Lecture #13: Review
Computational Concepts Toolbox

- Data type: values, literals, operations,
- Expressions, Call expression
- Variables
- Assignment Statement, Tuple assignment
- Sequences: tuple, list
- Dictionaries
- Function Definition Statement
- Conditional Statement
- Iteration: list comp, for, while
- Lambda function expr.

- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values

Higher order function patterns
  - Map, Filter, Reduce

Function factories – create and return functions
- Recursion
- Abstract Data Types
- Mutation
- Class & Inheritance
- Exceptions
- Iterators & Generators
The Goal of CS88 ...

Idea

Expressed in a clear computational form

Understood by others

Interpreted by a “machine”

Answers and Insights
You will use this understanding in many situations that are not .py files and notebooks
SQL Review

**SELECT**

[ALL or DISTINCT] expressions over columns (map/reduce), optionally **AS** names

**FROM**

specification of table or join of tables

**WHERE**

conditional expression specifying rows in cols of tables

**GROUP BY**

aggregation expression defining collections of rows in filtered cols of tables

**ORDER BY**

expression on rows of filter cols defining order of result
How would you write a Python interpreter?

Computational Concepts Toolbox

- Data type: values, literals, operations,
- Expressions, Call expression
- Variables
- Assignment Statement, Tuple assignment
- Sequences: tuple, list
- Dictionaries
- Function Definition Statement
- Conditional Statement
- Iteration: list comp, for, while
- Lambda function expr.

- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
  - Function factories – create and return functions
- Recursion
- Abstract Data Types
- Mutation
- Class & Inheritance
- Exceptions
- Iterators & Generators
What do you give to the interpreter?

• An Expression
• A sequence of Statements
• optionally followed by an expression
Basic Process

- Parse the input into logical pieces

- Expression
  - Value or variable (leaves) – of a “type”
  - Tree of operators and operand expressions
    » .. * .. , .. + .. , ...
    » .. ( .. ) , [ .. ], lambda .. : .. , ...
  - Comprehensions

- Sequence of statements
  - assignment
  - def
  - conditional
  - iteration
Values

• **Primitive Value**
  – int, float, boolean

• **Complex Values**
  – string, tuple, list, dict,
  – function, class
  – object, method

• **Variable**
  – Reference to a value
At the bottom it’s a bunch of bits

- How many distinct things represented in $N$ bits?
  - $2^N$ - Think recursively
    - 2 “things” in 1 bit – $\{0,1\}$
    - Assume $2^{N-1}$ things in $N$-1 bits
      - $0 \parallel \{0, \ldots, 2^{N-1} – 1\} \cup 1 \parallel \{0, \ldots, 2^{N-1} – 1\}$
  - “word” is now (typically) 64 bits
    - Can represent $2^{64}$ (over 18 quintillion or $1.8 \times 10^{19}$) different values

- Addresses (unsigned ints): $0 .. 2^N – 1$
- Signed Integers: $-2^{N-1} .. 2^{N-1} – 1$
- IEEE Float Point: $-1^S \times 1.f \times 2^{e-1023}$
Variable

• Starting with current frame
• Look up variable in frame
• If not present, try parent frame, repeatedly
• Until global frame is reached
• If not found there
• Raise an exception
Variable

- Starting with current frame
- Look up variable in frame
- If not present, try parent frame, repeatedly
- Until global frame is reached
- If not found there
- Raise an exception

```python
def f(a):
    x = a-1
    def g(b):
        q = x+a+b-z
        return q
    if a > 1:
        f(a-1)
    else:
        g(a*5)
z = 2
f(z)
```

Frames

- Global frame
  - f
  - z 2
  - g(b) [parent=f1]
  - f1: f
    - a 2
    - g
    - x 1
  - f2: f
    - a 1
    - g
    - x 0
  - g [parent=f2]
    - b 5
    - q 4
Data Structures

Python 2.7

1. \( a = 3.1415 \)
2. \( x = (1, 2) \)
3. \( y = [3, 4, 5] \)
4. \( z = \{ 'a': 6, 'b': 7 \} \)
5. \( x[0] + y[1] + z[ 'b' ] * a \)

Frames

Global frame:
- \( a = 3.1415 \)
- \( x \)
- \( y \)
- \( z \)

Objects

- Tuple: \( 0, 1, 2 \)
- List: \( 3, 1, 4, 2 \)
- Dictionary: "a": 6, "b": 7

Edit code

Program terminated
Operators

- Evaluate the operand expressions (recursively)
- Check the types of the resulting values to determine the operator for symbol
- If no valid combination, raise exceptions
- Apply operator to resulting values to produce result
Call Expressions

• Evaluate the operand expressions (recursively)
• Evaluate “function” expression to get function to apply
• This may involve function return values or “.” or ...
• Check that it is of function type
• If not, raise exception
• Apply function to resulting values to produce result
Built-in Data Structure Constructor

(     ,     , ... )   {     :     , ... }

- Evaluate each of the index and value expressions
  - Or raise error
- Allocate storage to hold the data structure
- Fill in values at indices/Key
- Return a reference to the object
Comprehension Expression

\[
\left[ \text{ for } <\text{var tuple}> \text{ in } i\text{-exp} \right]
\]

- Evaluate iterable expression
- For each element in iteration
- Bind var tuple to value tuple
- Evaluate with each of those variable bindings
- Construct resulting object and return reference to it
Lambda Expression

\[
\text{lambda } <\text{vars}> : \text{ }\
\]

- Construct a function object that evaluates expression in a frame with variables in \(<\text{vars}>\) bound to argument values and returning the result
- Return reference to the function object

Python 2.7

```python
  1  a = lambda x,y : x*y
  2  a(1,2)
```

Frames

<table>
<thead>
<tr>
<th>Global frame</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>function (\lambda(x, y) &lt;\text{line 1}&gt;)</td>
</tr>
</tbody>
</table>

Objects

<table>
<thead>
<tr>
<th>(\lambda &lt;\text{line 1}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
</tbody>
</table>

Return value

2
Assignment Statement

• Evaluate RHS expression to get value
  – Or raise and exception
• Locate LHS variable(s) in frame path
• For each variable
• If exists, set variable to expression value
• If not, create variable of name(s) <var list> in current frame

\[
<\text{var list}> =
\]

\[
x = 3 \\
y = x + 4 \\
\]

\[
a, b = 3, a+4
\]
Set operation

- Evaluate RHS to get value
  - Or raise and exception
- Evaluation LHS expressions to get object and index/key
  - Or raise exception
- Set obj [ key ] to expression value
Define Statement

```python
def <fun name> ( <var list> ) :
    <suite of statements>
```

- Construct a function object to evaluate `<suite of statements>` in a frame with `<var list>` as local variable bound to argument expressions
- `return` statements evaluate expression in current frame and return it as result of the call expression
- Introduce `<fun name>` into current frame, assigned a reference to the function object
Define

Python 2.7

1  def foo(x):
2     return x+1
3  def bar(f, y):
4     def cal(z):
5         return z+f(y)
6  bar(foo, 3)(4)

Frames

Global frame

foo

bar

Objects

function foo(x)

function bar(f, y)

function cal(z) [parent=f1]

f1: bar

f

y  3

cal

Return value

None

Edit code

Step 6 of 6

04/29/18
Control Flow
Sequence of Statements

• Evaluate each statement in sequence
• Introducing new variables up updating objects with each
Conditional Statement

if :

< true suite of statements >
else:
<false suite>

• Evaluate
• If it yields a truthy result, evaluate <true suite>
• Otherwise, if else: present, evaluate <false suite>
Call Expressions

- Evaluate the operand expressions (recursively)
- Evaluate “function” expression to get function to apply
- This may involve function return values or “.” or ...
- Check that it is of function type
- If not, raise exception
- Apply function to resulting values to produce result

Evaluate the statements within the function body
Functions plus conditionals ...

- Recursion

```python
    def fib(n):
        if n <= 2:
            return 1
        else:
            return fib(n-2)+fib(n-1)
    fib(4)
```

Frames:
- Global frame:
  - fib

Objects:
- function fib(n)
- fib
  - n: 4
  - n: 3
  - n: 2
- Return value: 1
While Statement

```python
while <condition>:
    < suite of statements >
else:
    < exit suite>
```

- Repeatedly evaluate `<condition>`
- If it yields a truthy result, evaluate `<suite>`
- Otherwise, if else: is present evaluate `<exit suite>`
  - `continue` skips remain statements in suite
  - `break` exits loop skipping `<exit suite>`
For Statement

for <var list> in <cloud> :
  < suite of statements >
else:
  < exit suite >

• Evaluate <cloud> to get an iterable
• Repeatedly bind <var list> to next
• Evaluate <suite> with these bindings
• Until StopIteration is raised
• if else: is present evaluate <exit suite>
Try statement

try:
    < suite of statements >
except <var>:

< except suite>

• Evaluate suite of statements
• If exception is raised which matches
• Evaluate except suite is var bound to exception object
class <classname> ( <inheritance> ):

< suite of statements >

- If present, evaluate the inheritance list to obtain a class object or class type.
- Create new namespace for classname
- Evaluate <suite> in a new execution frame using a newly created namespace and global namespace
  - Typically sequence of define statements
- self in define for methods, self otherwise for object attributes
- vars in class namespace for class attributes
- Return resulting class object
. operator

References `<var>` in namespace of
with statement

with  as <var> [, more ] :

< suite of statements >

• Evaluate suite of statements with vars bound to results of corresponding
Comprehension expressions

\[
\begin{align*}
\text{[ for <var list> in } \text{ ]} \\
\text{[ for <var list> in } \text{ if } \text{ ]}
\end{align*}
\]

- Iteratively,
  - Evaluate next
  - If present, evaluate on it
  - Evaluate
- until stop_iter exception
  - Collect all resulting values into result object
Software Design Patterns

• Higher Order Functions
• Recursion
• Data Parallel – Map-Reduce
• Abstract Data Types
  – Constructors, Selectors, Actions
• Object Oriented Programming
  – Encapsulation of behavior
• Iterators and Generators
  – Classes with __iter__ and __next__
  – yield statement
Uses of Computational Thinking

- Computational concepts model the world. Programming languages are mathematical formalisms just like any other: linear algebra, differential equations, statistics...

- Plus: Automatic verification of the model.

More CS:
- CS61b: More programming
- CS61c: Machine architecture (how the bits are moved)

So now …
Go model and change the world …