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
- \((3 \times 7 + 2) \times 0.1\)
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

Type **str:**
- Values: strings
- Double quotes: "abc"
- Single quotes: 'abc'
- Ops: + (concatenation)
An assignment statement:
• takes an expression
• evaluates it, and
• stores the value in a variable

Example:

\[ x = 4 + 1 \]
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**)

The type belongs to the **value**, not to the **variable**.
### Expressions vs. Statements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>Represents</strong> something</td>
<td>• <strong>Does</strong> something</td>
</tr>
<tr>
<td>▪ Python <em>evaluates it</em></td>
<td>▪ Python <em>executes it</em></td>
</tr>
<tr>
<td>▪ End result is a value</td>
<td>▪ Need not result in a value</td>
</tr>
<tr>
<td>• Examples:</td>
<td>• Examples:</td>
</tr>
<tr>
<td>▪ $2.3$</td>
<td>▪ $x = 2 + 1$</td>
</tr>
<tr>
<td>▪ $(3+5)/4$</td>
<td>▪ $x = 5$</td>
</tr>
<tr>
<td>▪ $x == 5$</td>
<td></td>
</tr>
</tbody>
</table>

*Look so similar but they are not!*
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:
  
  \texttt{best\_function\_ever}(x, y, \ldots)

- Arguments
  - Separated by commas
  - Can be any expression

A function might have 0, 1, \ldots 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

What Python can access directly

int()
float()
str()
type()
print()
...

x \ 3.14159
math

sqrt()
log()
e
pi
3.14159
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)

  - function name
  - parameters
  - local variables
  - instruction counter

**Not just a pretty picture!**
The information in this picture depicts *exactly* what is stored in memory on your computer.
Function Access to Global Space

```python
# height3.py
INCHES_PER_FT = 12

def get_feet(ht_in_inches):
    feet = ht_in_inches // INCHES_PER_FT
    return feet

answer = get_feet(68)
print(answer)
```

Python has just executed line 6.

```
C:\> python height3.py
5
```
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

Representative Tests for `vowel_count(w)`

- Word with just one vowel
  - For each possible vowel!
- Word with multiple vowels
  - Of the same vowel
  - Of different vowels
- Word with only vowels
- Word with no vowels
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

\[
\text{variable}.\text{method}(\ arguments\ )
\]

Example:

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

Global Space

Heap Space

```
Point3

id3
```

```
x  4
y  2
z  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 <em>values</em></td>
<td>• Refer to instances as <em>objects</em></td>
</tr>
<tr>
<td>• Instantiate with simple assignment statement</td>
<td>• Instantiate with assignment statement with a <em>constructor</em></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)
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

```
c1 = Course("CS 1110", 4)
```
Classes Have Folders Too

Object Folders
- Separate for each *instance*
- Example: 2 Student *objects*

```
s1  id5
s2  id6
```

```
id5  Student
  netID  'abc123'
  courses  id2
  major  "Music"
  n_credit  15

id6  Student
  netID  'def456'
  courses  id3
  major  "History"
  n_credit  14
```

Class Folders
- Data common to *all* instances
- Not just data!
- *Everything* common to all instances goes here!
Object Methods

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

```
class Student:
    def __init__(self, netID, courses, major):
        self.max_credit = 20
        self.netID = netID
        self.courses = courses
        self.major = major

id5 = Student('abc123', ['id2'], 'Music')
```
Defining a Subclass

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): ...

Superclass
  Parent class
  Base class
Subclass
  Child class
  Derived class

Shape
  Rectangle
  Circle

__init__(self,x,y)
draw(self)

Circle(Shape)
  __init__(self,x,y, radius)
draw(self)

Rectangle(Shape)
  __init__(self,x,y, ht, len)
draw(self)
```python
class Shape:
    """A shape @ location x,y """
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Circle(Shape):
    """Instance is Circle @ x,y w/size radius"""
    def __init__(self, x, y, radius):
        super().__init__(x,y)
        self.radius = radius
```

- **Want to use the original version of the method?**
  - New method = original + more
  - Don't repeat code from the original

- **Call old method explicitly**
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.
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

c1 = Circle(1,2,4.0)
r = c1.radius
c1.draw()
Operator Overloading: Equality

Implement __eq__ to check for equivalence of two Fractions instead

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
eq vs. is

== compares equality
is compares identity

c1 = Circle(1, 1, 25)
c2 = Circle(1, 1, 25)
c3 = c2

c1 == c2 → ? True

c1 is c2 → ? False

c2 == c3 → ? True

c2 is c3 → ? True
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`
Sequences: Lists of Values

**String**
- $s = 'abc\ d'$
  - Put characters in quotes
    - Use \ for quote character
  - Access characters with []
    - $s[0]$ is 'a'
    - $s[5]$ causes an error
    - $s[0:2]$ is 'ab' (excludes c)
    - $s[2:]$ is 'c\ d'
  - $\text{len}(s) \rightarrow 5$, 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 2\text{nd} 5)
    - $x[3:]$ is [9, 15, 23]
  - $\text{len}(x) \rightarrow 6$, length of list

**Sequence** is a name we give to both
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*

\[
x = [5, 7, 4, -2]\\
x[1] = 8\\
s = "Hello!"\\
s[0] = 'J'
\]

```
TypeError: 'str' object does not support item assignment
```
Things that Work for All Sequences

\[
s = \text{‘slithy’} \quad \quad x = [5, 6, 9, 6, 15, 5]
\]

- \(s\).index(‘s’) → 0
- \(s\).count(‘t’) → 1
- len(s) → 6
- \(s[4] \rightarrow \text{“h”}
- \(s[1:3] \rightarrow \text{“li”}
- \(s[3:] \rightarrow \text{“thy”}
- \(s[-2] \rightarrow \text{“h”}
- \(s + \text{‘toves’} \rightarrow \text{“slithy toves”}
- \(s * 2 \rightarrow \text{“slithyslithy”}
- ‘t’ in s → True
- s.index(‘s’) → 0
- x.index(5) → 0
- x.count(6) → 2
- len(x) → 6
- \(x[4] \rightarrow 15
- \(x[1:3] \rightarrow [6, 9]
- \(x[3:] \rightarrow [6, 15, 5]
- \(x[-2] \rightarrow 15
- \(x + [1, 2] \rightarrow [5, 6, 9, 6, 15, 5, 1, 2]
- \(x * 2 \rightarrow [5, 6, 9, 6, 15, 5, 5, 6, 9, 6, 15, 5]
- 15 in x → True
- ‘t’ in s → True
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']

Deleting key deletes both key and value

```
d = {'ec1':'Ezra', 'ec2':'Ezra', 'tm55':'Toni'}
```
Nested Lists

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

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

Global Space

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>id1</td>
</tr>
<tr>
<td>c</td>
<td>id2</td>
</tr>
<tr>
<td>a</td>
<td>id3</td>
</tr>
<tr>
<td>x</td>
<td>id4</td>
</tr>
</tbody>
</table>

Heap

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>id2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>id1</td>
</tr>
<tr>
<td>id3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>id4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>id3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>id2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

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

Conceptually, you can visualize nested lists like this:

\[
x = [1, [2, 1], [1, 4, [3, 1]], 5]
\]

\[
x = [1, [2, 1], [1, 4, [3, 1]], 5]
\]
Conditionals: “Control Flow” Statements

if b:
    s1  # statement
    s3  # statement

if b:
    s1
else:
    s2
    s3

Branch Point: Evaluate & Choose

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

```
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
def num_zeroes(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
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))
```

If you need to modify the list, you need to use range to get the indices.

Watch this in the python tutor!
Beyond Sequences: The while-loop

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)
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

- **data 1**
- **data 2**

Solve Problem P

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

factorial(3)
Search Algorithms

Recall from last lecture:

• Searching for data is a common task
  ▪ **Linear search**: on the order of n
    • input doubles? → work **doubles**!
  ▪ **Binary search**: on the order of log2 n
    • input doubles? → 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=xxcpvCGrCBo
https://www.youtube.com/watch?v=ZRPoEKHXTJg
Have an awesome Summer!