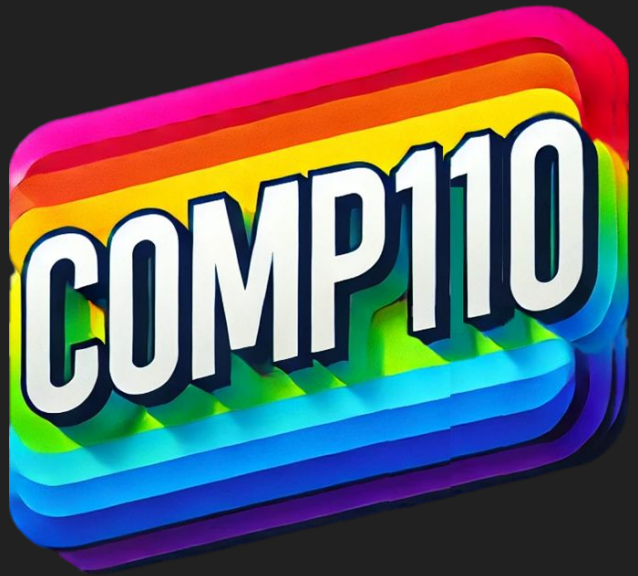

CL09 – Practice with while Loops & Intro to Lists

Review of a common problem: the dreaded *infinite loop*

If a condition in a **`while`** loop never becomes False, the loop will continue indefinitely.

To prevent this:

- Ensure that your loop's condition will eventually be False!

```python
def count_to_n(n: int) -> None:
    count: int = 0
    while count <= n:
        print(f"Count is: {count}")
        count = count + 1


count_to_n(n=4)
```

A common problem: the dreaded *infinite loop*

If a condition in a **`while`** loop never becomes False, the loop will continue indefinitely.

To prevent this:

- Ensure that your loop's condition will eventually be False!

```
1  def count_to_n(n: int) -> None:
2      count: int = 0
3      while count <= n:
4          print(f"Count is: {count}")
5          count = count + 1
6
7
8  count_to_n(n=4)
```

Which line of code in the code listing prevents an *infinite loop* from occurring? What would happen without it?

Common use cases of `while` loops

- **User input validation:** Prompt the user for a valid input until they give one to you!
  - *Think:* our word-guessing game example, or Wordle!
- **Game loops:** Keep a game running until some condition is met
  - Common examples: You run out of lives or attempts
- Iterating through values
  - Examples:
    - Counting from 0 to n ✅
    - Looping through every character in a string (via subscription notation)

Common use cases of `while` loops

- **User input validation:** Prompt the user for a valid input until they give one to you!
  - *Think:* our word-guessing game example, or Wordle!
- **Game loops:** Keep a game running until some condition is met
  - Common examples: You run out of lives or attempts
- Iterating through values
  - Examples:
    - Counting from 0 to n ✅
    - Looping through every character in a string (via subscription notation) ✨

```python
def reverse(a_str: str) -> str:
    """Reverse a string"""
    idx: int = 0
    result: str = ""
    while idx < len(a_str):
        result = a_str[idx] + result
        idx = idx + 1

    return result


print(reverse(a_str="abc"))
```

Announcements

CQ02 due today at 11:59pm

- Please take a photo or scan it and submit to Gradescope

EX02 (Wordle) due Sunday, June 1 at 11:59pm

- You'll be writing 4 functions to make Wordle!

Quiz 02 on Friday (May 30)

- Question about what we've covered thus far? Please visit Office Hours!
- Practice quiz will be posted today

Warm-Up: Memory Diagram

```python
"""A countdown program..."""


def main() -> None:
    seconds: int = 3
    countdown(seconds)
    print(f"main {seconds}")


def countdown(seconds: int) -> None:
    print("T minus")
    while seconds > 0:
        print(seconds)
        seconds = seconds - 1

    print(f"countdown {seconds}")


main()
```

```python
"""A countdown program..."""


def main() -> None:
    seconds: int = 3
    countdown(seconds)
    print(f"main {seconds}")


def countdown(seconds: int) -> None:
    print("T minus")
    while seconds > 0:
        print(seconds)
        seconds = seconds - 1

    print(f"countdown {seconds}")


main()
```

Relative Reassignment Operators

It's *very* common to need to update the value of a variable, relative to its current value, e.g.:

```
count: int = 1
```

```
count = count + 1
```

Relative reassignment operators offer a shorthand way of doing this!

```
count += 1
```

Relative Reassignment Operators

```
1   """A countdown program..."""
2
3
4   def main() -> None:
5       seconds: int = 3
6       countdown(seconds)
7       print(f"main {seconds}")
8
9
10  def countdown(seconds: int) -> None:
11      print("T minus")
12      while seconds > 0:
13          print(seconds)
14          seconds = seconds - 1
15
16      print(f"countdown {seconds}")
17
18
19  main()
```

Try writing line 14 using a relative reassignment operator!

Your task: Convert this recursive function to one that uses a while loop!

```python
def safe_icarus(x: int) -> int:
    """Bound aspirations!"""
    if x >= 2:
        return 1
    else:
        return 1 + safe_icarus(x=x + 1)

print(safe_icarus(x=0))
```

A nested while loop!

```python
def triangle(n: int) -> None:
    i: int = 1
    line: str
    while i <= n:
        line = ""
        while len(line) < i:
            line += "*"
        print(line)
        i += 1


triangle(2)
```

Lists

Examples of lists:

- To-do list
- Assignment due dates
- Grocery list

A list is a **data structure**–something that lets you organize and store data in a format such that they can be accessed and processed efficiently.

Lists are **mutable**, meaning their values can be changed after initialization.

NOTE: Lists can be an arbitrary (but finite) length! (Not a fixed number of items.)

Lists are Mutable Sequences in Python

Sequences are ordered, 0-indexed collections of values

Feature	Syntax	Purpose
Type Declaration		
Constructor (function)		
List Literal		
Access Value		
Assign Item		
Length of List		

Your job: Complete this table as we cover each topic today. Once you're finished, submit a .PDF of it to Gradescope! (blank copy on next slide)

Lists are Mutable Sequences in Python

Sequences are ordered, 0-indexed collections of values

Feature	Syntax	Purpose
Type Declaration		
Constructor (function)		
List Literal		
Access Value		
Assign Item		
Length of List		

Declaring the type of a list

<list name>: list[<item type>]

grocery_list: list[str]

Declaring the type of a list

<list name>: list[<item type>]

grocery_list: list[str]

str, int, float, etc.

Initializing a list

The constructor **list()** is a *function* that returns the literal **[]**

With a constructor:

- <list name>: list[<item type>] = list()
- grocery_list: list[str] = list()

With a literal:

- <list name>: list[<item type>] = []
- grocery_list: list[str] = []

declare variable initialize list

"create a var called grocery_list, a list of strings, which will initially be empty"

Initializing a list

The constructor **list()** is a *function* that returns the literal **[]**

With a constructor:

- <list name>: list[<item type>] = list()
- grocery_list: list[str] = list()

With a literal:

- <list name>: list[<item type>] = []
- grocery_list: list[str] = [“apples”, “bananas”, “pears”]

declare variable

initialize list

“create a var called grocery_list, a list of strings, which will initially contain these values”

Initializing a list

With a constructor:

- <list name>: list[<item type>] = list()
- grocery_list: list[str] = list()

With a literal:

- <list name>: list[<item type>] = []
- grocery_list: list[str] = []

The constructor **list()** is a *function* that returns the literal **[]**

Bringing it back to something we know, you can create an empty string using the constructor **str()** or the literal **“”**

Initializing a list

With a constructor:

- <list name>: list[<item type>] = list()
- grocery_list: list[str] = list()

With a literal:

- <list name>: list[<item type>] = []
- grocery_list: list[str] = []

The constructor **list()** is a *function* that returns the literal **[]**

Bringing it back to something we know, you can create an empty string using the constructor **str()** or the literal **“”**

Let’s try it!
Create an empty list of floats with the name my_numbers.

Adding an item to the end of a list

<list name>.append(<item>)

grocery_list.append(“bananas”)

Adding an item to the end of a list

<list name>.append(<item>)

grocery_list.append(“bananas”)

- Method: a function that *belongs* to the **list** class
- Like calling append(grocery_list, “bananas”)

Adding an item to the end of a list

<list name>.append(<item>)

grocery_list.append(“bananas”)

- Method: a function that *belongs* to the **list** class
- Like calling append(grocery_list, “bananas”)

Let’s try it!
Add the value 1.5 to my_numbers.

Initializing an already populated list

<list name>: list[<item type>] = [<item 0>, <item 1>, … , <item n>]

grocery_list: list[str] = [“bananas”, “milk”, “bread”]

Initializing an already populated list

<list name>: list[<item type>] = [<item 0>, <item 1>, … , <item n>]

grocery_list: list[str] = [“bananas”, “milk”, “bread”]

Let’s try it!
Create a list called game_points that stores the following numbers: 102, 86, 94

Indexing

grocery_list: list[str] = [“bananas”, “milk”, “bread”]

grocery_list[0]

***Starts at 0, like with strings!*

Indexing

grocery_list: list[str] = [“bananas”, “milk”, “bread”]

grocery_list[0]

***Starts at 0, like with strings!*

Let’s try it!
In game_points, use subscription notation to print out 94.

Modifying by index

grocery_list: list[str] = [“bananas”, “milk”, “bread”]

grocery_list[1] = “eggs”

Modifying by index

grocery_list: list[str] = [“bananas”, “milk”, “bread”]

grocery_list[1] = “eggs”

Let’s try it!
In game_points, use subscription notation to change 86 to 72.

Modifying by index

grocery_list: list[str] = [“bananas”, “milk”, “bread”]

grocery_list[1] = “eggs”

Let’s try it!
In game_points, use subscription notation to change 86 to 72.

Question: Could you do this type of modification with a string? Try it out!

Length of a list

grocery_list: list[str] = [“eggs”, “milk”, “bread”]

len(grocery_list)

Length of a list

grocery_list: list[str] = [“eggs”, “milk”, “bread”]

len(grocery_list)

Let’s try it!
Print the length of game_points.

Remove an item from a list – “pop off!”

grocery_list: list[str] = [“eggs”, “milk”, “bread”]

grocery_list.pop(2)

Index of item you want to remove

Remove an item from a list – “pop off!”

grocery_list: list[str] = [“eggs”, “milk”, “bread”]

grocery_list.pop(2)

Index of item you want to remove

Before: [“eggs”, “milk”, ~~“bread”~~]

Index: 0 1 2

After: [“eggs”, “milk”]

Index: 0 1

Remove an item from a list – “pop off!”

grocery_list: list[str] = [“eggs”, “milk”, “bread”]

grocery_list.pop(1)

Index of item you want to remove

Before: [“eggs”, ~~“milk”~~, “bread”]

Index: 0 1 2

After: [“eggs”, “bread”]

Index: 0 1

Remove an item from a list – “pop off!”

grocery_list: list[str] = [“eggs”, “milk”, “bread”]

grocery_list.pop(2)

Index of item you want to remove

Let’s try it!
Remove 72 from game_points.