

ASK THE RECRUITER: UNC CS CAREER PANEL

Learn how to stand out as an applicant, the dos and don'ts of reaching out, recruiting timelines, and more!

Plus, you will have a chance to ask your questions and network following the event.



pendo Sas Vanguard



college of arts and sciences

Computer Science

CO-HOSTED BY:





OCT 1
6:30 PM | SN014





Sets and Dictionaries

Announcements

- CQ02 (released on Friday) due today at 11:59pm
 - Follow along with the <u>video</u> and submit your completed memory diagram to Gradescope
 - o If you submitted a different memory diagram, please resubmit with this memory diagram
- EX03 List Utils due today at 11:59pm
- Quiz 01 regrade requests open till Wednesday at 11:59pm
- Quiz 02 on Friday, Oct 10
 - If you will have a university-approved absence on 10/10 and you want to take the quiz, let me know ahead of time
 - o If you take your quizzes with ARS, please ensure you are scheduled to take it with them

Limits of Lists for collections of data (1/2)

Using a list, we could store everyone in COMP110's PID associated with ONYEN

list[str]	
Index	Value
0	""
1	""
710,453,081 items elided	
710453084	"krisj"
9,857,700 i	tems elided
720310785	"abyrnes1"
9,809,924 i	tems elided
730120710	"ihinks"

Warm-up question:
Why does using a list[str] feel wrong/inefficient?

Limits of Lists for collections of data (2/2)

onyens:

list[str]	
Index	Value
0	"ihinks"
1	"abyrnes1"
2	"sjiang3"
296 items elided	
299	"krisj"

pids:

list[int]	
Index	Value
0	730120710
1	720310785
2	730820837
296 items elided	
299	710453084

Suppose we model ONYENs and PIDs with lists. One list has ONYENs, the other has the person's PID at the same index.

Given the onyen "sjiang3", how do you algorithmically find their PID?

We could use the in operator (a new concept)...

```
# Pretend we initialized pids to hold all of our PIDs
pids: list[int] = [700000000, 700000001, 700000002, ..., 710453084, 730120710]

pids_of_interest: list[int] = [710453084, 730120710]

idx: int = 0

while idx < len(pids_of_interest):

if pids_of_interest[idx] in pids:

print("We found a PID in the list!")

idx += 1</pre>
```

... but try to avoid using it on lists!

Enter: sets!

Sets, like lists, are a data structure for storing collections of values.

Unlike lists, sets are unordered and each value has to be unique.

Lists: always zero-based, sequential, integer indices!

Benefit of sets: testing for the existence of an item takes only one "operation," regardless of the set's size.

```
pids: set[int] = {730120710, 730234567, 730000000}
```

Great! ... But what if we want to associate people's PIDs with their ONYENs in a data structure?

Enter: Dictionaries!

Dictionaries, like lists, are a *data structure* for storing collections of values.

Unlike lists, dictionaries give *you* the ability to decide what to *index* your data by.

Lists: *always* zero-based, sequential, integer indices!

Dictionaries are indexed by <u>keys</u> associated with <u>values</u>. This is a unique, one-way mapping!

Analogous: A real-world dictionary's <u>keys</u> are words and associated <u>values</u> are definitions.

pid_to_onyen:

dict[int, str]	
key	value
730120710	"ihinks"
710453084	"krisj"
720310785	"abyrnes1"

onyen_to_seat:

dict[str, str]	
key	value
"ihinks"	"A1"
"abyrnes1"	"A2"
"sjiang3"	"A3"
"krisj"	"N17"

Let's diagram key concepts

```
# USD exchange rate to other currencies
     exchange: dict[str, float] = {
         "CNY": 7.10, # Chinese Yuan
         "GBP": 0.77, # British Pound
         "DKK": 6.86, # Danish Kroner
     dollars: float = 100.0
     # Access dictionary value by its key
11
     pounds: float = dollars * exchange["GBP"]
12
     # Append a key-value entry to dictionary
13
     exchange["EUR"] = 0.92
15
     # Update a key-value entry in dictionary
     exchange["CNY"] -= 1.00
17
     # len is the number of key-value entries
19
     count: int = len(exchange)
```

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

In your cl directory, add a file named cl22_dictionaries.py with the following starter:

```
"""Examples of dictionary syntax with Ice Cream Shop order tallies."""
ice_cream: dict[str, int] = {
   "chocolate": 12,
   "vanilla": 8,
   "strawberry": 4,
}
```

Save, then open up this file in Trailhead's REPL and we will explore key syntax together. Ready to go? 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] = {}

Let's try it!

Create a dictionary called ice_cream that stores the following orders

Keys	Values
chocolate	12
vanilla	8
strawberry	5

Construct a populated dict:

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

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:

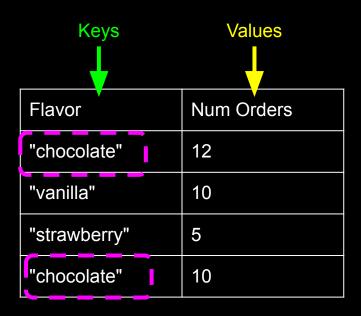
```
<dict name>[<key>] = new_value
temps["DC"] = 53.1 or temps["DC"] += 1
```

Let's try it!

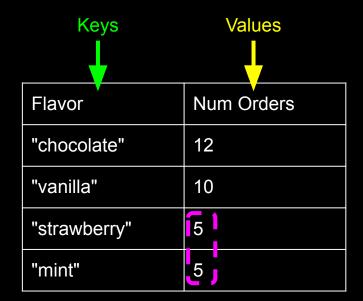
Print out how many orders there are of "chocolate".

Update the number of orders of Vanilla to 10.

Important Note: Can't Have Multiple of Same Key



(Duplicate *values* are okay.)



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(ice_cream[key])

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

This is the code we wrote together, for reference.

```
"""Examples of dictionary syntax with Ice Cream Shop order tallies."""
# Dictionary literals are curly brackets
# that surround with key:value pairs.
ice cream: dict[str, int] = {
    "chocolate": 12.
    "vanilla": 8,
    "strawberry": 4,
# len evaluates to number of key-value entries
print(f"{len(ice_cream)} flavors")
# Add key-value entries using subscription notation
ice_cream["mint"] = 3
# Access values by their key using subscription
print(ice cream["chocolate"])
# Re-assign values by their key using assignment
ice_cream["vanilla"] += 10
# Remove items by key using the pop method
ice_cream.pop("strawberry")
# Loop through items using for-in loops
total_orders: int = 0
# The variable (e.g. flavor) iterates over
# each key one-by-one in the dictionary.
for flavor in ice cream:
    print(f"{flavor}: {ice_cream[flavor]}")
    total_orders += ice_cream[flavor]
print(f"Total orders: {total_orders}")
```

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

```
def intersection(a: list[str], b: list[str]) -> list[str]:
    result: list[str] = []
    idx_a: int = 0
    while idx_a < len(a):
        idx b: int = 0
        found: bool = False
        while not found and idx_b < len(b):
            if a[idx_a] == b[idx_b]:
                found = True
                result.append(a[idx_a])
            idx b += 1
        idx a += 1
    return result
foo: list[str] = ["a", "b"]
bar: list[str] = ["c", "b"]
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...

- Example of values of a and b that will cause the fewest operations to occur?
 intersection(a=["a", "a", "a"], b=["a", "b", "c"])
 - 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*M operations to occur.

Comparing lists and sets

```
def intersection(a: list[str], b: set[str]) -> set[str]:
def intersection(a: list[str], b: list[str]) -> list[str]:
                                                                       result: set[str] = set()
    result: list[str] = []
                                                                       idx a: int = 0
    idx_a: int = 0
                                                                       while idx_a < len(a):
    while idx a < len(a):
                                                                           if a[idx_a] in b:
        if a[idx_a] in b:
                                                                               result.add(a[idx a])
            result.append(a[idx a])
                                                                           idx a += 1
        idx a += 1
                                                                       return result
    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 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.