Middlebury

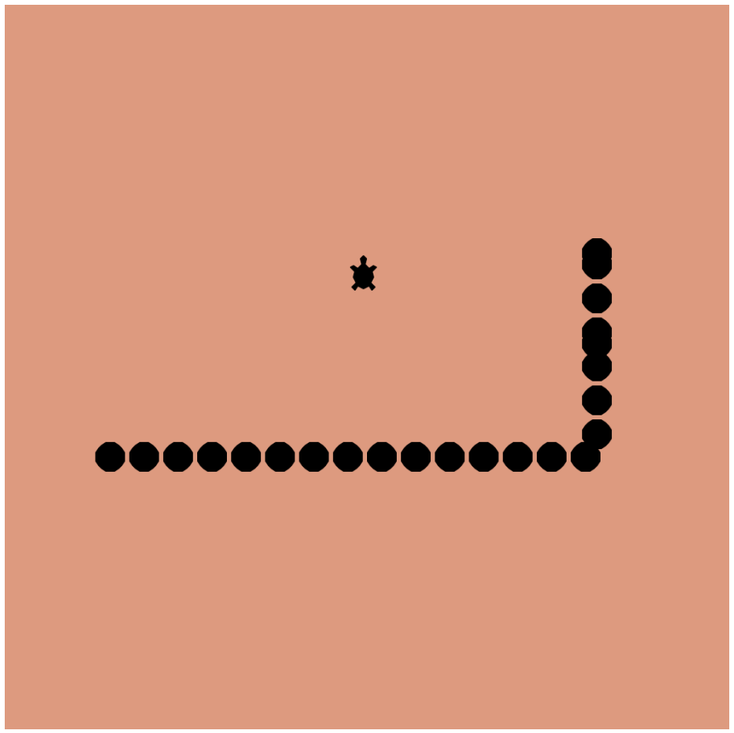
CSCI 146: Intensive Introduction to Computing

Fall 2025

Lecture 8: `while` loops

Goals for today

- Use relational operators to compare values.
- Describe the execution of `for` and `while` loops and differentiate between them.
- Use `break` and `continue` keywords to further control iterative algorithms.
- Appropriately choose between `for` and `while` loops for a computational problem.



Note that we can also use non-`bool` variables as conditions (but I don't recommend this).

```
if "a string?":
  print("Do I get executed?")
```

- "falsy" values: `False`, `0`, `0.0`, `None`, empty sequences (`""`, `[]`)
- "truthy" values: everything else

Recall a question from last class: `a == b or a == 5` is **not** the same as `a == b or 5` (i.e. `==` isn't distributive).

What about? `a == (b or 5)`

Python (and other programming languages) will "short-circuit" your conditionals, so be careful!

```
1 def i_really_want_this_function_to_be_called():
2   print("WHY WON'T THIS PRINT")
3
4 if True or i_really_want_this_function_to_be_called():
5   print("oh well")
6
7 if False and i_really_want_this_function_to_be_called():
8   print("this won't print either, the conditional is already false")
```

The code above only prints `"oh well"`.

We can also compare strings, which is done in lexicographic order (alphabetical by character).

Main idea:

- Compare first character, if not the same, return result of < between characters.
- Compare second character, if not the same, return result of < between characters.
- ... continue comparing characters ...
- If one is a substring of another, it is "less than" the other.

```
>>> "Aardvark" < "Zebra"
>>> "aardvark" < "Zebra"
```

Investigate the built-in `ord` function!

97	141	61	01100001	a	&#97;		Lowercase a
98	142	62	01100010	b	&#98;		Lowercase b
99	143	63	01100011	c	&#99;		Lowercase c
100	144	64	01100100	d	&#100;		Lowercase d
101	145	65	01100101	e	&#101;		Lowercase e
102	146	66	01100110	f	&#102;		Lowercase f
103	147	67	01100111	g	&#103;		Lowercase g
104	150	68	01101000	h	&#104;		Lowercase h
105	151	69	01101001	i	&#105;		Lowercase i
106	152	6A	01101010	j	&#106;		Lowercase j
107	153	6B	01101011	k	&#107;		Lowercase k
108	154	6C	01101100	l	&#108;		Lowercase l
109	155	6D	01101101	m	&#109;		Lowercase m
110	156	6E	01101110	n	&#110;		Lowercase n
111	157	6F	01101111	o	&#111;		Lowercase o
112	160	70	01110000	p	&#112;		Lowercase p
113	161	71	01110001	q	&#113;		Lowercase q
114	162	72	01110010	r	&#114;		Lowercase r
115	163	73	01110011	s	&#115;		Lowercase s
116	164	74	01110100	t	&#116;		Lowercase t
117	165	75	01110101	u	&#117;		Lowercase u
118	166	76	01110110	v	&#118;		Lowercase v
119	167	77	01110111	w	&#119;		Lowercase w
120	170	78	01111000	x	&#120;		Lowercase x
121	171	79	01111001	y	&#121;		Lowercase y
122	172	7A	01111010	z	&#122;		Lowercase z

`while` loops are useful for iterating when we don't know how many iterations to do.

```
while condition:
    statement1
    statement2
statement3
statement4
```

```
while condition1:
    statement1
    statement2
    if condition2:
        break
statement3
statement4
```

Make sure your `while` loops terminate!
Iterations should eventually reach your exit condition.

Question 1: What does this code print?

```
n = 3
while n > 0:
    if (n == 5):
        n = -99
    print(n)
    n = n + 1
```

A.

```
3
4
```

B.

```
3
4
5
```

C.

```
3
4
-99
```

D.

```
3
4
5
-99
```

Question 2: creating a valid password

A valid password is one that is length `5` and starts with `"xy"`. A valid password should terminate the loop. Which of these implements that specification? Note, the `input` function prints its argument as a prompt and returns whatever the user types as a string (after the user hits enter).

A.

```
while True:
    s = input("Enter a password: ")
    if len(s) == 5 and s[:2] == 'xy':
        break
```

B.

```
s = input("Enter a password: ")
while len(s) == 5 and s[:2] == 'xy':
    s = input ("Enter a password: ")
```

C. Both A and B are correct.

D. Neither A or B are correct.

Question 3: Will this loop terminate, be guaranteed to be an infinite loop or will it depend?

```
1 a = 0
2 i = 0
3 while i < 10:
4     a = a + 1
```

1. Terminate or not execute.
2. Infinite loop.
3. Depends.
4. `SyntaxError`

Example: guessing game

1. Start by investigating what the current game is doing.
2. Make a copy of the game function called `number_guessing_game2` and refactor the code to remove the `correct` variable.
3. Make a copy of the game function called `number_guessing_game3` and refactor the code to remove the `correct` variable and not use a `break` keyword.

`for` loops versus `while` loops: when to pick one over the other?

```for
for i in range(2, 11, 2):
    print(i)
```

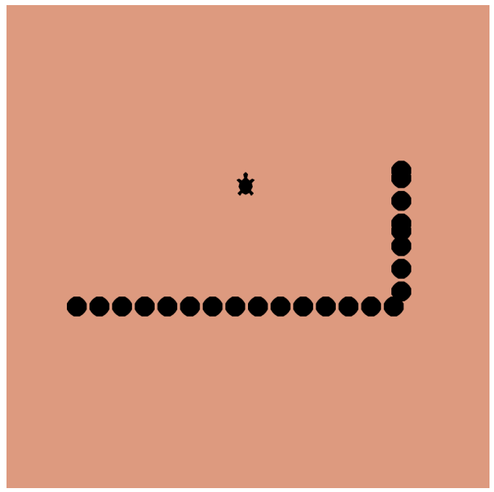
```while
i = 2
while i <= 10:
    print(i)
    i += 2
```

Another way?

```
for i in range(2, 11):
    if i % 2 == 1:
        continue
    print(i)
```

- Pick a `for` loop if you know the number of iterations (e.g. from a variable or a constant) beforehand `AND` the loop variable increments in the same way on each iteration.
- Pick a `while` loop if (1) you don't know the number of iterations beforehand `OR` (2) the loop variable might update differently on each iteration.

A better way to program our snake game.



```python
def game_over():
    """
    Determines if the game is over.
    """
    # Here, we just determine if the player is on the screen, but other options could be:
    # 1. Use a time limit on the game.
    # 2. Check if the snake somehow intersected itself, which may not be allowed.
    return not inbounds()
```

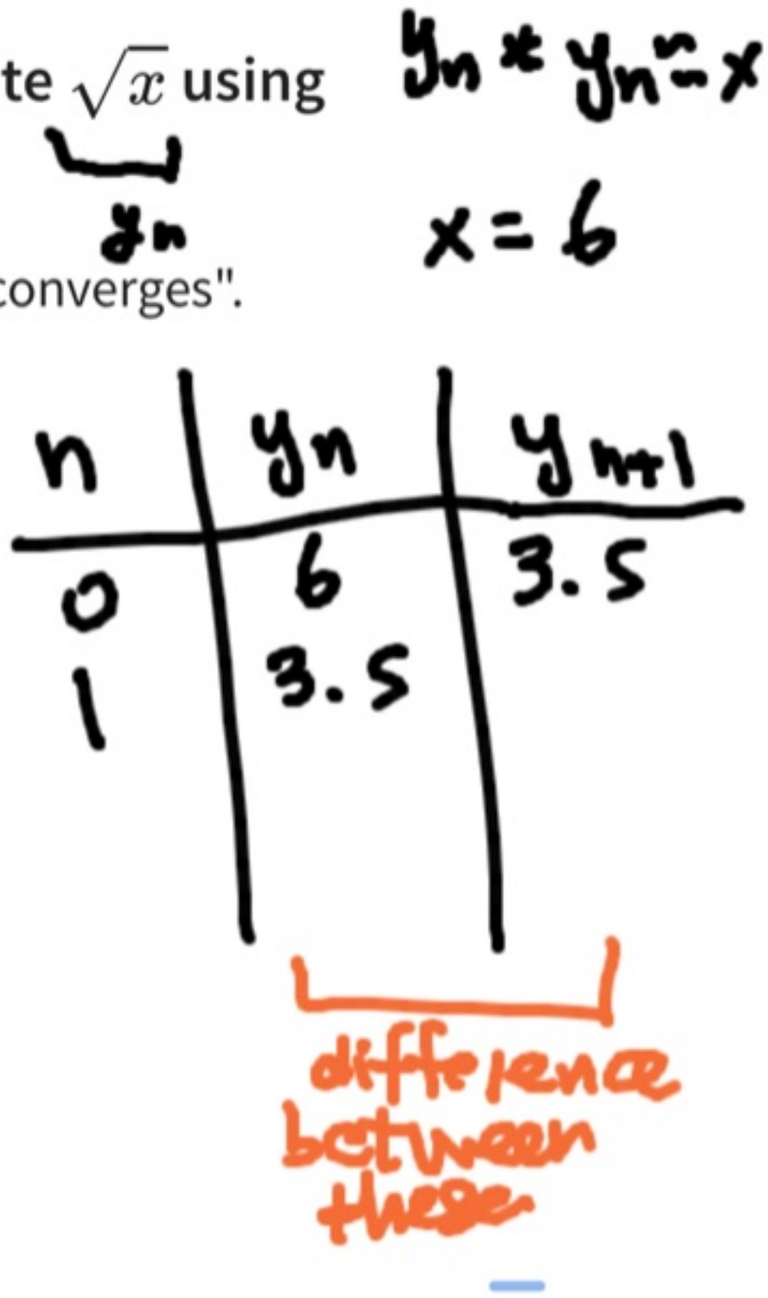
```python
while not game_over():
    ...
```

Exercise: write a function called `diy_sqrt` to calculate $\sqrt{x}$ using Heron's method.

Start with $y_0 = x$ and update $y_{n+1} = \frac{1}{2}\left(y_n + \frac{x}{y_n}\right)$ until it "converges".

- How will you know when the answer y_n is close to the answer?
- Think about writing a condition that expresses this.
- Then write the body of a loop (and decide on a `for` or `while` loop).

```
1  TOLERANCE = 1e-12 # scientific notation for 10 ** (-12)
2  def diy_sqrt(x):
3    """
4    Calculates the square root of some number x using Heron's method.
5
6    Args:
7      x: number to calculate the square root of. MUST BE >= 0.
8
9    Returns:
10     Square-root of x.
11   """
12   y_n = x
13   while abs(y_n * y_n - x) > TOLERANCE:
14     y_n = 0.5 * (y_n + x / y_n)
15   return y_n
```



Another method: write a function called `diy_sqrt2` to calculate $\sqrt{x}$ using the Bakhshali method.

Start with $y_0 = x$ and iteratively calculate:

1. $a_n = \frac{x - y_n^2}{2y_n}$
2. $z_{n+1} = y_n + a_n$
3. $y_{n+1} = z_{n+1} - \frac{a_n^2}{2z_{n+1}}$
4. Then y_{n+1} becomes the guess for the next iteration until convergence.

```python
TOLERANCE = 1e-12 # scientific notation for 10 ** (-12)
def diy_sqrt2(x):
  """
  Calculates the square root of some number x using the Bakhshali method.

  Args:
    x: number to calculate the square root of. MUST BE >= 0.

  Returns:
    Square-root of x.
  """
  yn = x
  while abs(yn * yn - x) > TOLERANCE:
    an = (x - yn * yn) / (2 * yn)
    yn += an
    yn -= an * an / (2 * yn)
  return yn
```

Reminders

- Midterm 1 on Thursday October 9th at 7:30pm. Please email me if you have a conflict or accommodations. Covers material through this week.
- Have a look at feedback on Programming Assignment 2 (resubmit until Thursday October 9th).
- Programming Assignment 1 due tomorrow (final submission).
- Programming Assignment 3 due tomorrow (initial submission).
- Quiz 4 on Friday with retakes of previous quiz questions.
- Help hours: see go/smith (Smith Gakuya, ASI) and go/cshelp.