CS 1110:
Introduction to Computing Using Python

Lecture 20

**isinstance** and **While Loops**

[Andersen, Gries, Lee, Marschner, Van Loan, White]
Announcements

• A4: Due 4/20 at 11:59pm
  ▪ Should only use our `str` method to test `__init__`
  ▪ Testing of all other methods should be done as usual
• Thursday 4/20: Review session in lecture
• Prelim 2 on Tuesday 4/25, 7:30pm – 9pm
  ▪ Covers material up through Tuesday 4/18
  ▪ Lecture: Professor office hours
  ▪ Labs: TA/consultant office hours
• No labs on 4/26
More Mixed Number Example

• What if we want to add mixed numbers and fractions?
The `isinstance` Function

- `isinstance(<obj>,<class>)`
  - True if `<obj>`’s class is same as or a subclass of `<class>`
  - False otherwise
- **Example:**
  - `isinstance(e,Executive)` is True
  - `isinstance(e,Employee)` is True
  - `isinstance(e,object)` is True
  - `isinstance(e,str)` is False
- Generally preferable to `type`
  - Works with base types too!
>>> e = Employee('Bob', 2011)
>>> isinstance(e, Executive)

FALSE
More Mixed Number Example

- What if we want to add mixed numbers and fractions?
Review: For Loops

The for-loop:

```
for x in seq:
    print x
```

- **loop sequence**: `seq`
- **loop variable**: `x`
- **body**: `print x`

To execute the for-loop:
1. Check if there is a “next” element of **loop sequence**
2. If not, terminate execution
3. Otherwise, assign element to the **loop variable**
4. Execute all of the **body**
5. Repeat as long as 1 is true
Beyond Sequences: The while-loop

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

- Relationship to for-loop
  - Must explicitly ensure condition becomes false
  - *You* explicitly manage what changes per iteration
While-Loops and Flow

print 'Before while'
count = 0
i = 0

while i < 3:
    print 'Start loop ' + str(i)
count = count + i
    i = i + 1
    print 'End loop '

print 'After while'

Output:
Before while
Start loop 0
End loop
Start loop 1
End loop
Start loop 2
End loop
After while
What gets printed?

```python
a = 0
while a < 1:
    a = a + 1
print a
```

prints 1

print a
What gets printed?

a = 0
while a < 2:
    a = a + 1
    print a

print a
a = 0

while a > 2:
    a = a + 1

print a
What gets printed?

\[
a = 0
\]

\[
\textbf{while} \ a < 3:
\]
\[
\quad \textbf{if} \ a < 2:
\]
\[
\quad \quad a = a + 1
\]

\[
\text{print} \ a
\]
What gets printed?

\[
a = 4
\]

\[
\text{while } a > 0:
\]

\[
a = a - 1
\]

\[
\text{print } a
\]

prints 0
What gets printed?

```
a = 8
b = 12
while a != b:
    if a > b:
        a = a - b
    else:
        b = b - a
print a
```

A: INFINITE LOOP
B: 8
C: 12
D: 4   CORRECT
E: I don’t know

This is Euclid’s Algorithm for finding the greatest common factor of two positive integers.

Trivia: It is one of the oldest recorded algorithms (~300 B.C.)
More Mixed Number Example

• Adding with greatest common factor, finally!
• Reducing
Note on Ranges

- **m..n** is a range containing **n+1-m** values
  - 2..5 contains 2, 3, 4, 5. Contains 5+1 – 2 = 4 values
  - 2..4 contains 2, 3, 4. Contains 4+1 – 2 = 3 values
  - 2..3 contains 2, 3. Contains 3+1 – 2 = 2 values
  - 2..2 contains 2. Contains 2+1 – 2 = 1 values

- Notation **m..n** always implies that **m <= n+1**
  - If **m = n+1**, the range has 0 values
while Versus for

# process range b..c-1
for k in range(b,c)
    # code involving k
while k < c:
    # code involving k
    k = k+1

Must remember to increment

# process range b..c
for k in range(b,c+1)
    # code involving k
while k <= c:
    # code involving k
    k = k+1
while Versus for

# incr seq elements
for k in range(len(seq)):
    seq[k] = seq[k]+1

# incr seq elements
k = 0
while k < len(seq):
    seq[k] = seq[k]+1
    k = k+1

while is more flexible, but often requires more code
Patterns for Processing Integers

range a..b-1

i = a
while i < b:
    # process integer i
    i = i + 1

# store in count # of '/s in string s
count = 0
i = 0
while i < len(s):
    if s[i] == '/':
        count = count + 1
    i = i + 1

# count is # of '/s in s[0..s.length()-1]

range c..d

i = c
while i <= d:
    # process integer i
    i = i + 1

# Store in v the sum 1/1 + 1/2 + ...+ 1/n
v = 0
i = 0
while i <= n:
    v = v + 1.0 / i
    i = i + 1

# v = 1/1 + 1/2 + ...+ 1/n
# list of squares to N

def list_squares(N):
    seq = []
    n = n = floor(sqrt(N)) + 1
    for k in range(n):
        seq.append(k*k)
    return seq

# list of squares to N

def list_squares_while(N):
    seq = []
    k = 0
    while k*k <= N:
        seq.append(k*k)
        k = k+1
    return seq

A for-loop requires that you know where to stop the loop **ahead of time**

A while loop can use complex expressions to check if the loop is done

4/13/17 While Loops
while Versus for

# List of n Fibonacci numbers
fib = [1, 1]
for k in range(2, n):
    fib.append(fib[-1] + fib[-2])

# List of n Fibonacci numbers
fib = [1, 1]
while len(fib) < n:
    fib.append(fib[-1] + fib[-2])

Sometimes you do not use the loop variable at all

Do not need to have a loop variable if you don’t need one
Cases to Use while

Great for when you must modify the loop variable

# Remove all 3's from list t

```python
i = 0
while i < len(t):
    # no 3's in t[0..i–1]
    if t[i] == 3:
        del t[i]
    else:
        i += 1
```

# Remove all 3's from list t

```python
while 3 in t:
    t.remove(3)
```
Cases to Use while

Great for when you must modify the loop variable
But first, +=

- Can shorten \( i = i + 1 \) as:
  - \( i += 1 \)
- Also works for -=, *=, /=, %=.
Cases to Use **while**

Great for when you must **modify** the loop variable

```python
# Remove all 3's from list t
i = 0
while i < len(t):
    # no 3's in t[0..i-1]
    if t[i] == 3:
        del t[i]
    else:
        i += 1
```

```python
# Remove all 3's from list t
while 3 in t:
    t.remove(3)
```

Stopping point keeps changing.

The stopping condition is not a numerical counter this time. Simplifies code a lot.
Collatz Conjecture

• Does this loop terminate for all \( x \)?

```python
while x != 1:
    if x % 2 == 0:  # if x is even
        x /= 2
    else:  # if x is odd
        x = 3 * x + 1
```

WHILE LOOPS CAN BE HARD. Must think formally.
Some Important Terminology

• **assertion**: true-false statement placed in a program to `assert` that it is true at that point
  - Can either be a `comment`, or an `assert` command

• **invariant**: assertion supposed to "always" be true
  - If temporarily invalidated, must make it true again
  - **Example**: class invariants and class methods

• **loop invariant**: assertion supposed to be true before and after each iteration of the loop

• **iteration of a loop**: one execution of its body
Preconditions & Postconditions

- **Precondition:** assertion placed before a segment
- **Postcondition:** assertion placed after a segment

Relationship Between Two

If precondition is true, then postcondition will be true