CS 134 Lecture 11:

While Loops

Announcements & Logistics

- **HW 5** will be released today on GLOW
- **Lab 4** Part 1 due Wed/Thurs 10 pm
  - We will return feedback (including tests not found in `runtests.py`)
- Reminder that Midterm is **Thursday March 14**
  - Two exam slots: 6-7.30 pm, 8-9.30 pm
  - Room: Bronfman auditorium
- Midterm review Monday March 11 evening 7-9 pm in Bronfman Auditorium
- How to study: review lectures
  - Practice past HW and labs (using pencil and paper)
  - Additional POGIL worksheets posted on course website (resources)

Do You Have Any Questions?

Last Time

- Wrapped up examples of nested for loops and nested lists
- Discussed the difference between importing functions vs running python code as a script
    - Role of special variable `__name__`
- Introduced list comprehensions
    - Short-hand expressions for common looping patterns
    - "Pythonic feature": anything we can do with list comprehensions, we can do with standard looping patterns

Today's Plan

- New iteration statement: the **`while`** loop
- Discuss the mutability of different data types and the implications

When you don't know when to stop (ahead of time):

While Loop

Story so far: for loops

- **for loops** in Python are meant to iterate directly over a **fixed sequence**
    - No need to know the sequence's length ahead of time
- Interpretation of for loops in Python:

```
for each item in given sequence:
    (do something with item)
```

- Other programming languages (like Java) have for loops that require you to explicitly specify the length of the sequence or a stopping condition
- Thus Python for loops are sometimes called "**for each**" loops
- **Takeaway**: For loops in Python are meant to iterate directly over each item of a given ***iterable*** object (such as a sequence)

What If We Don't Know When to Stop?

- Stopping condition of for loop: **no more elements in sequence**

```
["A", "chilly", "autumn", "day"]
```

```
item
```

- What if we don't know when to stop?
    - Suppose you had to write a program to ask a user to enter a name, repeatedly, until the user enters "quit", in which case you stop asking for input and print "Goodbye"

- **How many times** should the loop execute?
- **Under what condition** should the loop end?

`while` loop

- **while loops** keep iterating until a continuation condition holds
- Syntax:

Repeat loop body as long as this evaluates to `True`

```
while boolean_expression:
    <loop body>
    <loop body>
```

Indentation defines the **loop body**

```
while True:
    print("never leaves")
```

"Infinite" loop!

```
while False:
    print("never enters")
```

Loop body never executes

While Loop Example

- Example of a while loop that depends on user input

```
prompt = "Please enter a name (type quit to exit): "
name = input(prompt)

while (name != "quit"):
    print("Hi,", name)
    name = input(prompt)
print("Goodbye")
```

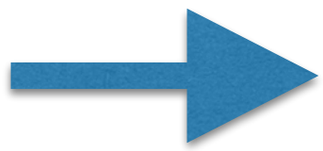
Stopping condition

While Loop to Print Halves

- Given a number, print all the positive "halves" : keep dividing n by two and printing the quotient until it becomes smaller than 0

```
def print_halves(n):
    while n > 0:
        print(n)
        n = n//2

print_halves(100)
```



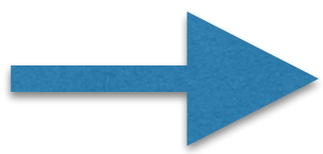
```
100
50
25
12
6
3
1
```

While Loop to Print Halves

- Given a number, print all the positive "halves" : keep dividing n by two and printing the quotient until it becomes smaller than 0

```
def print_halves(n):
    while n > 0:
        print(n)
        n = n//2

print_halves(100)
```



```
100
50
25
12
6
3
1
```

```
def print_halves(n):
    while n > 0:
        print(n)
        n = n//2

print_halves(100)
```

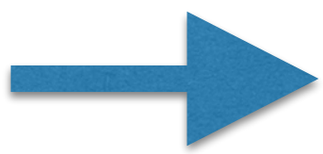
What about this loop?

While Loop to Print Halves

- Given a number, print all the positive "halves" : keep dividing n by two and printing the quotient until it becomes smaller than 0

```
def print_halves(n):
    while n > 0:
        print(n)
        n = n//2

print_halves(100)
```



```
100
50
25
12
6
3
1
```

```
def print_halves(n):
    while n > 0:
        print(n)
    n = n//2

print_halves(100)
```

Infinite loop! Indentation matters!

`while` and `if` side by side

```
if boolean_expresion:
    # statement 1
    # statement 2
    ....
    ....


# end of if
```

Execute this **once** if the **boolean expression** evaluates to true

```
while boolean_expresion:
    # statement 1
    # statement 2
    ....
    ....


# end of while
```

Keep executing this **while** the **boolean expression** (*continues*) to evaluate to true

Side by Side: for and while loops

```
for i in range(5):
    print('$' * i)
```

All these steps are implicit in a **Python** for loop: `i` takes on values 0, 1, 2, 3, 4

Explicitly **initialize** variable

Test stopping condition

```
i = 0
while i < 5:
    print('$' * i)
    i += 1
```

Update value of variable used in test condition

Common while loops steps:

- **Initialize** a variable used in the test condition
- **Test** condition that causes the loop to end when `False`
- Within the loop body, **update** the variable used in the test condition

Side by Side: for and while loops

```
vowels = 'aeiou'

for char in vowels:
    print(char)
```

No need to find `len` or to index using `[]`

Iterate directly over elements of sequence

```
i = 0
while i < len(vowels):
    print(vowels[i])
    i += 1
```

Explicitly **initialize** variable

Test stopping condition

Update value of variable used in test condition

Common while loops steps:

- **Initialize** a variable used in the test condition
- **Test** condition that causes the loop to end when `False`
- Within the loop body, **update** the variable used in the test condition

Breaking out of loops

- Stopping condition of for loop: **no more elements in sequence**
- What if we want to stop (break out) early: how did we handle this?
- Let's recap one such example: `index_of(elem, l)`
    - Write a function `index_of(elem, l)` that takes two arguments (`elem` of any type and list `l`) and returns the first index of `elem` if `elem` is in the list `l` otherwise returns `-1`

```
>>> index_of('blue', ['red', 'blue', 'blue'])
1
>>> index_of(14, [23, 1, 10, 11, 14])
4
>>> index_of('a', ['b', 'c', 'd', 'e'])
-1
```

Side by Side: `index_of`

```
def index_of(elem, l):

    for i in range(len(l)):
        # match?
        if l[i] == elem:
            # stop loop!
            return i

# if not found
return -1
```

```
def index_of(elem, l):

    found = False # flag
    index_of_elem = -1
    i = 0

    while not found and i < len(l):
        # match?
        if elem == l[i]:
            # stop the loop!
            found = True
            index_of_elem = i
        # keep going
        i += 1

    return index_of_elem
```

Takeaways

- New iteration statement: `while` loop as an alternative to `for` loops are meant to iterate for a fixed number of times
    - Used when the stopping condition is determined ***"on the fly"***
    - Keeps iterating as long as Boolean condition evaluates to `True`