

Hack110 Sign-Up Form!

When? Saturday, April 5th from 10 AM - 12 AM (Midnight)

Where? In Sitterson Lower Lobby

Who can join? Anyone in COMP 110! No prior experience required. Bring a partner or come as yourself (we'll have team-building activities if you want a partner)

Come for a fun day of coding, workshops and events (**food and CLE credit will be provided**):

- Choose between web development or game development track
- Go to various **workshops & events** such as: Navigating the CS Major, Resume workshop, ice cream station, and kahoot trivia and MORE!
- Link: Sign-Up Here! Or via the QR code 
- **Sign-Up form EXTENDED TO Monday, March 31st at 11:59 pm**
 - Spots are limited! So we'll prioritize interest!
 - If you have a partner, **ONLY ONE OF YOU** has to sign up - you will just enter your partner's info in the form.

Sign-Up Here!





CL24: Time Complexity and Practice with Sets and Dictionaries

Announcements

- Quiz 02 grades were released on Friday – median ~85%!
 - *Please submit regrade requests by **this Friday at 11:59pm***
- LS11 – Dictionaries due today
- EX03 due *this Wednesday* (March 26)!
- Quiz 03 on Friday, March 28
 - Review Session on Wednesday (March 26) at 6:15pm in Fred Brooks (FB) 009
 - Other ways to prep:
 - Pause to consider how you've been studying for the quizzes; what has helped and what hasn't?
 - Review Quiz 02 to address your gaps in understanding
 - Finish EX03 and review your code – try to diagram example function calls!
 - Practice Quiz
 - Please visit us in Office Hours + Tutoring!

Review of the previous lecture

```
1  def intersection(a: list[str], b: list[str]) -> list[str]:
2      result: list[str] = []
3
4      idx_a: int = 0
5      while idx_a < len(a):
6          idx_b: int = 0
7          found: bool = False
8          while not found and idx_b < len(b):
9              if a[idx_a] == b[idx_b]:
10                 found = True
11                 result.append(a[idx_a])
12                 idx_b += 1
13             idx_a += 1
14
15     return result
16
17
18 foo: list[str] = ["a", "b"]
19 bar: list[str] = ["c", "b"]
20 print(intersection(foo, bar))
```

Assume our unit of "operation" is the number of times the block of lines #9-12 are evaluated.

Q1. Can different values of a and b lead to a difference in the number of operations required for the intersection function evaluation to complete?

Q2. If so, provide example item values for a and b which require the fewest operations to complete? Then try for the maximal operations to complete?

Q3. Assuming the item values of a and b are random and unpredictable, about how many operations does this function take to complete?

As the lengths of **a** and **b** grow, the number of operations grows *quadratically*

```
1 def intersection(a: list[str], b: list[str]) -> list[str]:
2     result: list[str] = []
3
4     idx_a: int = 0
5     while idx_a < len(a):
6         idx_b: int = 0
7         found: bool = False
8         while not found and idx_b < len(b):
9             if a[idx_a] == b[idx_b]:
10                 found = True
11                 result.append(a[idx_a])
12                 idx_b += 1
13             idx_a += 1
14
15     return result
16
17
18 foo: list[str] = ["a", "b"]
19 bar: list[str] = ["c", "b"]
20 print(intersection(foo, bar))
```

- Outer while loop iterates through each element of **a**
 - If there are N elements, we'll iterate N times
- And within each iteration of the outer while loop...
- The inner while loop iterates through elements of **b** until either:
 - We find a value that $==$ the current element in **a** OR,
 - We have “visited” (accessed) every element in **b**
 - If there are M elements in **b**, we'll iterate up to M times

Assuming **a** and **b** both have 3 elements...

1. Example of values of **a** and **b** that will cause the **fewest** operations to occur?
`intersection(a=["a", "a", "a"], b=["a", "b", "c"])`
2. Example of values of **a** and **b** that will cause the **most** operations to occur?
`intersection(a=["a", "b", "c"], b=["d", "e", "f"])`

If list **a** has N elements and list **b** has M elements, the “worst case scenario” is that this code will cause $N \cdot M$ operations to occur.

Comparing lists and sets

```
1 def intersection(a: list[str], b: list[str]) -> list[str]:
2     result: list[str] = []
3
4     idx_a: int = 0
5     while idx_a < len(a):
6         if a[idx_a] in b:
7             result.append(a[idx_a])
8             idx_a += 1
9
10    return result
```

```
1 def intersection(a: list[str], b: set[str]) -> set[str]:
2     result: set[str] = set()
3
4     idx_a: int = 0
5     while idx_a < len(a):
6         if a[idx_a] in b:
7             result.add(a[idx_a])
8             idx_a += 1
9
10    return result
```

Suppose **a** and **b** each had 1,000,000 elements. The worst case difference here is approximately 1,000,000 operations, versus $1,000,000^{**}2$ or 1,000,000,000,000 operations.

If your device can perform 100,000,000 operations per second, then...

A call to **a** will complete in 2.78 hours and **b** will complete in 1/100th of a second.

Let's explore Set syntax in VSCode together...

In your cl directory, add a file named cl24_dictionaries.py with the following starter:

```
"""Examples of set and dictionary syntax."""
```

```
pids: set[int] = {710000000, 712345678}
```

One quirk about sets: to add a value, use the `.add()` method!

Try evaluating the following expression:

```
pids.add(730120710)
```

Let's explore Dictionary syntax in VSCode together...

In your `cl24_dictionaries.py` file, add the code:

```
ice_cream: dict[str, int] = {  
    "chocolate": 12,  
    "vanilla": 8,  
    "strawberry": 4,  
}
```

Try evaluating the following expression:

```
ice_cream["vanilla"] += 110
```

Syntax

Data type:

```
name: dict[<key type>, <value type>]  
temps: dict[str, float]
```

Construct an empty dict:

```
temps: dict[str, float] = dict() or  
temps: dict[str, float] = {}
```

Construct a populated dict:

```
temps: dict[str, float] = {"Florida": 72.5, "Raleigh": 56.0}
```

Let's try it!

Create a dictionary called
ice_cream that stores the
following orders

Keys	Values
chocolate	12
vanilla	8
strawberry	5

Length of dictionary

`len(<dict name>)`

`len(temps)`

Let's try it!

Print out the length of ice_cream.

What exactly is this telling you?

Adding elements

We use subscription notation.

`<dict name>[<key>] = <value>`

`temps["DC"] = 52.1`

Let's try it!

Add 3 orders of "mint" to your ice_cream dictionary.

Access + Modify

To access a value,
use subscription notation:

```
<dict name>[<key>]  
temps["DC"]
```

To modify, also use subscription notation:

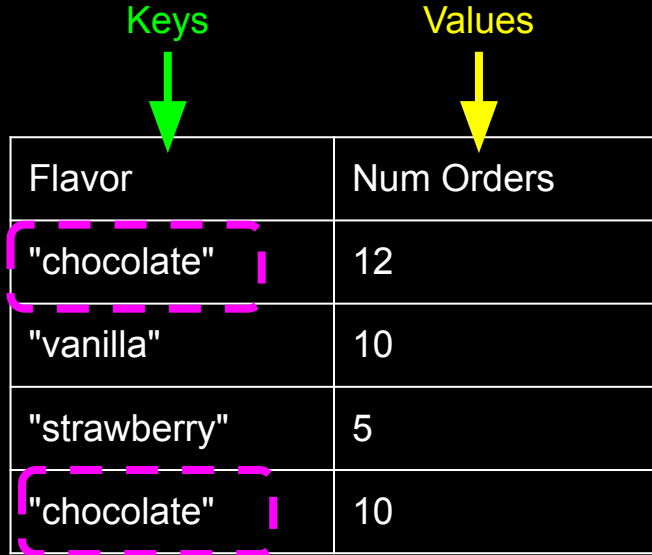
```
<dict name>[<key>] = new_value  
temps["DC"] = 53.1 or temps["DC"] += 1
```

Let's try it!

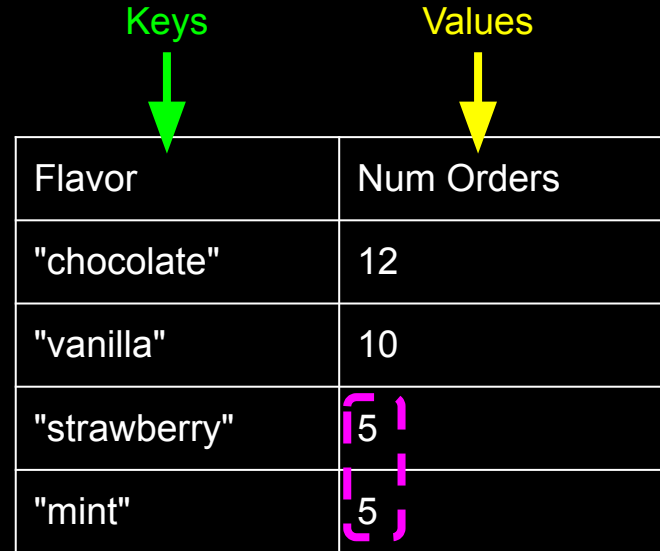
Print out how many orders there
are of "chocolate".
Update the number of orders of
Vanilla to 10.

Important Note: Can't Have Multiple of Same Key

(Duplicate values are okay.)



Flavor	Num Orders
"chocolate"	12
"vanilla"	10
"strawberry"	5
"chocolate"	10



Flavor	Num Orders
"chocolate"	12
"vanilla"	10
"strawberry"	5
"mint"	5

Check if key in dictionary

`<key> in <dict name>`

`"DC" in temps`

`"Florida" in temps`

Let's try it!

Check if both the flavors "mint" and "chocolate" are in ice_cream.

Write a conditional that behaves the following way:
If "mint" is in ice_cream, print out how many orders of "mint" there are.
If it's not, print "no orders of mint".

Removing elements

Similar to lists, we use pop()

```
<dict name>.pop(<key>)
```

```
temps.pop("Florida")
```

Let's try it!

Remove the orders of "strawberry"
from ice_cream.

"for" Loops

"for" loops iterate over the **keys** by default

Let's try it!

Use a for loop to print:
chocolate has 12 orders.
vanilla has 10 orders.
strawberry has 5 orders.

```
for key in ice_cream:  
    print(key)
```

```
for key in ice_cream:  
    print(ice_cream[key])
```

Flavor	Num Orders
"chocolate"	12
"vanilla"	10
"strawberry"	5

Final Notes

This is the code we worked through together in class, for reference.

```
1  """Examples of dictionary syntax with Ice Cream Shop order tallies."""
2
3  # Dictionary type is dict[key_type, value_type].
4  # Dictionary literals are curly brackets
5  # that surround with key:value pairs.
6  ice_cream: dict[str, int] = {
7      "chocolate": 12,
8      "vanilla": 8,
9      "strawberry": 4,
10 }
11
12 # len evaluates to number of key-value entries
13 print(f"{len(ice_cream)} flavors")
14
15 # Add key-value entries using subscription notation
16 ice_cream["mint"] = 3
17
18 # Access values by their key using subscription
19 print(ice_cream["chocolate"])
20
21 # Re-assign values by their key using assignment
22 ice_cream["vanilla"] += 10
23
24 # Remove items by key using the pop method
25 ice_cream.pop("strawberry")
26
27 # Loop through items using for-in loops
28 total_orders: int = 0
29 # The variable (e.g. flavor) iterates over
30 # each key one-by-one in the dictionary.
31 for flavor in ice_cream:
32     print(f"{flavor}: {ice_cream[flavor]}")
33     total_orders += ice_cream[flavor]
34
35 print(f"Total orders: {total_orders}")
```