Computational Thinking

Lecture 11a: While Loop

University of Engineering and Technology
VIETNAM NATIONAL UNIVERSITY HANOI

Outline

Previously:

- For Loops
- Recursion

Today:

- While-loops
- The "Four Loopy Questions"
- The break-statement

Readings:

Think Python, 2nd ed., 7.3 – 7.5

While-Loops

Guessing Game:

```
> python game.py
What number am I thinking of?
```

- **At first:** always answered 'wrong'
- **With if-statements:** could say whether right or wrong, but only give a **single** guess
- **With for-loops:** could give a **bounded** number of guesses, e.g. up to 5
- Could we give the player an **unbounded** number of guesses: keep guessing until they get it right?

Review: A Single Guess

```
# game.py
the_num = random.randint(1, 100)

def play():
    try:
        guess = input('What number am I thinking of? ')
        num = int(guess)
        if num != the_num:
            print('That is not correct.')
        else: #num == the_num
            print('Correct!')
    except ValueError:
        print('Sorry, you did not enter a number.')
```

For-Loop: Bounded Guesses

```
# game.py
def play():
    still_playing = True
    for i in range(5): # 5 guesses
        if still_playing:
            try:
                guess = input('What number am I thinking of?')
                num = int(guess)
                if num != the_num:
                    print('That is not correct.')
                else: #num == the_num
                    print('Correct!')
                    still_playing = False
            except ValueError:
                print('Sorry, you did not enter a number.')
```

Initialize flag

Boolean **flag** variable: indicates whether or not to do something

Change flag

While-Loop: Unbounded Guesses

```
# game.py
def play():
    still_playing = True
    while still_playing:
        try:
            guess = input('What number am I thinking of?')
            num = int(guess)
            if num != the_num:
                print('That is not correct.')
            else: #num == the_num
                print('Correct!')
                still_playing = False
        except ValueError:
            print('Sorry, you did not enter a number.')
```

Flag indicates whether to keep executing the loop

Change flag

While-Loops

Syntax

```
while <expr>:
    <statements>
```

Example

```
while still_playing:
    <statements>
```

Execution:

- Check whether <expr> is **`True`** vs. **`False`**.
  - If it is **`False`**, stop executing the loop. Move on to next unindented statement.
  - If it is **`True`**, execute the indented statements.
- Check whether <expr> is **`True`** vs. **`False`**.
  - If it is **`False`**, stop executing the loop. Move on to next unindented statement.
  - If it is **`True`**, execute the indented statements.
- (keep repeating that, possibly forever, until loop execution stops)

Terminology

Syntax

```
while <expr>:
    <statements>
```

Example

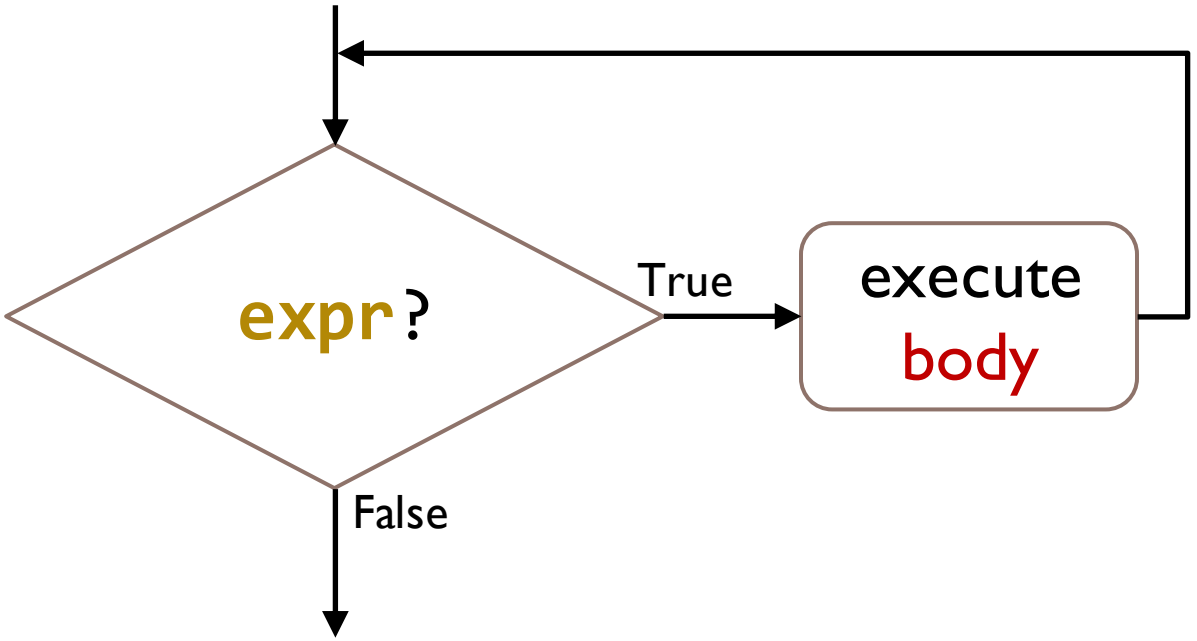
```
while still_playing:
    ...
```

loop **condition** aka **guard**

loop **body**

Execution Flowchart

```
while expr:
    body
```



Infinite Loops

- An **infinite loop** is a loop that never stops executing
- Press Control+C to cancel execution of the program

```
# loops.py
def infinite_loop():
    x = 0
    while True:
        print(x)
        x = x + 1
```

```
>>> import loops
>>> loops.infinite_loop()
0
1
2
...
^C
KeyboardInterrupt
```

Question

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

- What is printed?

A: Nothing: infinite loop
B: 8
C: 12
D: 4
E: 0

While-Loop Design: The Four Questions

Q1. Does it start right?

Any variables the loop will access need to be initialized to the right values.

Q2. Does it maintain the proper relationship* among the variables?

Each variable the loop uses might need to have its value updated with each iteration. All the variables must stay consistent with each other.

Q3. Does it make progress?

Each iteration of the loop body should work toward making the loop condition become `False`.

Q4. Does it stop right?

When the loop condition is finally `False` and the loop stops, the loop must have achieved its goal.

* That relationship is aka the **loop invariant**

Loop Design: Guessing Game

```
# game.py
def play():
    still_playing = True
    while still_playing:
        try:
            guess = input('What number am I thinking of?')
            num = int(guess)
            if num != the_num:
                print('That is not correct.')
            else: #num == the_num
                print('Correct!')
                still_playing = False
        except ValueError:
            print('Sorry, you did not enter a number.')
```

Q1. Starts right: we're playing

Q3. Makes progress: gives the player a chance to win

Q2. Maintains relationships: update still_playing based on values of num and the_num

Q4. Stops right: when we stop playing, num and the_num are the same

While-Loop Examples

Ex. 1: Simulation of Dice Rolling

```
# loops.py
def snake_eyes():
    """Roll a pair of dice until they both come up 1s."""
    snake_eyes = False
    num_tries = 0
    while not snake_eyes:
        r1 = roll()
        r2 = roll()
        print('Rolled: ' + str(r1) + ' and ' + str(r2))
        num_tries = num_tries + 1
        if r1 == 1 and r2 == 1:
            snake_eyes = True
    print('Snake eyes!')
    print('It took ' + str(num_tries) + ' tries.')
```

Initialize accumulator **(Q1: starts right)**

Another **flag**.

can read "while not" as "until"

Accumulator maintained by the loop. **(Q3: makes progress)**

Relationship between flag and rolls maintained. **(Q2: maintains relationships)**

Report accumulator's final value when both rolls are 1s. **(LQ4: stops right)**

Ex. 2: Financial Calculation

```
# loops.py
def print_growth(period, amount):
    """Print one line of table."""
    # ...
def double_investment(initial, rate_pct):
    """Print table of investment growth until doubled."""
    amount = initial
    period = 0
    print_growth(period, amount)
    while amount < 2*initial:
        amount = (1 + rate_pct/100) * amount
        period = period + 1
        print_growth(period, amount)
```

Two **accumulators** maintained by loop

No **flag** variable;
instead, directly check
whether stopping **condition**
has been reached

Ex. 3: Syracuse Sequence

```
# loops.py
def syr(start):
    """Returns the Syracuse sequence."""
    x = start
    lst = []
    while x != 1:
        lst.append(x)
        if x%2 == 0:
            x = x//2
        else:
            x = 3*x + 1
    lst.append(x)
    return lst
```

Accumulator maintained by loop

Directly check stopping **condition**

```
f(n)
 = 1      if n is 1
 = n/2    if n is even
 = 3n+1   if n is odd
```

Ex. 3: Syracuse Sequence

```
# loops.py
def syr(start):
    """Returns the Syracuse sequence."""
    x = start
    lst = []
    while x != 1:
        lst.append(x)
        if x%2 == 0:
            x = x//2
        else:
            x = 3*x + 1
    lst.append(x)
    return lst
```

Accumulator maintained by loop

Directly check stopping **condition**

```
f(n)
 = 1      if n is 1
 = n/2    if n is even
 = 3n+1   if n is odd
```

Full Circle: Recursion vs. Iteration

- **Recursion:** the programming pattern of using recursive functions
- **Iteration:** the programming pattern of using loops
- Both accomplish the task of repetition
  - Loop: repeat body of loop
  - Recursion: repeat body of function
- Recursion is **strictly more powerful** than iteration with for-loops*
- Recursion is **equivalent in power** to iteration with while-loops**
- What about the power of while- vs. for-loops?
  - We already saw that unbounded guessing could be done in the number game with a while-loop but not a for-loop, so while-loops can do some things for-loops cannot
  - Is the reverse true: are there computations you can do with a for-loop but not with a while-loop...?

The Break Statement

The Break Statement

- Inside a loop body a **break** statement causes execution of the* loop to stop
- Because they tend to be badly abused by beginners, you should be **very careful** when using the break statements

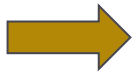
* Only the **innermost** loop stops if there are nested loops

```
# loops.py
# get input from user and print
still_going = True
while still_going:
    x = input('? ')
    if x == 'quit':
        still_going = False
    else:
        print(x)
```

```
# same behavior
while True:
    x = input('? ')
    if x == 'quit':
        break
    print(x)
```

For-loops Translate to While-loops

```
for x in iterable:
    <body>
```



```
iterator = iter(iterable)
while True:
    try:
        x = next(iterator)
    except StopIteration:
        break
    <body>
```

Every computation you can do with a for-loop you can also do with a while-loop. While-loops are therefore **strictly more powerful** than for-loops. We never needed for-loops, but they are quite useful!