Lecture 22

While Loops

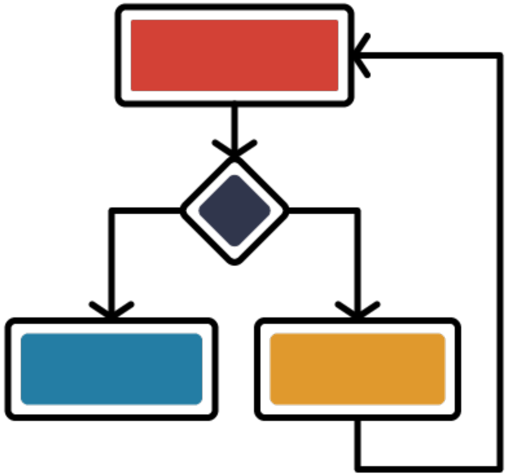
Announcements for Today

Assignments

- A4 is now graded
  - **Mean**: 88.9 **Median**: 92
  - **Mean**: 8.9 hrs **SDev**: 4.5 hrs
- A5 graded by Saturday
- Keep working on A6
  - Follow micro-deadlines
  - Should be on **Task 3** now
  - Finish Part C **tomorrow**
- A7 posted on **Friday**

Video Lessons

- **Lesson 24** for today
- **Lesson 25** for Thurs
- **Lessons 26, 27** for Tues



Lunch with the Professor!

- A tradition we started last year
  - Nine students get (paid) lunch to talk to prof
  - But did not get reauthorized till last Fri!
- Hoping to make up some lost ground
  - **Monday 12-1** in Mattins
  - **Friday 1:15-2:15** in Trillium
- Sign-up form in CMS up today!
  - Will have a separate form for each day
  - But you can sign up for both

Recall: The For-Loop

```
# Create local var x
x = seqn[0]
print(x)
x = seqn[1]
print(x)
...
x = seqn[len(seqn)-1]
print(x)
```

Not valid Python

```
# Write as a for-loop
for x in seqn:
    print(x)
```

Key Concepts

- **iterable**: seqn
- **loop variable**: x
- **body**: print(x)

Important Concept in CS: Doing Things Repeatedly

1. Process each item in a sequence
   - Compute aggregate statistics fo[r a dataset,] such as the mean, median, stan[dard deviation...]
   - Send everyone in a Facebook group an appointment time

```
for x in sequence:
    process x
```

2. Perform *n* trials or get *n* samples.
   - A4: draw a triangle six times to m[ake a hexagon]
   - Run a protein-folding simulati[on...]

```
for x in range(n):
    do next thing
```

3. Do something an unknown number of times
   - CUAUV team, vehicle keeps moving until reached its goal

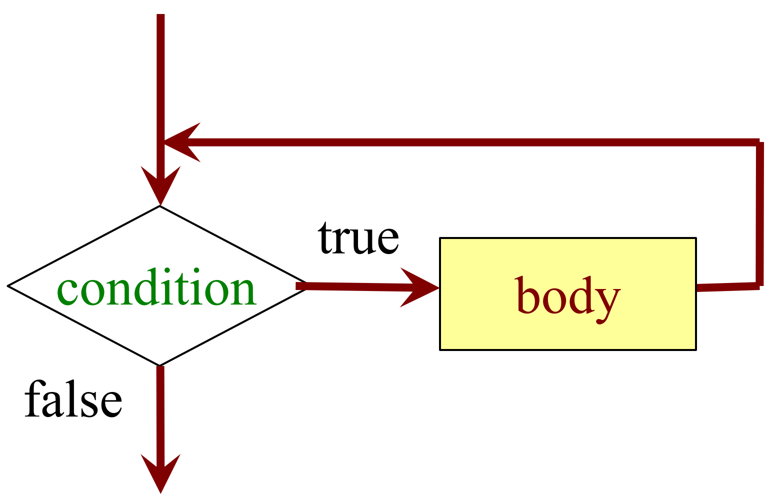
????

Beyond Sequences: The **while-loop**

```
while <condition>:
    statement 1
    …
    statement n
```

loop condition

loop body



Vs For-Loop

- Broader notion of loop
  - You define "more to do"
  - Not limited sequences
- Must manage loop var
  - You create it before loop
  - You update it inside loop
  - For-loop automated it
- Trickier to get right

while Versus for

For-Loop

```
def sum_squares(n):
    """Rets: sum of squares
    Prec: n is int > 0"""
    total = 0

    for x in range(n):
        total = total + x*x
```

While-Loop

```
def sum_squares(n):
    """Rets: sum of squares
    Prec: n is int > 0"""
    total = 0
    x = 0
    while x < n:
        total = total + x*x
        x = x+1
```

Must remember to increment

The Problem with While-Loops

- Infinite loops are possible
  - Forget to update a loop variable
  - Incorrectly write the boolean expression
- Will hang your program
  - Must type control-C to abort/quit
- But detecting problems is not easy
  - Sometimes your code is just slow
  - Scientific computations can take hours
- **Solution:** Traces

Tracing While-Loops

```
print('Before while')
total = 0
x = 0
while x < n:
    print('Start loop '+str(x))
    total = total + x*x
    x = x + 1
    print('End loop ')
print('After while')
```

Important

Important

Output:

```
Before while
Start loop 0
End loop
Start loop 1
End loop
Start loop 2
End loop
After while
```

How to Design While-Loops

- Many of the same rules from for-loops
  - Often have an **accumulator variable**
  - Loop body adds to this accumulator
- Differences are loop variable and iterable
  - Typically **do not have iterable**
- Breaks up into three **design patterns**
  1. Replacement to range()
  2. Explicit goal condition
  3. Boolean tracking variable

While Loops and Lists

For-Loop

```
def increment_for(seq):
    """Increments each
    element of seq list
    Prec: seq contains ints"""
    for k in range(len(seq)):
        seq[k] = seq[k]+1
```

While-Loop

```
def increment_while(seq):
    """Increments each
    element of seq list
    Prec: seq contains ints"“
    k = 0
    while k < len(seq):
        seq[k] = seq[k]+1
        k = k + 1
```

Must still remember to increment

Using the Goal as a Condition

```
def prompt(prompt,valid):
    """Returns: the choice from a given prompt.

    This function asks the user a question, and waits for a response.  It
    checks if the response is valid against a list of acceptable answers.
    If it is not valid, it asks the question again. Otherwise, it returns
    the player's answer.

    Precondition: prompt is a string
    Precondition: valid is a tuple of strings"""
    pass # Stub to be implemented
```

Tells you the stop condition

Using the Goal as a Condition

```
def prompt(prompt,valid):
    """Returns: the choice from a given prompt.

    Preconditions: prompt is a string, valid is a tuple of strings"""
    response = input(prompt)

    # Continue to ask while the response is not valid.
    while not (response in valid):
        print('Invalid response.  Answer must be one of ')+str(valid)
        response = input(prompt)

    return response
```

Using a Boolean Variable

```
def roll_past(goal):
    """Returns: The score from rolling a die until passing goal.

    This function starts with a score of 0, and rolls a die, adding the
    result to the score.  Once the score passes goal, it stops and
    returns the result as the final score.
    If the function ever rolls a 1, it stops and the score is 0.

    Preconditions: goal is an int > 0"""
    pass # Stub to be implemented
```

Condition is too complicated

Introduce a boolean variable.
Use it to track condition.

Using a Boolean Variable

```
def roll_past(goal):
    """Returns: The score from rolling a die until passing goal."""
    loop = True  # Keep looping until this is false
    score = 0
    while loop:
        roll = random.randint(1,6)
        if roll == 1:
            score = 0; loop = False
        else:
            score = score + roll; loop = score < goal
    return score
```

Track the condition

Advantages of **while** vs **for**

```
# table of squares to N
seq = []
n = floor(sqrt(N)) + 1
for k in range(n):
    seq.append(k*k)
```

```
# table of squares to N
seq = []
k = 0
while k*k < N:
    seq.append(k*k)
    k = k+1
```

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

Advantages of while vs for

Fibonacci numbers:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

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

Sometimes you do not use the loop variable at all

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

Do not need to have a loop variable if you don't need one

Difficulties with while

Be careful when you **modify** the loop variable

```
def rem3(lst):
    """Remove all 3's from lst"""
    i = 0
    while i < len(lst):
        # no 3's in lst[0..i–1]
        if lst[i] == 3:
            del lst[i]
        i = i+1
```

```
>>> a = [3, 3, 2]
>>> rem3(a)
>>> a
```

A: [2]
B: [3]
C: [3,2]
D: []
E: something else

Difficulties with while

Be careful when you modify the loop variable

```
def rem3(lst):
    """Remove all 3's from lst"""
    i = 0
    while i < len(lst):
        # no 3’s in lst[0..i–1]
        if lst[i] == 3:
            del lst[i]
        i = i+1
```

```
>>> a = [3, 3, 2]
>>> rem3(a)
>>> a
```

A: [2]
B: [3]
C: [3,2] **Correct**
D: []
E: something else

Difficulties with while

Be careful when you **modify** the loop variable

```
def rem3(lst):
    """Remove all 3's from lst"""
    i = 0
    while i < len(lst):
        # no 3's in lst[0..i–1]
        if lst[i] == 3:
            del lst[i]
        else:
            i = i+1
```

Stopping point keeps changing

```
def rem3(lst):
    """Remove all 3's from lst"""
    while 3 in lst:
        lst.remove(3)
```

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

Application: Convergence

- How to implement this function?

```
def sqrt(c):
    """Returns the square root of c"""
```

- Consider the polynomial $f(x) = x^2 – c$
  - Value sqrt(c) is a *root* of this polynomial
- Suggests a use for **Newton's Method**
  - **Start with a guess** at the answer
  - Use calculus formula to improve guess

Example: Sqrt(2)

- Actual answer: 1.414235624
- $x_{n+1} = x_n/2 + c/2x_n$
- $x_0 = 1$ # Rough guess of sqrt(2)
- $x_1 = 0.5 + 1 = 1.5$
- $x_2 = 0.75 + 2/3 = 1.41666$
- $x_3 = 0.7083 + 2/2.833 = 1.41425$

When Do We Stop?

- We don't know the sqrt(c)
  - This was thing we wanted to compute!
  - So we cannot tell how far off we are
  - But we do know $\text{sqrt}(c)^2 = c$
- So square approximation and compare
  - while x*x is not close enough to c
  - while abs(x*x – c) > threshold

When Do We Stop?

- We don't know the sqrt(c)
  - This was thing we wanted to compute!
  - So we cannot tell how far off we are
  - But we do know $\text{sqrt}(c)^2 = c$
- So square approximation and compare

While-loop computes until
the answer **converges**

The Final Result

```
def sqrt(c,err=1e-6):
    """Returns: sqrt of c with given margin of error.

    Preconditions: c and err are numbers > 0"""
    x = c/2.0

    while abs(x*x-c) > err:
        # Get x_{n+1} from x_n
        x = x/2.0+c/(2.0*x)

    return x
```

Using `while`-loops Instead of `for`-loops

Advantages

- Better for **modifying data**
  - More natural than range
  - Works better with deletion
- Better for **convergent tasks**
  - Loop until calculation done
  - Exact steps are unknown
- Easier to **stop early**
  - Just set loop var to False

Disadvantages

- Performance is **slower**
  - Python optimizes `for`-loops
  - Cannot optimize `while`
- **Infinite loops** more likely
  - Easy to forget loop vars
  - Or get stop condition wrong
- **Debugging** is harder
  - Will see why in later lectures

Optional Exercise

The Game of Pig: A Random Game

- Play progresses clockwise
- On your turn, throw the die:
  - If roll 1: lose turn, score zero
  - Anything else: add it to score
    - Can also roll again (and lose)
    - If stop, score is "banked"
- First person to 100 wins

The Game of Pig: A Random Game

- Play progresses clockwise
- On your turn, throw the die:
  - If roll 1: lose t
  - Anyt ore
    - Can also roll again (and lose)
    - If stop, score is "banked"
- First person to 100 wins

Easy to write without classes

Designing an AI for Opponent is Easy

# Throws	Survial	Expected Gain	Expected Value
1	83%	3.33	3.33
2	69%	2.78	6.11
3	58%	2.32	8.43
4	48%	1.92	10.35
5	40%	1.61	11.96
6	33%	1.34	13.30
7	28%	1.12	14.42
8	23%	.93	15.35
9	19%	.77	16.12
10	16%	.65	16.77
…	…	…	…
50	0.01%	0.0004	19.998

Designing an AI for Opponent is Easy

# Throws	Survial	Expected Gain	Expected Value
1	83%	3.33	3.33
2	69%	2.78	6.11
3	58%	2.32	8.43
4	48%	1.92	10.35
5	40%	1.61	11.96
6	33%	1.34	13.30
7	28%	1.12	14.42
8	23%	.93	15.35
9	19%	[illegible]	16.12
10	16%	[illegible]	16.77
…	…	…	…
50	0.01%	0.0004	19.998

Strategy:
Bank at 20

The Primary Function

```
def play(target):
    """Plays a single game of Pig to target score.

    Precondition: target is an int > 0"""
    # Initialize the scores
    # while no one has reached the target
        # Play a round for the player
        # If the player did not reach the target
            # Play a round for the opponent
    # Display the results
```

The Player Round

```
def player_turn():
    """ Runs a single turn for the player."""
    # while the player has not stopped
        # Roll the die
        # If is a 1
            # Set score to 0 and stop the turn
        # else
            # Add the to the score
            # Ask the player whether to continue
    # Return the score
```

Prompt helper

The Opponent Round

```
def roll_past(goal):
    """Returns: The score from rolling a die until passing goal."""
    loop = True  # Keep looping until this is false
    score = 0
    while loop:
        roll = random.randint(1,6)
        if roll == 1:
            score = 0; loop = False
        else:
            score = score + roll; loop = score < goal
    return score
```

Look familiar?