Final Exam Review

CS 1110
Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]

Announcements

• No post-lecture office hours today
• Study Guide is published
• Extra review sessions happening
• Final Exam is Sunday, May 15

Where and When is your Exam?

• Check on Canvas
  - Final Exam Date & Time Assignments
  - Pretty much everyone is taking it in Barton
  - Only a few exceptions
  - Extended Time Exam Accommodations

• Closed Notes & Book, Reference Sheet
• Bring your Cornell ID

Expressions

An expression **represents** something

- Python **evaluates it** (turns it into a value)
- Similar to a calculator

Examples:

- 2.3
- (3 * 7 + 2) * 0.1

Types

**Type: set of values & operations on them**
Meaning of operations depends on type

Type **float:**
- Values: real numbers
- Ops: +, -, *, //, **, %

Type **int:**
- Values: integers
- Ops: +, -, *, //, %, **

Type **bool:**
- Values: True, False
- Ops: not, and, or

Variable Assignment

**Example:**

\[ x = 4 + 1 \]

Expression evaluates to 5

variable

equals sign
(just one!)

An **assignment statement:**
- takes an **expression**
- evaluates it, and
- stores the **value** in a **variable**
In More Detail: Variables

- **A variable**
  - is a **named** memory location (box)
  - contains a **value** (in the box)

- **Examples:**
  - Variable `x`, with value 5 (of type `int`)
  - Variable `area`, w/ value 20.1 (of type `float`)

Expressions vs. Statements

- **Expression**
  - Represents something
  - Python **evaluates** it
  - End result is a value
  - **Examples:**
    - 2.3
    - (3+6)/4
    - `x == 5`

- **Statement**
  - Does something
  - Python **executes** it
  - Need not result in a value
  - **Examples:**
    - `x = 2 + 1`
    - `x = 5`

Executing an Assignment Statement

The command: \[ x = 3.0 \times x + 1.0 \]

"Executing the command":
1. **Evaluate** right hand side \( 3.0 \times x + 1.0 \)
2. **Store** the value in the variable `x`'s box

- Requires both evaluate AND store steps
- Critical mental model for learning Python

Function Calls

- Function calls have the form:
  - `best_function_ever(x,y,...)`
  - **Function name**
  - **Argument(s)**

- Arguments
  - Separated by commas
  - Can be any expression

A function might have 0, 1, … or many arguments

Modules: Libraries vs. Scripts

- **Library**
  - Provides functions, variables
  - `import` it into Python shell, don’t include ".py"
  - Within Python shell you have access to the functions and variables of the imported module

- **Script**
  - Behaves like an application
  - At command line prompt, tell python to run the file (use full filename, including ".py")
  - After running the app you’re back at the command line

Files look the same. Difference is how you use them.

Visualizing functions & variables

**Running Example:**
1. Built-in functions
2. Define a new variable
3. Import a module
4. Use a module variable

C:\> python
>>> `x` = 7
>>> `import` `math`
>>> `x` = `math.pi`
Understanding How Functions Work

- We draw pictures to show what is in memory
- **Call Frame**: representation of function call
  - Line number of the next statement in the function body to execute
  - Starts with 1st statement in function body
  - Draw parameters as variables (named boxes)

Not just a pretty picture!
The information in this picture depicts exactly what is stored in memory on your computer.

A Precondition Is a Contract

- If precondition is met, the function will work!
- If precondition is **not** met... no guarantees!

Representative Tests

- Cannot test all inputs
  - “Infinite” possibilities
- Limit ourselves to tests that are **representative**
  - Each test is a significantly different input
  - Every possible input is similar to one chosen
- An art, not a science
  - If easy, never have bugs
  - Learn with much practice

Objects: Organizing Data in Folders

- An object is like a manila folder
- It contains other variables
  - Variables are called **attributes**
  - These values can change
- It has an ID that identifies it
  - Unique number assigned by Python (just like a NetID for a Cornellian)
  - Cannot ever change
  - Has no meaning; only identifies

Storage in Python

- **Global Space**
  - What you “start with”
  - Stores global variables
  - Lasts until you quit Python
- **Heap Space**
  - Where “folders” are stored
  - Have to access indirectly
- **Call Stack** (with Frames)
  - Parameters
  - Other variables local to function
  - Lasts until function returns
**Methods**: a special kind of function

Methods are:
- Defined for specific classes
- Called using objects of that class

**Example**:
```python
>>> import shapes
>>> u = shapes.Point3(4, 2, 3)
>>> u.greet()
"Hi! I am a 3-dimensional point located at (4,2,3)"
```

**Built-in Types vs. Classes**

<table>
<thead>
<tr>
<th>Built-in types</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Built-into Python</td>
<td>• Provided by modules</td>
</tr>
<tr>
<td>• Refer to instances as values</td>
<td>• Refer to instances as objects</td>
</tr>
<tr>
<td>• Instantiate with simple assignment statement</td>
<td>• Instantiate with assignment statement with a constructor</td>
</tr>
<tr>
<td>• Can ignore the folders</td>
<td>• Must represent with folders</td>
</tr>
</tbody>
</table>

**Classes are user-defined Types**

Defining new classes = adding new types to Python

**Example Classes**
- Point3
- Rect
- Freq (A3), for word frequencies
- Doll (class, lab)
- Song, Mix (A4)

**Classes Have Folders Too**

<table>
<thead>
<tr>
<th>Object Folders</th>
<th>Class Folders</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Separate for each instance</td>
<td>• Data common to all instances</td>
</tr>
<tr>
<td>Example: 2 Student objects</td>
<td></td>
</tr>
</tbody>
</table>

**Object Methods**

- Attributes live in object folder
- Class Attributes live in class folder
- Methods live in class folder

**Evaluating a Constructor Expression**

1. Constructor creates a new object (folder) of the class Course on the Heap
   - Folder is initially empty
   - Has id
2. Constructor calls `__init__` (self, "CS 1110", 4)
   - `self` = identifier ("Fill this folder!")
   - Other args come from the constructor call
   - commands in `__init__` populate folder
   - `__init__` has no return value! ("I filled it!")
3. Constructor returns the id
4. LHS variable created, id is value in the box

```python
>>> c1 = Course("CS 1110", 4)
```

**Heap Space**

```
id3
Point3
id3
x 4
y 2
z 3
```

**Global Space**

```
id3
```

**Classes Have Folders Too**

```
s1
s2
id5
id6
```

**Object Methods**

```
id5
```

```
id6
```

**Example Classes**

```python
>>> import shapes
>>> u = shapes.Point3(4, 2, 3)
>>> u.greet()
"Hi! I am a 3-dimensional point located at (4,2,3)"
```
Defining a Subclass

```python
class Shape:
    """A shape located at x, y""
    def __init__(self, x, y): ..
    def draw(self): ..

class Circle(Shape):
    """An instance is a circle.""
    def __init__(self, x, y, radius): ..
    def draw(self): ..

class Rectangle(Shape):
    """An instance is a rectangle.""
    def __init__(self, x, y, ht, len): ..
    def draw(self): ..
```

Understanding Method Overriding

```python
c1 = Circle(1, 2, 4.0)
print(str(c1))
```

- Which `__str__` do we use?
  - Start at bottom class folder
  - Find first method with name
  - Use that definition
- Each subclass automatically inherits methods of parent.
- New method definitions override those of parent.

Operator Overloading: Equality

Implement `__eq__` to check for equivalence of two `Fractions` instead

```python
class Fraction():
    """Instance attributes:
    numerator: top [int]
    denominator: bottom [int > 0]"
    def __eq__(self, q):
        """Returns: True if self, q equal,
        False if not, or q not a Fraction"
        if type(q) != Fraction:
            return False
        left = self.numerator*q.denominator
        right = self.denominator*q.numerator
        return left == right
```

Name Resolution Revisited

- To look up attribute/method name
  1. Look first in instance (object folder)
  2. Then look in the class (folder)
- Subclasses add two more rules:
  3. Look in the superclass
  4. Repeat 3. until reach object
- Often called the **Bottom-Up Rule**

```python
object
  __init__(self)
  __str__(self)
  __eq__(self)
Shape
  __init__(self, x, y)
  draw(self)
Circle(Shape)
  __init__(self, x, y, radius)
  draw(self)
Circle
  __init__(self, x, y, radius)
  __str__(self)
  __eq__(self)
  draw(self)
```

```
26

29
```
The `isinstance` Function

`isinstance(<obj>,<class>)`
- True if `<obj>`'s class is same as or a subclass of `<class>`
- False otherwise

**Example:**

```python
c1 = Circle(1,2,4.0)
• `isinstance(c1,Circle)` is True
• `isinstance(c1,Shape)` is True
• `isinstance(c1,object)` is True
• `isinstance(c1,str)` is False
```

Generally preferable to `type`
- Works with base types too!

Lists: objects with special "string-like" syntax

- **List**
  - Attributes are indexed
  - Example: `x[2]`
- **Objects**
  - Attributes are named
  - Example: `p.x`

List is **mutable**; strings are not

- **Format:** `<var>[<index>]=<value>`
  - Reassign at index
  - Affects folder contents
  - Variable is unchanged
- Strings cannot do this
  - Strings are immutable

```python
x = [5, 7, 4, -2]
x[1] = 8
s = "Hello!"
s[0] = 'J'  # TypeError: 'str' object does not support item assignment
```

```python
x = [5, 6, 5, 9, 15, 23]
len(x) → 6, length of list
```

Things that Work for All Sequences

### String
- `s = 'abc d'`
- Put characters in quotes
  - Use \ for quote character
- Access characters with []
  - `s[0]` is 'a'
  - `s[5]` causes an error
  - `s[2]` is 'b' (excludes c)
- `len(s) → 5`, length of string

### List
- `x = [5, 6, 5, 9, 15, 23]`
- Put values inside [ ]
  - Separate by commas
- Access values with []
  - `x[0]` is 5
  - `x[6]` causes an error
  - `x[0:2]` is [5, 6] (excludes 2nd 5)
  - `x[3:]` is [9, 15, 23]
  - `len(x) → 6`, length of list

```python
s = 'slithy'
x = [5, 6, 9, 6, 15, 5]
```

Dictionaries are **mutable**

1. Can reassign values
   - `d['ec1'] = 'Ellis'`
2. Can add new keys
   - `d['psb26'] = 'Pearl'`
3. Can delete keys
   - `del d['tm55']`

```python
d = {'ec1': 'Ezra',
     'ec2': 'Ezra',
     'tm55': 'Toni'}
d = {'ec1': 'Ellis',
     'ec2': 'Ezra',
     'tm55': 'Toni'}
dict  

0 5
d  
1 X 8
2 4
3 -2
```

Deleting key deletes both key and value
Nested Lists

- Lists can hold any objects
- Lists are objects
- Therefore lists can hold other lists!

\[
b = [3, 1] \\
c = [1, 4, b] \\
a = [2, 1] \\
x = [1, a, c, 5]
\]

This is drawing accurate, but a little hard to reason about...

Conditionals: “Control Flow” Statements

```python
if b:
    s1 # statement
    s3 # statement

else:
    s2
    s3
```

Flow
Program only takes one path during an execution (something will not be executed!)

For Loops: Processing Sequences

```python
for x in grades:
    print(x)
```

- loop sequence: grades
- loop variable: x
- loop body: print(x)

To execute the for-loop:
1) Check if there is a “next” element of loop sequence
2) If so:
   • assign next sequence element to loop variable
   • Execute all of the body
   • Go back to 1)
3) If not, terminate execution

X = [1, 2, 1, [1, 4, [3, 1]], 5]

Conditionals: If-Elif-Else-Statements (2)

Format
```
if <Boolean expression>:
    <statement> 
elif <Boolean expression>:
    <statement>
else:
    <statement>
```

Notes on Use
- No limit on number of elif
  - Must be between if, else
- else is optional
  - if-elif by itself is fine
- Booleans checked in order
  - Once Python finds a true <Boolean-expression>, skips over all the others
  - else means all <Boolean-expression> are false

For Loop with labels

```python
def num_zeros(the_list):
    '''Returns: the number of zeroes in the_list
    Precondition: the_list is a list'''
    count = 0
    for x in the_list:
        if x == 0:
            count = count + 1
    return count
```

Accumulator variable
- Loop sequence
- Loop variable
- Loop body

Accumulator variable
- Loop sequence
- Loop variable
- Loop body
Modifying the Contents of a List

```python
def add_bonus(grades):
    """Adds 1 to every element in a list of grades
    (either floats or ints)""
    size = len(grades)
    for k in range(size):
        grades[k] = grades[k]+1

lab_scores = [8,9,10,5,9,10]
print("Initial grades are: "+str(lab_scores))
add_bonus(lab_scores)
print("With bonus, grades are: "+str(lab_scores))
```

Beyond Sequences: The while-loop

```python
while <condition>:
    statement 1 ...
    statement n
```

Relationship to for-loop
- Broader notion of “keep working until done”
- Must explicitly ensure condition becomes false
- You explicitly manage what changes per iteration

Recursion

**Recursive Function:**
A function that calls itself

**Two parts to every recursive function:**
1. A simple case: can be solved easily
2. A complex case: can be made simpler (and simpler, and simpler... until it looks like the simple case)

Three Steps for Divide and Conquer

1. Decide what to do on “small” data
   - Some data cannot be broken up
   - Have to compute this answer directly
2. Decide how to break up your data
   - Both “halves” should be smaller than whole
   - Often no wrong way to do this (next lecture)
3. Decide how to combine your answers
   - Assume the smaller answers are correct
   - Combine them to give the aggregate answer

Recursion is great for Divide and Conquer

**Goal:** Solve problem P on a piece of data

**Idea:** Split data into two parts and solve problem

```python
def factorial(n):
    """Returns: factorial of n.
    Precondition: n ≥ 0 an int""
    if n == 0:
        return 1
    return n*factorial(n-1)
```

Recursive Call Frames (all calls complete!)
Search Algorithms

Recall from last lecture:

- Searching for data is a common task
  - Linear search: on the order of n
    - input doubles? $\Rightarrow$ work doubles!
  - Binary search: on the order of $\log_2 n$
    - input doubles? $\Rightarrow$ work increases by just 1 unit!
    - BUT data needs to be sorted...

- Sorting data now suddenly interesting...

Sorting Algorithms

- Sorting data is a common task
  - Insertion sort: on the order of $n^2$
    - input doubles? $\Rightarrow$ work quadruples! (yikes)
  - Merge sort: on the order of $n \cdot \log_2(n)$
    - input doubles? $\Rightarrow$ work increases by a bit more than double

For fun, check out the visualizations:
https://www.youtube.com/watch?v=xxcpvCGrCBc
https://www.youtube.com/watch?v=ZRPoEKX7jg

Good Luck

Have an awesome Summer!

STAY CALM

Read the instructions