while BOOL_CONDITION:
  LOOP_BODY
# modify variables, affect expression

When is a game of chess over?

- If you were to write a program to play chess
  - how many rounds in a game?

https://xkcd.com/710/

Collatz Conjecture (Hailstone)

If number is even
  Divide by 2
If number is odd
  multiply by 3 and add 1
Why Solve This? In Python?

- We want to illustrate an indefinite loop
  - One of many mathematical sequences, but...
- There's an XKCD comic about it!
  - Not everyone enjoys XKCD, but...
- Mathematics is foundational in computer science, but
  - Not everyone enjoys logic/math puzzles, but...

Developing and Reasoning about While Loops

- Don't know: *how many times* loop executes
  - *a priori* knowledge, we'll know afterword
- Do know: condition that should be true after loop
  - Its negation is the expression for `BOOL_CONDITION` (loop guard)

```python
while BOOL_CONDITION:
    LOOP_BODY
    # modify variables, affect expression
```

History: From while to for loops

**while loop (sum list)**

```python
lst = [4,1,8,9]
s = 0
i = 0
while i < len(lst):
    s += lst[i]
i += 1
print(s)
```

**for loop (sum list)**

```python
lst = [4,1,8,9]
s = 0
for n in lst:
    s += n
print(s)
```

Concrete Example: Collatz/Hailstone

- Don't know: *how many times* loop executes
  - some numbers: long sequences, others short
- Do know: condition that should be true after loop
  - It's negation is the expression for loop guard!
  - What is true after loop below finishes?

```python
while value != 1:
    loop body
    # modify value somehow
```
What is this code doing? What gets updated? Is the loop guaranteed to stop?

What is new in this code? What does that new stuff do?

while BOOL_CONDITION:
    LOOP_BODY
    # modify variables, affect expression

```
def hailstone(start, printing=False):
    steps = 0
    current = start
    while current != 1:
        if printing:
            print("{:3d}\t{:6d}".format(steps, current))
        if current % 2 == 0:
            current //= 2
        else:
            current = current * 3 + 1
        steps += 1
        if printing:
            print("{:3d}\t{:6d}".format(steps, current))
    return steps
```
Collatz: New stuff

```python
def hailstone(start, printing=False):
    steps = 0
    current = start
    while current != 1:
        if printing:
            print("{:3d}{:6d}".format(steps, current))
        if current % 2 == 0:
            current //= 2
        else:
            current = current * 3 + 1
        steps += 1
    if printing:
        print("{:3d}{:6d}".format(steps, current))
    return steps
```

Default value, if no argument

Syntax for nicer formatting

Collatz: Guaranteed to stop?

```python
def hailstone(start, printing=False):
    steps = 0
    current = start
    while current != 1:
        if printing:
            print("{:3d}{:6d}".format(steps, current))
        if current % 2 == 0:
            current //= 2
        else:
            current = current * 3 + 1
        steps += 1
    if printing:
        print("{:3d}{:6d}".format(steps, current))
    return steps
```

current influences the stopping condition

Since current is always changed, this should eventually stop

Collatz Data – Average no. of steps

- How do we gather data for numbers <= 10,000?
  - In general for numbers in range(low,high)?
  - Call function, store result, store 10,000 results?

- We'd like counts[k] to be length of sequence for k
  - How do we allocate 10,000 list elements?
  - Like there is "hello" * 3
  - There is [0] * 10000

Think: Analysis in Collatz.py

```python
def analyze(limit):
counts = []
    # max index into count is limit, but start at 1
    for n in range(limit+1):
        counts.append(0)

    for n in range(1, limit+1):
        counts[n] = hailstone(n)

    avg = sum(counts)/len(counts)-1  # ignore index 0
    mx = max(counts)
    dex = counts.index(mx)
    print("average",avg)
    print("max is %d at %d" % (mx,dex))
```
def analyze(limit):
    counts = []
    # max index into count is limit, but start at 1
    for _ in range(limit+1):
        counts.append(0)
    
    for n in range(1, limit+1):
        counts[n] = hailstone(n)

    avg = sum(counts)/len(counts)-1  # ignore index 0
    mx = max(counts)
    dex = counts.index(mx)
    print("average",avg)
    print("max is \%d at \%d" % (mx,dex))

Counts is of size 8+1, we ignore slot 0
• hailstone(1), get 0
• hailstone(2), get 1 step, just divide by 2

Store answer for hailstone(2) in index 2
• hailstone(3), get 7 (10, 5, 16, 8, 4, 2, 1)
• hailstone(4), get 2
counts list when limit is 8?

• hailstone(3), get 7

```
analyze
limit 8
counts 0 0 1 7 0 0 0 0
```

Store answer for hailstone(3) in index 3

• hailstone(4), get 2

```
analyze
limit 8
counts 0 0 1 7 2 0 0 0
```

Store answer for hailstone(4) in index 4

• hailstone(5), get 5 (16, 8, 4, 2, 1)

```
analyze
limit 8
counts 0 0 1 7 2 5 0 0
```

Store answer for hailstone(5) in index 5

• And so on…..

• Hailstone(6) is 8, hailstone(7) is 16, hailstone(8) is 3

```
analyze
limit 8
counts 0 0 1 7 2 5 8 16
```

2/25/2021 Compsci 101, Spring 2021

---

Comp sci 101

Pancakes, While loops, Parallel Lists

Part 3 of 3

Susan Rodger

Nikki Washington

February 25, 2021

```
while BOOL_CONDITION:
    LOOP_BODY
    # modify variables, affect expression
```
High Level View

• We will use parallel lists to track data
  • Each word is stored in a list named **words**
  • Word’s count is stored in a list named **counts**
  • # occurrences of **words[k]** is in **counts[k]**

```
["apple", "fox", "vacuum", "lime"]
[5,2,25,15]
```

• What happens when we read a word?

  Read word “apple”?

```
["apple", "fox", "vacuum", "lime", "banana"]
[6,2,25,15]
```

• What happens when we read a word?

  Read word “banana”?

Add into words
High Level View

• We will use parallel lists to track data
  • Each word is stored in a list named `words`
  • Word’s count is stored in a list named `counts`
  • # occurrences of `words[k]` is in `counts[k]`

```python
["apple", "fox", "vacuum", "lime", "banana"]
[6, 2, 25, 15, 1]
```

• What happens when we read a word?

Read word “banana”?

Pseudo-code for `getFileData`

• Let user choose a file to open
  • Read each line of the file
    • Process each word on the line
      • If word never seen before? Add to words and counts
      • Update # occurrences using `.index` and location

• Think: What would we do for each color when doing step 5 (translate to code) of the 7 steps?

Step 3 of 7 steps: Generalize
Pseudo-code for getFileData

- Let user choose a file to open
  - SOME KIND OF CODE CHOOSES A FILE
- Read each line of the file
  - FOR LOOP
    - Process each word on the line
      - SPLIT, FOR LOOP
    - If word never seen before? Add to words and counts
      - IF STATEMENT, UPDATE LIST
    - Update # occurrences using .index and location
      - UPDATE LIST, USE INDEX FUNCTION

From Pseudo to Code

```python
for line in f:
    data = line.strip().split()
    for word in data:
        word = word.lower()
        if word not in words:
            words.append(word)
            counts.append(0)
        location = words.index(word)
        counts[location] += 1
```

Comparing Two Approaches

- Why do we have a loop in a loop?
  - Code mirrors structure:
    - file has lines, lines have words
  - Notice:
    - .strip
    - .split
    - .lower
    - not in
    - .append
    - .index
    - +=

```python
for line in f:
    data = line.strip().split()
    for word in data:
        word = word.lower()
        if word not in words:
            words.append(word)
            counts.append(0)
        location = words.index(word)
        counts[location] += 1
```
Comparing Two Approaches

- Why do we have only one loop?
  - Code mirrors structure, which is better?
  - File is a sequence of characters!!

```python
for word in f.read().lower().split():
    if word not in words:
        words.append(word)
        counts.append(0)
    location = words.index(word)
    counts[location] += 1
```