Module 26

While-Loops
A Motivating Example

try:

    result = input('Number: ')  # get number
    x = float(input)            # convert to float
    print('The next number is ' + str(x+1))

except:

    print('That is not a number! Try again')
    result = input('Number: ')  # What if this crashes?
    x = float(input)
    print('The next number is ' + str(x+1))
The Basic Idea

• Keep asking user until get a number
  ▪ This sounds like using a loop
• However, a for-loop will not work
  ▪ For-loops run a fixed number of times
  ▪ Determined by size of sequence or range
  ▪ You need to run “until done”
• Requires a different type of loop
  ▪ Motivation for the **while-loop**
Beyond Sequences: The while-loop

\[\text{while } <\text{condition}> :\]

\[\begin{align*}
\text{statement 1} \\
\ldots \\
\text{statement n}
\end{align*}\]

\[\text{true} \quad \text{false} \]

- 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

Vs For-Loop
Solving Our Original Problem

```python
loop = True
while loop:
    try:
        result = input('Number: ')  # get number
        x = float(input)           # convert to float
        print('The next number is ' + str(x+1))
        loop = False
    except:
        print('That is not a number! Try again')
```
Important Concept in CS: Doing Things Repeatedly

1. Process each item in a sequence
   - Compute aggregate statistics for a dataset, such as the mean, median, standard deviation, etc.
   - Send everyone in a Facebook group an appointment time

2. Perform $n$ trials or get $n$ samples.
   - A4: draw a triangle six times to make a hexagon
   - Run a protein-folding simulation for $10^6$ time steps

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

for x in sequence:
    process x

for x in range(n):
    do next thing
Important Concept in CS: Doing Things Repeatedly

1. Process each item in a sequence
   - Compute aggregate statistics for a dataset, such as the mean, median, standard deviation, etc.
   - Send everyone in a Facebook group an appointment time

2. Perform $n$ trials or get $n$ samples.
   - A4: Draw a triangle six times to make a hexagon
   - Run a protein-folding simulation for $10^6$ time steps

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

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

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

```python
while condition:
    keep going
```
# while Versus for

## For-Loop

```python
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

```python
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
```

**Demo in Tutor**

**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')

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
Replacing the Range Iterable

\textbf{range}(a,b)

\begin{align*}
i &= a \\
\text{\textbf{while } } i &< b: \\
&\quad \text{process integer } i \\
&\quad i = i + 1
\end{align*}

\begin{itemize}
\item # store in count # of '/'s in String s
\item count = 0
\item i = 0
\item \textbf{while } i < \text{len}(s):
\item \quad \text{if } s[i] == '/':
\item \qquad count = count + 1
\item \qquad i = i + 1
\end{itemize}

\begin{itemize}
\item # count is # of '/'s in s[0..s.length()-1]
\end{itemize}

\textbf{range}(c,d-1)

\begin{align*}
i &= c \\
\text{\textbf{while } } i &\leq d: \\
&\quad \text{process integer } i \\
&\quad i = i + 1
\end{align*}

\begin{itemize}
\item # Store in double var. \(v\) the sum
\item \# \(1/1 + 1/2 + \ldots + 1/n\)
\item \(v = 0; \quad # \text{call this 1/0 for today}\)
\item i = 1
\item \textbf{while } i \leq n:
\item \quad v = v + 1.0 / i
\item \quad i = i + 1
\end{itemize}

\begin{itemize}
\item # \(v = 1/1 + 1/2 + \ldots + 1/n\)
\end{itemize}
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
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

```python
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 \texttt{while} vs \texttt{for}

\begin{itemize}
\item \texttt{# table of squares to N}
\begin{verbatim}
seq = []
n = floor(sqrt(N)) + 1
for k in range(n):
    seq.append(k*k)
\end{verbatim}
\end{itemize}

\begin{itemize}
\item \texttt{# table of squares to N}
\begin{verbatim}
seq = []
k = 0
while k*k < N:
    seq.append(k*k)
k = k+1
\end{verbatim}
\end{itemize}

A for-loop requires that you know where to stop the loop \textbf{ahead of time}

A while loop can use complex expressions to check if the loop is done
Advantages of **while vs for**

Fibonacci numbers:

\[
\begin{align*}
F_0 &= 1 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2}
\end{align*}
\]

# Table of n Fibonacci nums

```python
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

```python
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
Defrem3(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)

Be careful when you modify the loop variable

A: [2]
B: [3]
C: [3,2]
D: []
E: something else
Difficulties with *while*

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

```python
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]
    >>> foo(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

```python
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
```

```python
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.

Stopping point keeps changing
Application: Convergence

• How to implement this function?

```python
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
Newton’s Method

- Newton’s Method uses the poly derivative
  - Gives formula for computing next step
    - $x_{n+1} = x_n/2 + c/2x_n$
  - Won’t give details for this formula
- How to use the method?
  - Start with guess $x_0 = c$
  - Compute $x_1$ using formula above
  - Continue until $x_n$ is good enough
Example: Sqrt(2)

• Actual answer: 1.414235624

• \( x_{n+1} = x_n/2 + c/2x_n \)

• \( x_0 = 1 \)

• \( x_1 = 0.5 + 1 = 1.5 \)

• \( x_2 = 0.75 + 2/3 = 1.41666 \)

• \( x_3 = 0.7083 + 2/2.833 = 1.41425 \)
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 \( \sqrt{c}^2 = c \)

• So square approximation and compare
  ▪ \texttt{while} \( x^2 \) is not close enough to \( c \)
  ▪ \texttt{while} \( \text{abs}(x^2 - c) > \text{threshold} \)
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 $\sqrt{c}^2 = c$

• So square approximation and compare

While-loop computes until the answer converges
def sqrt(c, err=1e-6):
    """Returns: \sqrt{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
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 Primary Function

```python
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
```
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
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