 for line 3, m[13] and m[16]; note loop back to m[13]

5 for line 4, sum = sum + v and jump back to m[13] to get next iterator value

Return to Analysis of Algorithms

- Finally, note that the instruction at m[13] is executed N+1 times: N times it continues in the loop body and 1 time it raises **StopIteration** and jumps to m[32].
- So, the I(N) is 8 (instructions done once) + 7N (instructions done in the loop) + 1 (m[13] done on N+1 iteration raising **StopInteration**) instructions.
- I(N) = 7N + 9 instructions