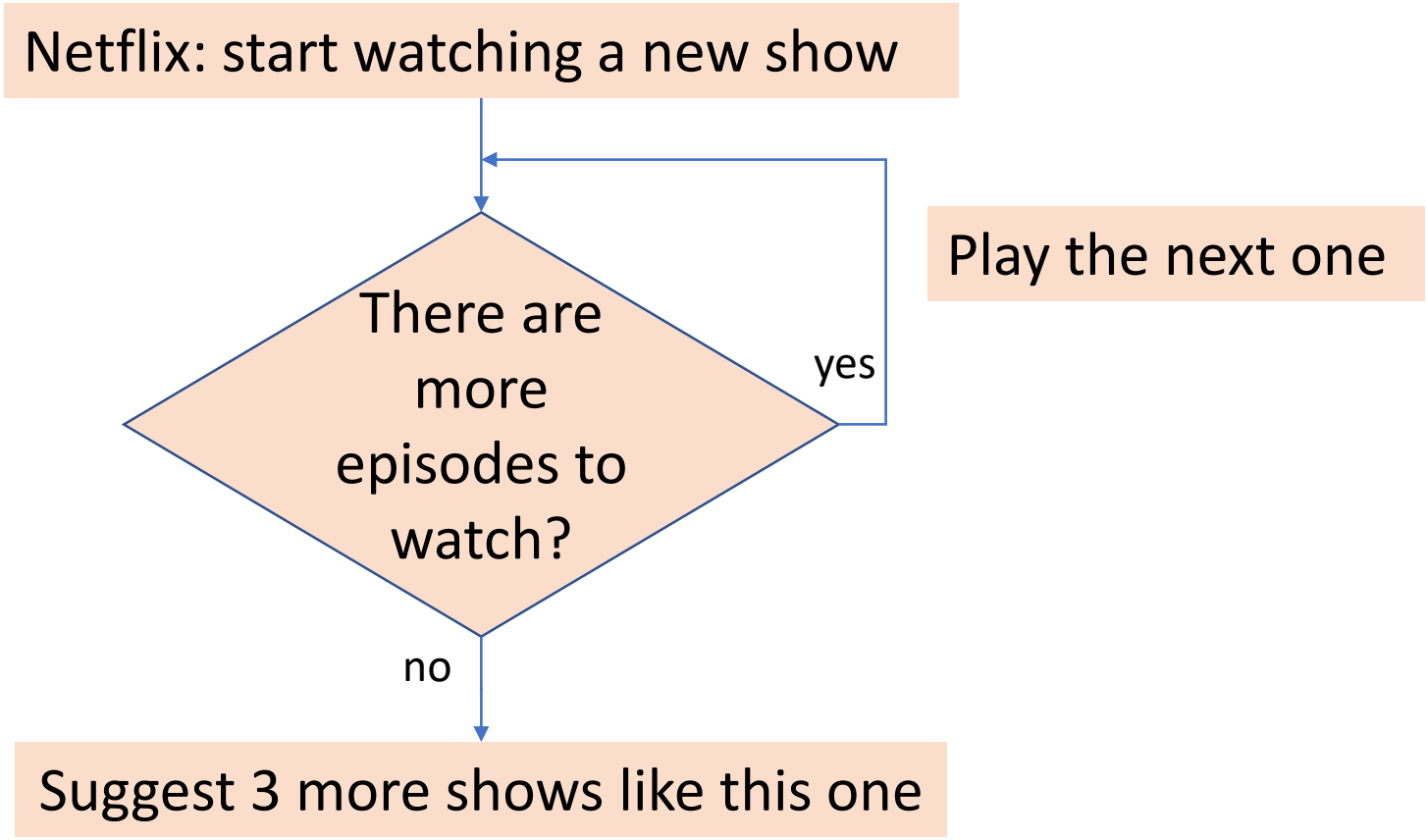
Lecture 2

Python Loops and non-scalar variables

`while` LOOPS

BINGE ALL EPISODES OF ONE SHOW



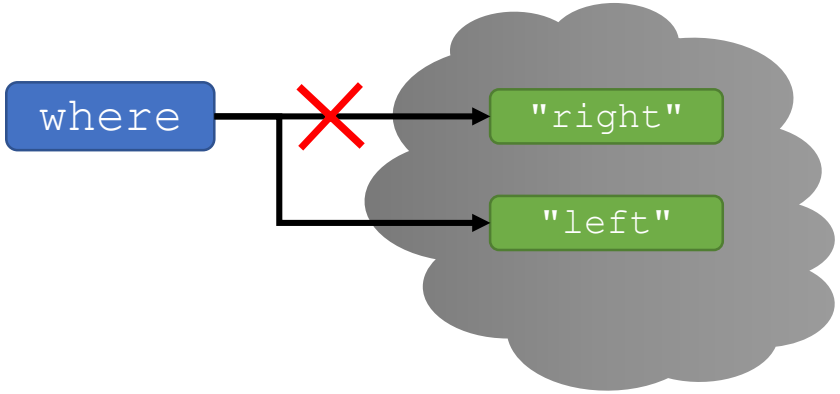
CONTROL FLOW: while LOOPS

```
while <condition>:
    <code>
    <code>
    ...
```

- `<condition>` **evaluates to a Boolean**
- If `<condition>` is `True`, **execute all the steps inside** the while code block
- **Check** `<condition>` again
- **Repeat** until `<condition>` is `False`
- If `<condition>` is never `False`, then will loop forever!!

`while` LOOP EXAMPLE

```
You are in the Lost Forest.
**********
**********
   ☺
**********
**********
Go left or right?
```



PROGRAM:

```
where = input("You're in the Lost Forest. Go left or right? ")
while where == "right":
    where = input("You're in the Lost Forest. Go left or right? ")
print("You got out of the Lost Forest!")
```

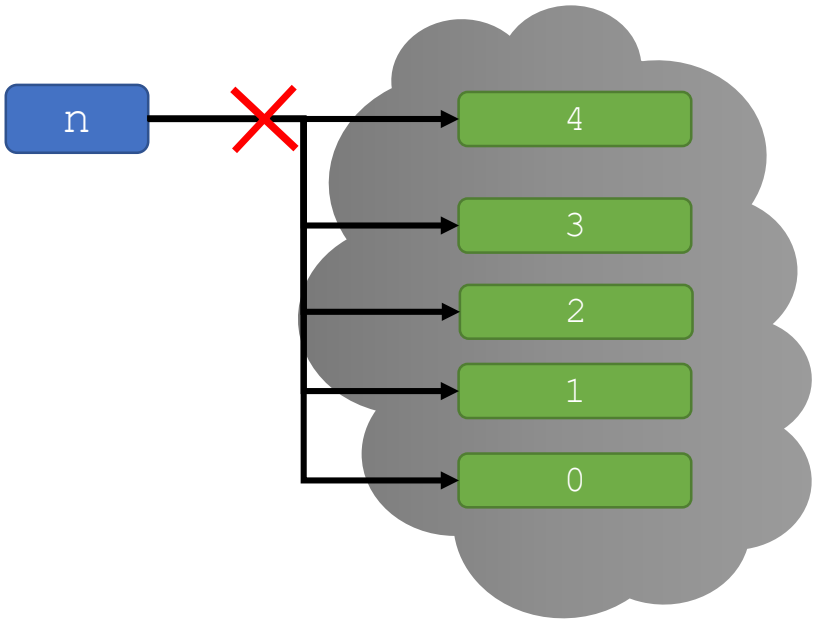
YOU TRY IT!

- What is printed when you type "RIGHT"?

```
where = input("Go left or right? ")
while where == "right":
    where = input("Go left or right? ")
print("You got out!")
```

while LOOP EXAMPLE

```
n = int(input("Enter a non-negative integer: "))
while n > 0:
    print('x')
    n = n-1
```



while LOOP EXAMPLE

```
n = int(input("Enter a non-negative integer: "))
while n > 0:
    print('x')
    n = n-1
```

What happens without this last line?
Try it!

- To terminate:
  - Hit CTRL-c or CMD-c in the shell
  - Click the red square in the shell

YOU TRY IT!

- Run this code and stop the infinite loop in your IDE

```
while True:
    print("noooooo")
```

BIG IDEA

`while` loops can repeat code inside indefinitely!

Sometimes they need your intervention to end the program.

YOU TRY IT!

- Expand this code to show a sad face when the user entered the while loop more than 2 times.
- Hint: use a variable as a counter

```
where = input("Go left or right? ")
while where == "right":
    where = input("Go left or right? ")
print("You got out!")
```

CONTROL FLOW: while LOOPS

- Iterate through **numbers in a sequence**

Set loop variable outside while loop

```
n = 0
while n < 5:
    print(n)
    n = n+1
```

Test loop variable in condition

Increment loop variable inside while loop

n = n+1
equivalent to
n += 1

A COMMON PATTERN

- Find 4!
- `i` is our loop variable
- `factorial` keeps track of the product

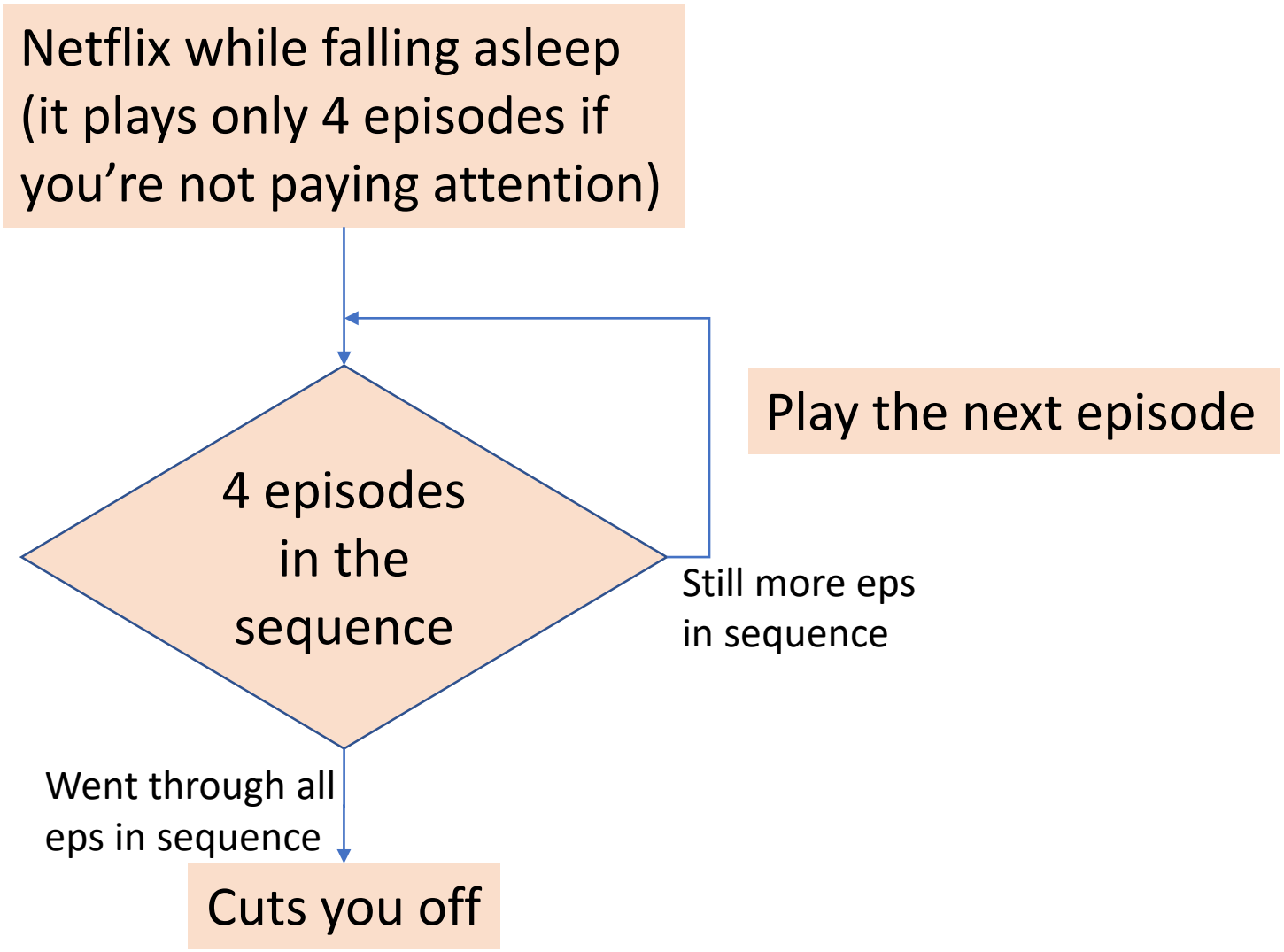
```
x = 4
i = 1
factorial = 1
while i <= x:
    factorial *= i
    i += 1
print(f'{x} factorial is {factorial}')
```

Set loop variable outside while loop
Initialize the factorial product to 1
Test loop variable in condition
Keep a running product (eq to factorial = factorial*i)
Increment loop variable inside while loop (eq to i = i+1)

- Python Tutor LINK

`for` LOOPS

ARE YOU STILL WATCHING?



CONTROL FLOW: `while` and `for` LOOPS

- Iterate through **numbers in a sequence**

```
# very verbose with while loop
n = 0
while n < 5:
    print(n)
    n = n+1


# shortcut with for loop
for n in range(5):
    print(n)
```

STRUCTURE of `for` LOOPS

```
for <variable> in <sequence of values>:
    <code>
    ...
```

- **Each time through the loop**, `<variable>` takes a value

- First time, `<variable>` is the **first value in sequence**
- Next time, `<variable>` gets the **second value**
- etc. until `<variable>` runs out of values

A COMMON SEQUENCE of VALUES

```
for <variable> in range(<some_num>):
    <code>
    <code>
    ...
```

Sequence is 0 then 1 then 2 then 3 then 4

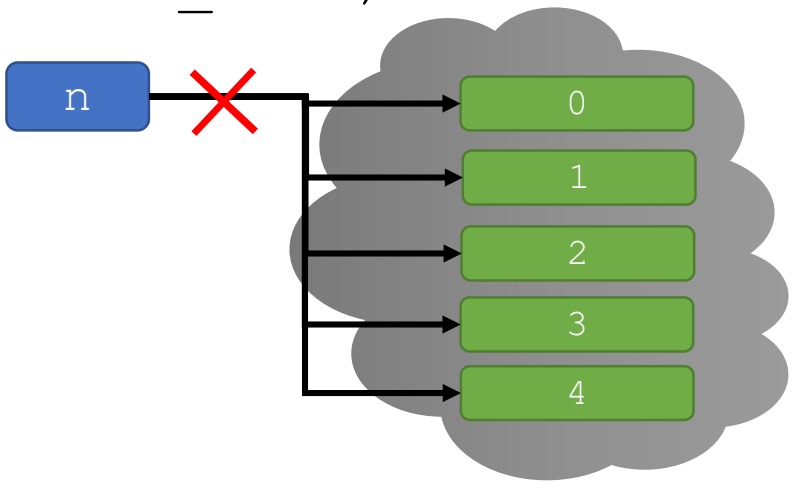
```
for n in range(5):
    print(n)
```

- **Each time through the loop**, `<variable>` takes a value
- First time, `<variable>` **starts at 0**
- Next time, `<variable>` gets the value **1**
- Then, `<variable>` gets the value **2**
- **…**
- etc. until `<variable>` gets **some_num -1**

A COMMON SEQUENCE of VALUES

```
for <variable> in range(<some_num>):
    <code>
    <code>
    ...
```

```
for n in range(5):
    print(n)
```



- **Each time through the loop**, `<variable>` takes a value
- First time, `<variable>` **starts at 0**
- Next time, `<variable>` gets the value **1**
- Then, `<variable>` gets the value **2**
- **…**
- etc. until `<variable>` gets **some_num -1**

`range`

- Generates a **sequence** of ints, following a pattern
- `range(start, stop, step)`
  - `start`: first int generated
  - `stop`: controls last int generated (go up to but not including this int)
  - `step`: used to generate next int in sequence
- A lot like what we saw for **slicing**
- Often omit start and step
  - e.g., `for i in range(4):`
    - `start` defaults to 0
    - `step` defaults to 1
  - e.g., `for i in range(3,5):`
    - `step` defaults to 1

Remember strings? It had a similar syntax, but with colons not commas and square brackets not parentheses.

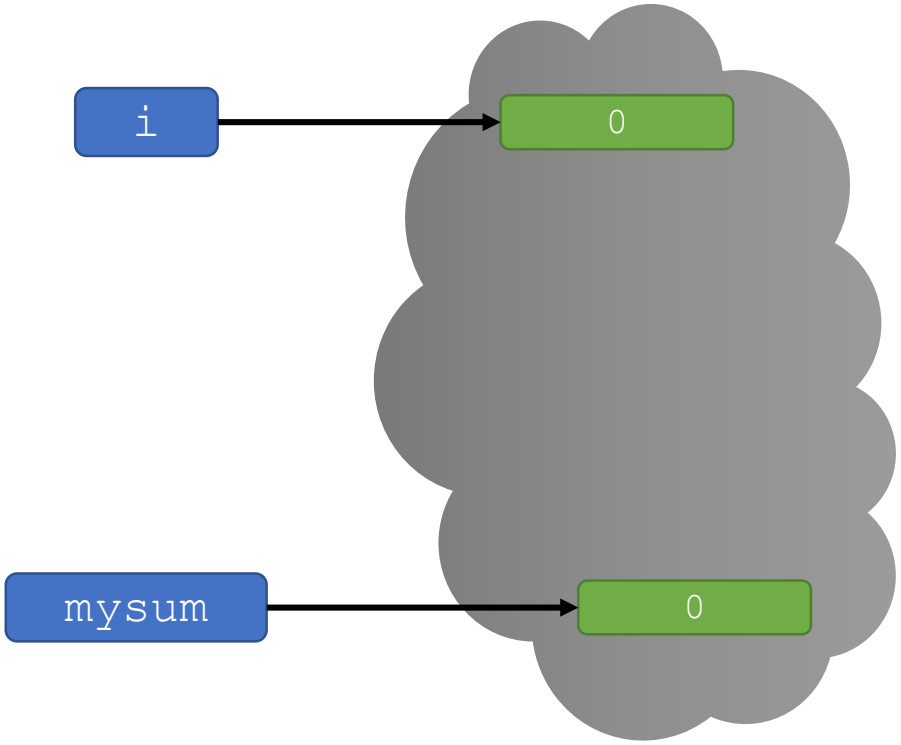
YOU TRY IT!

- What do these print?
- ```
  for i in range(1,4,1):
      print(i)
  ```
- ```
  for j in range(1,4,2):
      print(j*2)
  ```
- ```
  for me in range(4,0,-1):
      print("$"*me)
  ```
  ```

RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

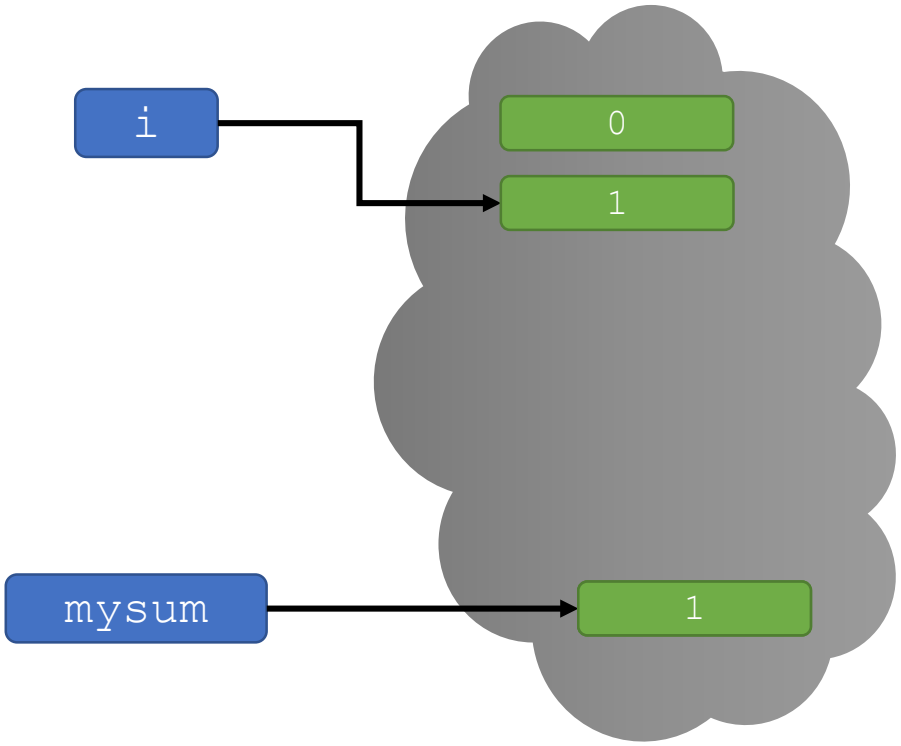
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

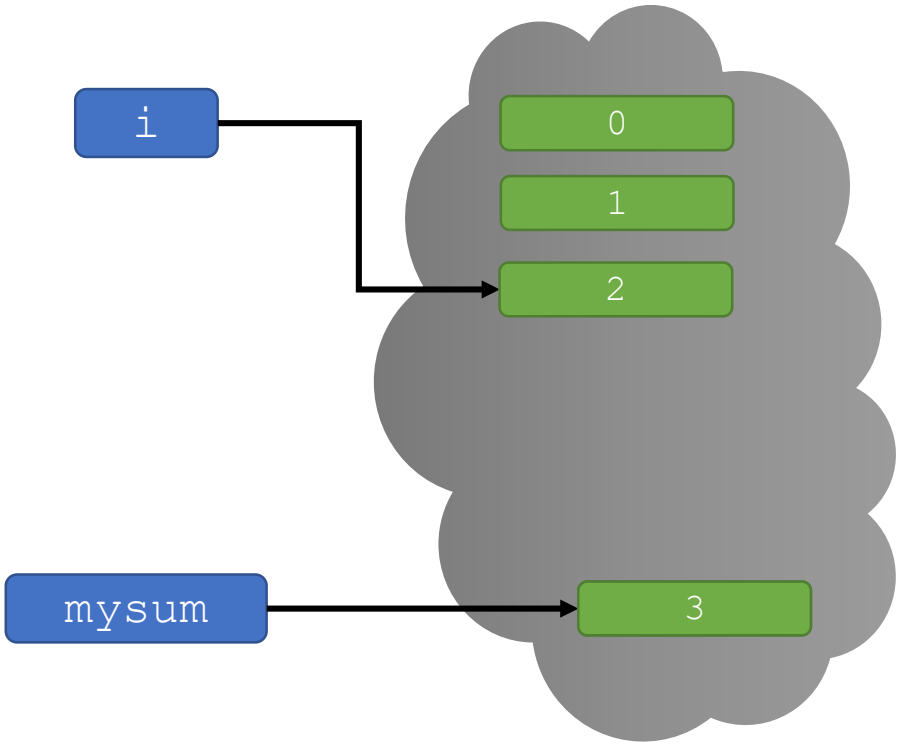
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

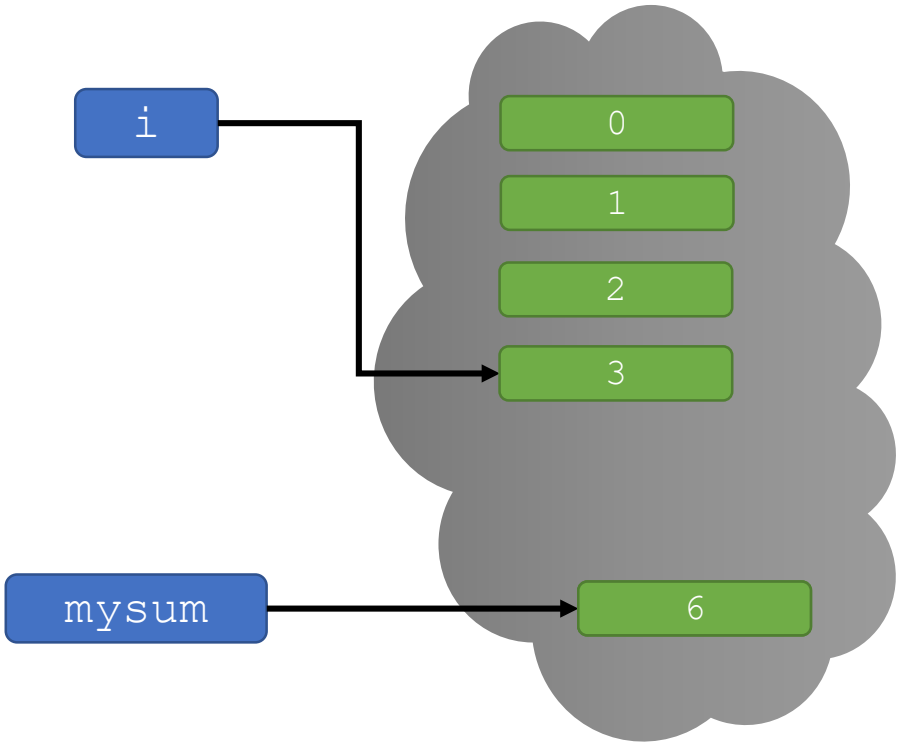
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

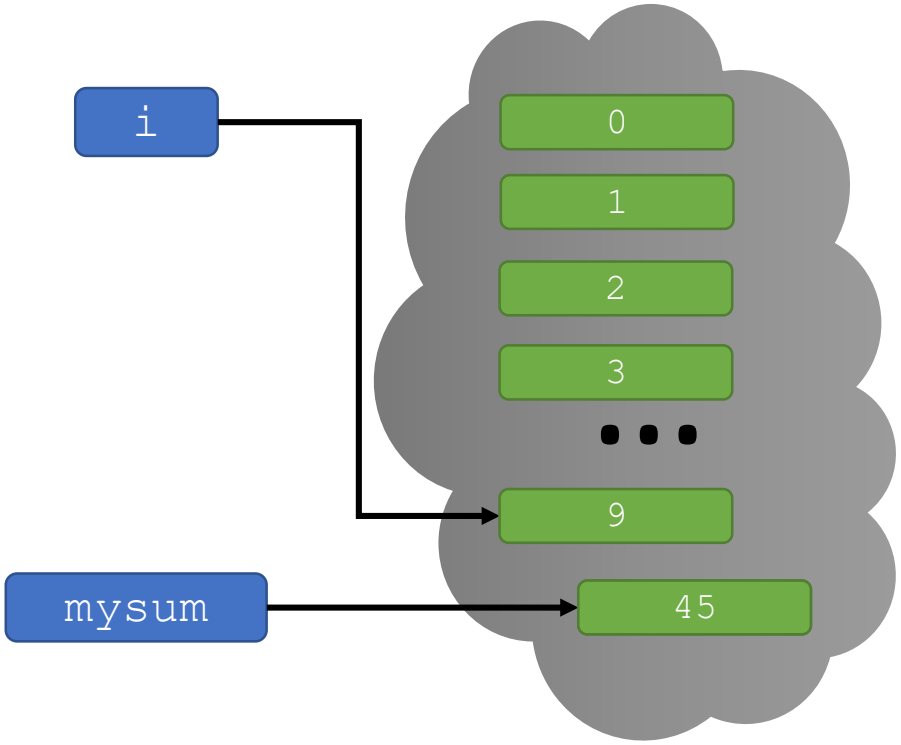
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



YOU TRY IT!

- Fix this code to use variables `start` and `end` in the `range`, to get the total sum between and including those values.
- For example, if `start=3` and `end=5` then the sum should be 12.

```
mysum = 0
start = ??
end = ??
for i in range(start, end):
    mysum += i
print(mysum)
```

`for` LOOPS and `range`

- Factorial implemented with a `while` loop (seen this already) and a `for` loop

```
x = 4
i = 1
factorial = 1
while i <= x:
    factorial *= i
    i += 1
print(f'{x} factorial is {factorial}’)
```

Uses a while loop

```
x = 4
factorial = 1
for i in range(1, x+1, 1):
    factorial *= i
print(f'{x} factorial is {factorial}')
```

Uses a for loop

BIG IDEA

`for` loops only repeat for however long the sequence is

The loop variables takes on these values in order.

`break` STATEMENT

- Immediately exits whatever loop it is in
- Skips remaining expressions in code block
- **Exits only innermost loop**!

```
while <condition_1>:
    while <condition_2>:
        <expression_a>
        break
        <expression_b>
    <expression_c>
```

Evaluated when <condition_1> and <condition_2> are True

Never evaluated (don't write code like this)

Evaluated when <condition_1> is True

`break` STATEMENT

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
    if mysum == 5:
        break
        mysum += 1
print(mysum)
```

- What happens in this program?
- Python Tutor LINK

YOU TRY IT!

- Write code that loops a `for` loop over some range and prints how many even numbers are in that range. Try it with:
  - `range(5)`
  - `range(10)`
  - `range(2,9,3)`
  - `range(-4,6,2)`
  - `range(5,6)`

STRINGS and LOOPS

- Code to check for letter i or u in a string.
- All 3 do the same thing

```
s = "demo loops - fruit loops"
for index in range(len(s)):
    if s[index] == 'i' or s[index] == 'u':
        print("There is an i or u")
```

Uses range to iterate through index of s

```
for char in s:
    if char == 'i' or char == 'u':
        print("There is an i or u")
```

Iterates through characters of s directly

```
for char in s:
    if char in 'iu':
        print("There is an i or u")
```

Iterates through characters of s directly (most "pythonic")

BIG IDEA

The sequence of values in a `for` loop isn't limited to numbers

ROBOT CHEERLEADERS

```
an_letters = "aefhilmnorsxAEFHILMNORSX"

word = input("I will cheer for you! Enter a word: ")
times = int(input("Enthusiasm level (1-10): "))

for c in word:
    if c in an_letters:
        print(f'Give me an {c}: {c}')
    else:
        print(f'Give me a {c}: {c}')
print("What's that spell?")
for i in range(times):
    print(word, '!!!')
```

c is a loop variable whose value is each letter that the user gave

i is a loop variable whose value is 0 through times-1, one at a time

YOU TRY IT!

- Assume you are given a string of lowercase letters in variable s. Count how many unique letters there are in the string. For example, if

```
s = "abca"
```

Then your code prints 3.

HINT:
Go through each character in s.
Keep track of ones you've seen in a string variable.
Add characters from s to the seen string variable if they are not already a character in that seen variable.

SUMMARY SO FAR

- Objects have **types**
- Expressions are **evaluated to one value**, and bound to a variable name
- Branching
  - if, else, elif
  - Program executes **one set of code or another**
- Looping mechanisms
  - `while` and `for` loops
  - Code executes repeatedly **while some condition is true**
  - Code executes repeatedly **for all values in a sequence**

TUPLES

A NEW DATA TYPE

- Have seen scalar types: `int`, `float`, `bool`
- Have seen one compound type: `string`
- Want to introduce more general **compound data types**
  - Indexed sequences of elements, which could themselves be compound structures
  - **Tuples** – immutable
  - **Lists** – mutable

- Next lecture, will explore ideas of
  - Mutability
  - Aliasing
  - Cloning

TUPLES

Remember strings?

- **Indexable ordered sequence** of objects
  - Objects can be any type – int, string, tuple, tuple of tuples, …
- Cannot change element values, **immutable**

```
te = ()
ts = (2,)
```

Empty tuple

Extra comma means tuple with one element

Compare with `ts = (2)`

```
t = (2, "mit", 3)
```

Multiple elements in tuple separated by commas

```
t[0]
```
→ evaluates to 2 Indexing starts at 0

```
(2,"mit",3) + (5,6)
```
→evaluates to a new tuple `(2,"mit",3,5,6)`

```
t[1:2]
```
→ slice tuple, evaluates to `("mit",)`

```
t[1:3]
```
→ slice tuple, evaluates to `("mit",3)`

```
len(t)
```
→ evaluates to 3

```
max((3,5,0))
```
→ evaluates 5

Other functions also work, e.g, sum

```
t[1] = 4
```
→ gives error, **can't modify object**

INDICES AND SLICING

Remember strings?

```
seq = (2,'a',4,(1,2))
  index:  0   1   2    3
print(len(seq))          → 4
print(seq[3])            → (1,2)
print(seq[-1])           → (1,2)
print(seq[3][0])         → 1
print(seq[4])            → error
```

An element of a sequence is at an **index**, indices start at 0

```
print(seq[1])            → 'a'
print(seq[-2:]           → (4,(1,2))
print(seq[1:4:2]         → ('a',(1,2))
print(seq[:-1])          → (2,'a',4)
print(seq[1:3])          → ('a',4)
```

Slices extract subsequences.
Indices evaluated from left to right

```
for e in seq:            → 2
    print(e)               a
                           4
                         (1,2)
```

Iterating over sequences

TUPLES

- Conveniently used to **swap** variable values

```
x = 1
y = 2
x = y
y = x
```

```
x = 1
y = 2
temp = x
x = y
y = temp
```

```
x = 1
y = 2
(x, y) = (y, x)
```

First, create tuple
y = 2
x = 1

Then, bind values in new tuple

TUPLES

- Used to **return more than one value** from a function

```
def quotient_and_remainder(x, y):
    q = x // y
    r = x % y
    return (q, r)
```

One object!
(with 2 elements/values)

(3,1)

(3, 1)

```
both = quotient_and_remainder(10,3)
```

1

2

(2, 1)

```
(quot, rem) = quotient_and_remainder(5,2)
```

BIG IDEA

Returning
one object (a tuple)
allows you to return
multiple values (tuple elements)

YOU TRY IT!

- Write a function that meets these specs:
- Hint: remember how to check if a character is in a string?

```
def char_counts(s):
    """ s is a string of lowercase chars
    Return a tuple where the first element is the
    number of vowels in s and the second element
    is the number of consonants in s """
```

VARIABLE NUMBER of ARGUMENTS

- Python has some built-in functions that take variable number of arguments, e.g, `min`
- Python allows a programmer to have same capability, using *** notation**

```
def mean(*args):
    tot = 0
    for a in args:
        tot += a
    return tot/len(args)
```

- `numbers` is bound to a **tuple of the supplied values**
- Example:
  - `mean(1,2,3,4,5,6)`

args → `(1,2,3,4,5,6)`

LISTS

LISTS

- **Indexable ordered sequence** of objects
  - Usually homogeneous (i.e., all integers, all strings, all lists)
  - But can contain mixed types (not common)
- Denoted by **square brackets**, [ ]
- **Mutable**, this means you can change values of specific elements of list

Tuples were ()

Remember tuples are immutable – you **cannot** change element values.
Lists are mutable, you can change them directly.

INDICES and ORDERING

Remember strings and tuples?

```
a_list = []
```
empty list

```
L = [2, 'a', 4, [1,2]]
```

`[1,2]+[3,4]` → evaluates to `[1,2,3,4]`

`len(L)` → evaluates to 4 — Gives length of top level of tuple

`L[0]` → evaluates to 2 — Indexing starts at 0

`L[2]+1` → evaluates to 5

`L[3]` → evaluates to `[1,2]`, another list!

`L[4]` → gives an error

```
i = 2
```

`L[i-1]` → evaluates to `'a'` since `L[1]='a'`

`max([3,5,0])` → evaluates 5

ITERATING OVER a LIST

- Compute the **sum of elements** of a list
- Common pattern

```
total = 0
for i in range(len(L)):
    total += L[i]
print(total)
```

```
total = 0
for i in L:
    total += i
print(total)
```

Like strings, can iterate over elements of list **directly**

This version is more "pythonic"!

- Notice
  - list elements are indexed `0` to `len(L)-1` and `range(n)` goes from `0` to `n-1`

ITERATING OVER a LIST

- Natural to capture iteration over a list inside a function

```
total = 0
for i in L:
    total += i
print(total)
```

```
def list_sum(L):
    total = 0
    for i in L:
        # i is 8 then 3 then 5
        total += i
    return total
```

- Function call `list_sum([8,3,5])`
  - **Loop variable `i` takes on values in the list in order!** 8 then 3 then 5
  - To help you write code and debug, comment on what the loop var values are so you don't get confused!

LISTS SUPPORT ITERATION

- Because lists are ordered sequences of elements, they naturally interface with iterative functions

Add the ***elements*** of a list

```
def list_sum(L):
    total = 0
    for e in L:
        total += e
    return(total)
```

e is: 1 then 3 then 5

```
list_sum([1,3,5]) → 9
```

Add the ***length of elements*** of a list

```
def len_sum(L):
    total = 0
    for s in L:
        total += len(s)
    return(total)
```

s is: 'ab' then 'def' then 'g'

2 then 3 then 1

```
len_sum(['ab', 'def', 'g']) → 6
```

DICTIONARIES

HOW TO STORE STUDENT INFO

- Suppose we want to store and use grade information for a set of students
- Could store using separate lists for each kind of information

```
names =   ['Ana', 'John', 'Matt', 'Katy']
grades = ['A+' ,  'B'   ,   'A' ,   'A' ]
microquizzes = ...
psets = ...
```

- Info stored across lists at **same index**, each index refers to information for a different person
- Indirectly access information by finding location in lists corresponding to a person, then extract

HOW TO ACCESS STUDENT INFO

```
def get_grade(student, name_list, grade_list):
    i = name_list.index(student)
    grade = grade_list[i]
    return (student, grade)
```

find location in list for person

Use location index to access other info

- **Messy** if have a lot of different info of which to keep track, e.g., a separate list for microquiz scores, for pset scores, etc.
- Must maintain **many lists** and pass them as arguments
- Must **always index** using integers
- Must remember to change multiple lists, when adding or updating information

HOW TO STORE AND ACCESS STUDENT INFO

- Alternative might be to use a list of lists

```
eric = ['eric', ['ps', [8, 4, 5]], ['mq', [6, 7]]]
ana = ['ana', ['ps', [10, 10, 10]], ['mq', [9, 10]]]
john = ['john', ['ps', [7, 6, 5]], ['mq', [8, 5]]]

grades = [eric, ana, john]
```

- Then could access by searching lists, but code is still messy

```
def get_grades(who, what, data):
    for stud in data:
        if stud[0] == who:
            for info in stud[1:]:
                if info[0] == what:
                    return who, info

print(get_grades('eric', 'mq', grades))
print(get_grades('ana', 'ps', grades))
```

But idea of associating data with names is worth exploring

A BETTER AND CLEANER WAY – A DICTIONARY

- Nice to use **one data structure**, no separate lists
- Nice to **index item of interest directly**
- A Python **dictionary has entries** that map a key:value

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
…	…

index element

A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
…	…

custom index element

BIG IDEA

Dict value refers to the value associated with a key.

This terminology is may sometimes be confused with the regular value of some variable.

A PYTHON DICTIONARY

- Store **pairs of data** as an **entry**
  - key (any immutable object)
    - str, int, float, bool, tuple, etc
  - value (any data object)
    - Any above plus lists and other dicts!

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'

custom index by label

element

empty dictionary

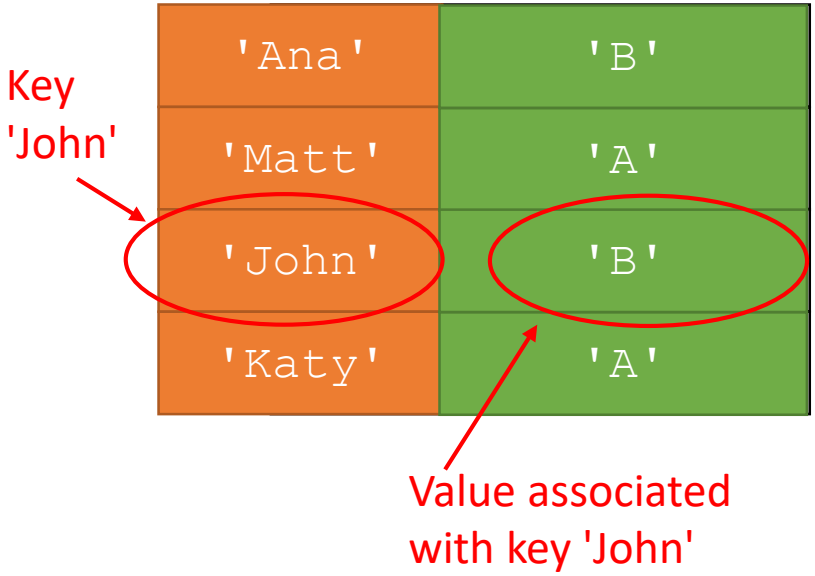
colon maps key:value

```
my_dict = {}
d = {4:16}
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

DICTIONARY LOOKUP

- Similar to indexing into a list
- **Looks up** the **key**
- **Returns** the **value** associated with the key
  - If key isn't found, get an error
- There is **no simple expression to get a key back given some value**!




```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

```
grades['John']
```
→ evaluates to `'B'`

```
grades['Grace']
```
→ gives a `KeyError`

YOU TRY IT!

- Write a function according to this spec

```
def find_grades(grades, students):
    """ grades is a dict mapping student names (str) to grades (str)
        students is a list of student names
    Returns a list containing the grades for students (in same order) """

# for example

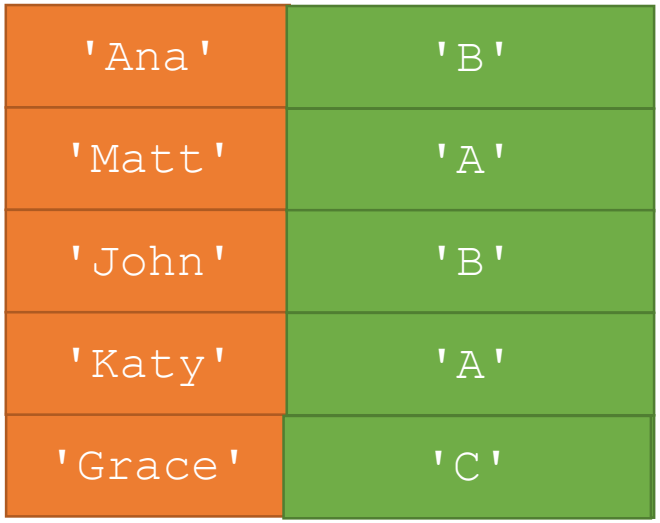
d = {'Ana':'B', 'Matt':'C', 'John':'B', 'Katy':'A'}
print(find_grades(d, ['Matt', 'Katy'])) # returns ['C', 'A']
```

BIG IDEA

Getting a dict value is just a matter of indexing with a key.

No. Need. To. Loop

DICTIONARY OPERATIONS




```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

- **Add** an entry

  ```
  grades['Grace'] = 'A'
  ```

- **Change** entry

  ```
  grades['Grace'] = 'C'
  ```

- **Delete** entry

  ```
  del(grades['Ana'])
  ```

An assignment statement, but to a location in a dictionary – different from a list

Note that the dictionary is being mutated!

DICTIONARY OPERATIONS

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'

```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

- **Test** if key in dictionary

```
'John' in grades      → returns True
'Daniel' in grades    → returns False
'B' in grades         → returns False
```

The in keyword only checks keys, not values

YOU TRY IT!

- Write a function according to these specs

```
def find_in_L(Ld, k):
    """ Ld is a list of dicts
        k is an int
    Returns True if k is a key in any dicts of Ld and False otherwise """

# for example
d1 = {1:2, 3:4, 5:6}
d2 = {2:4, 4:6}
d3 = {1:1, 3:9, 4:16, 5:25}

print(find_in_L([d1, d2, d3], 2)  # returns True
print(find_in_L([d1, d2, d3], 25)  # returns False
```

DICTIONARY OPERATIONS

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'

- Can iterate over dictionaries but assume there is no guaranteed order

```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

- Get an **iterable that acts like a tuple of all keys**

`grades.keys()` → returns `dict_keys(['Ana', 'Matt', 'John', 'Katy'])`

`list(grades.keys())` → returns `['Ana', 'Matt', 'John', 'Katy']`

- Get an **iterable that acts like a tuple of all dict values**

`grades.values()` → returns `dict_values(['B', 'A', 'B', 'A'])`

`list(grades.values())` → returns `['B', 'A', 'B', 'A']`

DICTIONARY OPERATIONS
most useful way to iterate over dict entries (both keys and vals!)

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'

- Can iterate over dictionaries but assume there is no guaranteed order

```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

- Get an **iterable that acts like a tuple of all items**

`grades.items()`

→ returns `dict_items([('Ana', 'B'), ('Matt', 'A'), ('John', 'B'), ('Katy', 'A')])`

```
list(grades.items())
```

→ returns `[('Ana', 'B'), ('Matt', 'A'), ('John', 'B'), ('Katy', 'A')]`

- Typical use is to **iterate over key,value tuple**

```
for k,v in grades.items():
    print(f"key {k} has value {v}")
```

key Ana has value B
key Matt has value A
key John has value B
key Katy has value A

YOU TRY IT!

- Write a function that meets this spec

```
def count_matches(d):
    """ d is a dict
    Returns how many entries in d have the key equal to its value """


# for example
d = {1:2, 3:4, 5:6}
print(count_matches(d))   # prints 0
d = {1:2, 'a':'a', 5:5}
print(count_matches(d))   # prints 2
```

DICTIONARY KEYS & VALUES

- Dictionaries are **mutable** objects (aliasing/cloning rules apply)
  - Use = sign to make an alias
  - Use d.copy() to make a copy
- **Assume there is no order** to keys or values!
- Dict values
  - Any type (**immutable and mutable**)
    - Dictionary values can be lists, even other dictionaries!
  - Can be **duplicates**
- Keys
  - Must be **unique**
  - **Immutable** type (`int`, `float`, `string`, `tuple`, `bool`)
    - Actually need an object that is **hashable,** but think of as immutable as all immutable types are hashable
  - Be careful using `float` type as a key

WHY IMMUTABLE/HASHABLE KEYS?

- A dictionary is stored in memory in a special way
- Next slides show an example

- Step 1: A **function is run on the dict key**
  - The function **maps any object to an int**
    E.g. map “a” to 1, “b” to 2, etc, so “ab” could map to 3
  - The int corresponds to a position in a block of memory addresses
- Step 2: At that memory address, **store the dict value**
- To do a **lookup** using a key, **run the same function**
  - If the object is immutable/hashable then you get the same int back
  - If the object is changed then the function gives back a different int!

Hash function:

1) Sum the letters
2) Take mod 16 (to fit in a memory block with 16 entries)

1 + 14 + 1 = 16
16%16 = 0

A n a	C

5 + 18 + 9 + 3 = 35
35%16 = 3

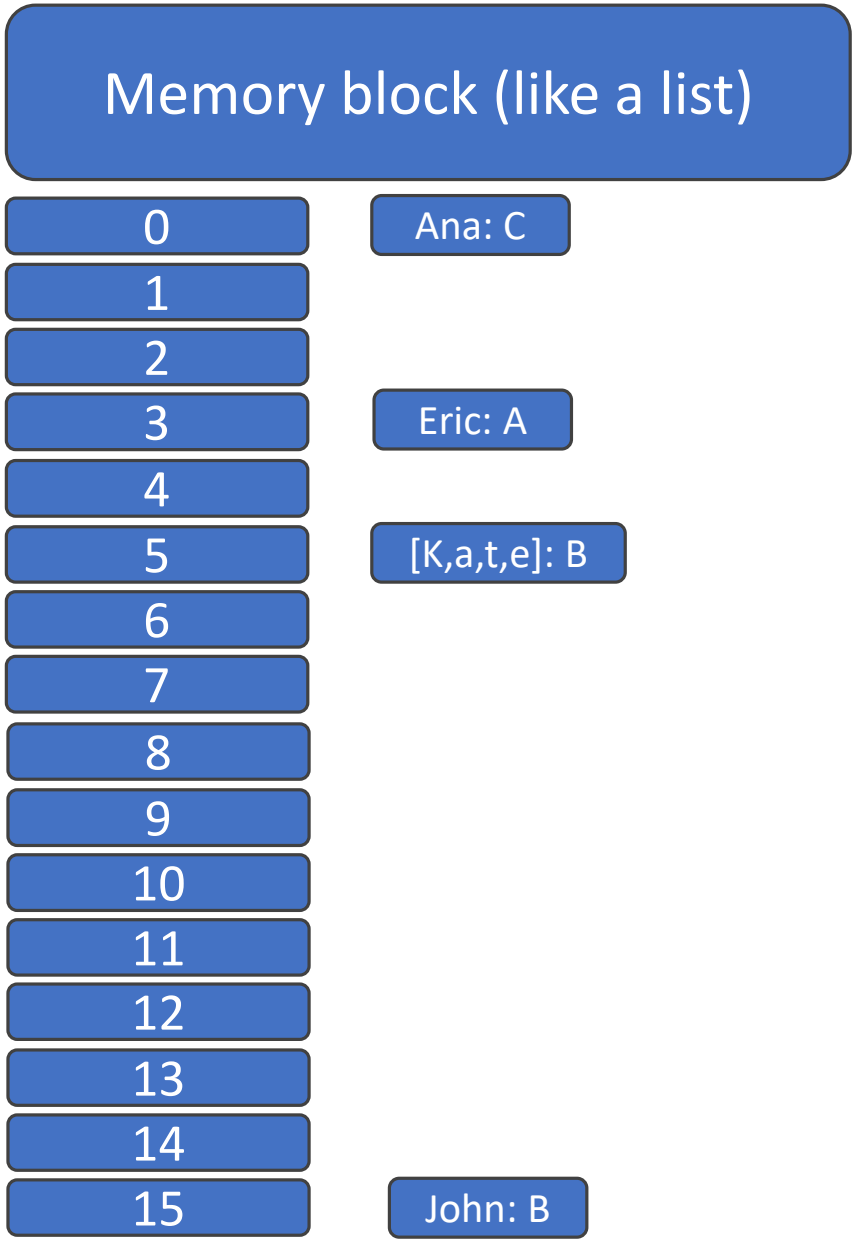
E r i c	A

10 + 15 + 8 + 14 = 47
47%16 = 15

J o h n	B

11 + 1 + 20 + 5 = 37
37%16 = 5

[K, a, t, e]	B



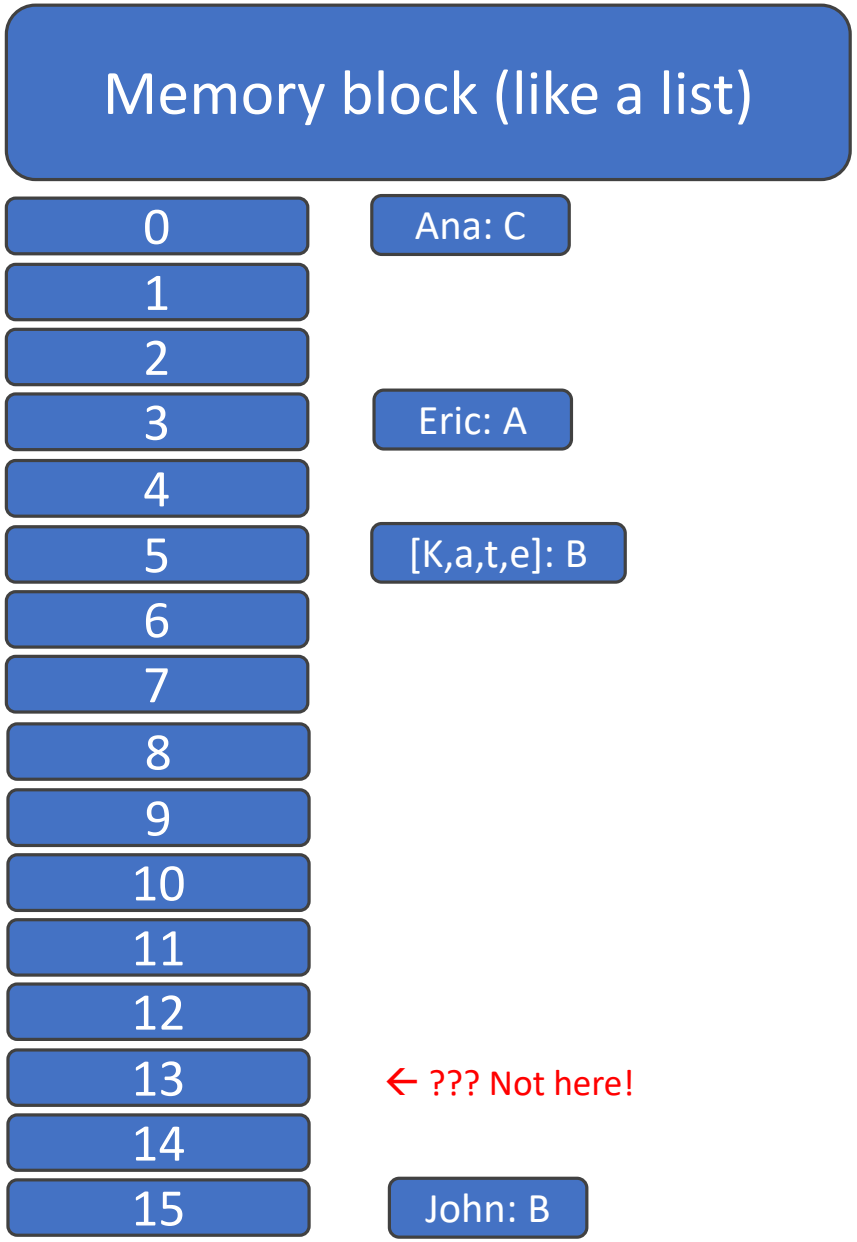
Hash function:

1) Sum the letters
2) Take mod 16 (to fit in a memory block with 16 entries)

Kate changes her name to Cate. Same person, different name. Look up her grade?

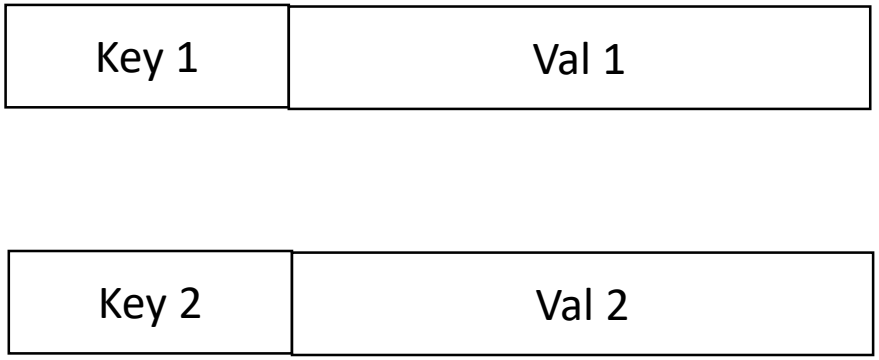
3 + 1 + 20 + 5 = 29
29%16 = 13

[C, a, t, e]



A PYTHON DICTIONARY for STUDENT GRADES

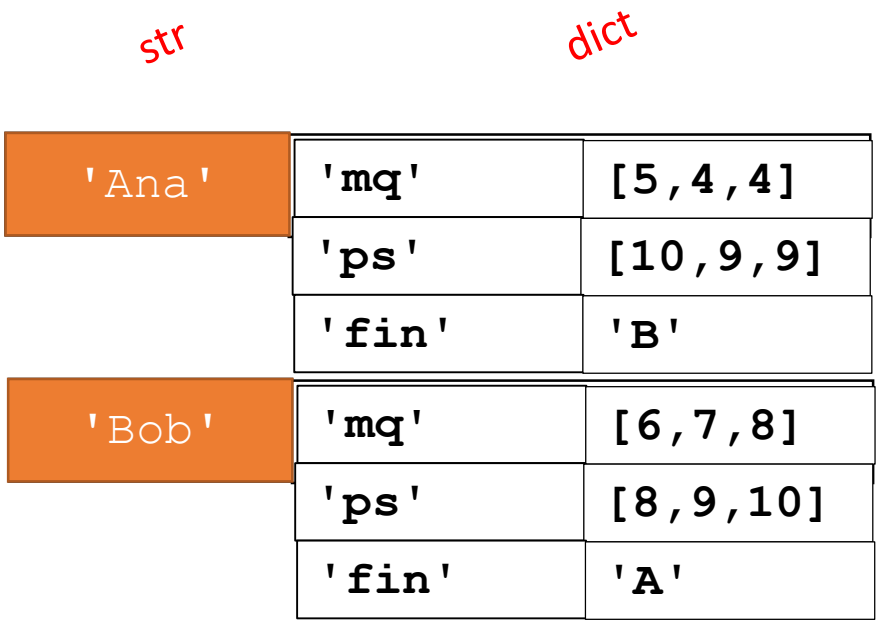
- Separate students are separate dict entries
- Entries are separated using a comma




```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

A PYTHON DICTIONARY for STUDENT GRADES

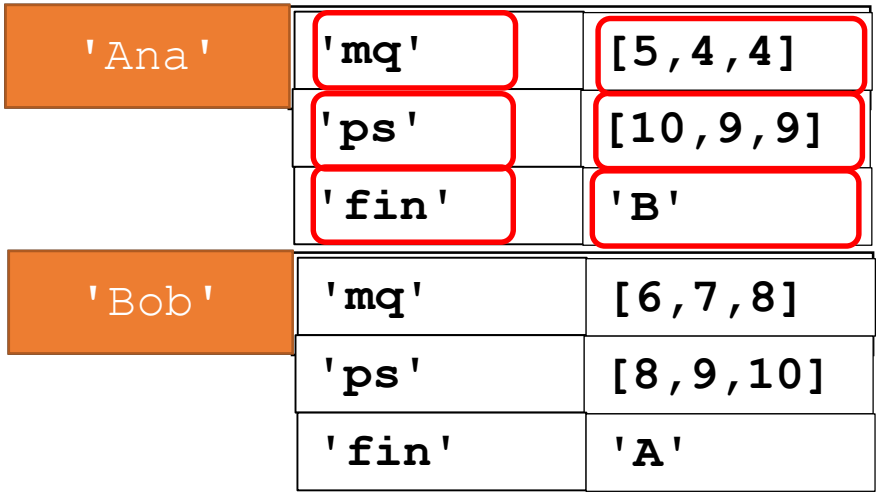
- Each dict entry maps a key to a value
- The mapping is done with a : character
- grades maps str:dict




```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

A PYTHON DICTIONARY for STUDENT GRADES

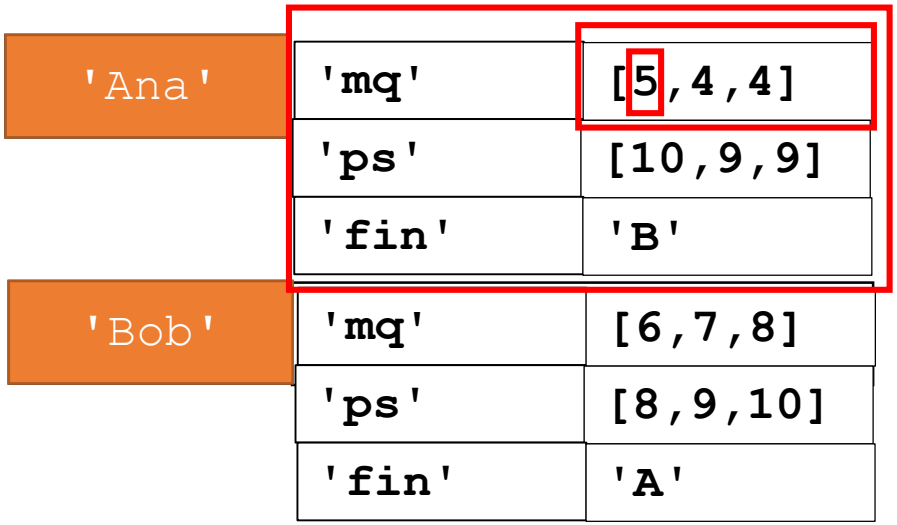
- The values of grades are dicts
- Each value maps a
  - str:list
  - str:str




```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

A PYTHON DICTIONARY for STUDENT GRADES

- The values of grades are dicts
- Each value maps a
  - str:list
  - str:str




```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

`grades['Ana']['mq'][0]` returns `5`

YOU TRY IT!

```
my_d ={'Ana':{'mq':[10], 'ps':[10,10]},
       'Bob':{'ps':[7,8], 'mq':[8]},
       'Eric':{'mq':[3], 'ps':[0]}      }

def get_average(data, what):
    all_data = []
    for stud in data.keys():
         INSERT LINE HERE
    return sum(all_data)/len(all_data)
```

Given the dict `my_d`, and the outline of a function to compute an average, which line should be inserted where indicated so that `get_average(my_d, 'mq')` computes average for all 'mq' entries? i.e. find average of all mq scores for all students.

A) `all_data = all_data + data[stud][what]`
B) `all_data.append(data[stud][what])`
C) `all_data = all_data + data[stud[what]]`
D) `all_data.append(data[stud[what]])`

`list` vs `dict`

- **Ordered** sequence of elements
- Look up elements by an integer index
- Indices have an **order**
- Index is an **integer**
- Value can be any type

- **Matches** “keys” to “values”
- Look up one item by another item
- **No order** is guaranteed
- Key can be any **immutable** type
- Value can be any type

EXAMPLE: FIND MOST COMMON WORDS IN A SONG'S LYRICS

1) Create a **frequency dictionary** mapping `str:int`
2) Find **word that occurs most often** and how many times
   - Use a list, in case more than one word with same number
   - Return a tuple `(list,int)` for (words_list, highest_freq)
3) Find the **words that occur at least X times**
   - Let user choose "at least X times", so allow as parameter
   - Return a list of tuples, each tuple is a `(list, int)` containing the list of words ordered by their frequency
   - IDEA: From song dictionary, find most frequent word. Delete most common word. Repeat. It works because you are mutating the song dictionary.

CREATING A DICTIONARY

Python Tutor LINK

```
song = "RAH RAH AH AH AH ROM MAH RO MAH MAH"
def generate_word_dict(song):
    song_words = song.lower()
    words_list = song_words.split()
    word_dict = {}
    for w in words_list:
        if w in word_dict:
            word_dict[w] += 1
        else:
            word_dict[w] = 1
    return word_dict
```

Convert all chars to lower case

Convert string to list of words; divides based on spaces

Can iterate over list of words in song

If word in dict (as a key), increase # times you've seen it, update entry

If word not in dict, first time seeing word, create entry

Return is a dict mapping str:int

USING THE DICTIONARY

Python Tutor LINK

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}


def find_frequent_word(word_dict):
    words = []
    highest = max(word_dict.values())
    for k,v in word_dict.items():
        if v == highest:
            words.append(k)
    return (words, highest)
```

Highest frequency in dict's values

Loop to see which word has the highest freq

Append to list of all words that have that highest freq

Return is a tuple of (['ah', 'mah'], 3)

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Repeat the next few steps as long as the highest frequency is greater than x
- Find highest frequency

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Use function `find_frequent_word` to get words with the biggest frequency

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Remove the entries corresponding to these words from dictionary by mutation

```
word_dict = {'rah':2,          'rom':1,          'ro':1}
```

- Save them in the result

```
freq_list = [(['ah','mah'],3)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Find highest frequency in the mutated dict

```
word_dict = {'rah':2,          'rom':1,          'ro':1}
```

- The result so far…

```
freq_list = [(['ah','mah'],3)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Use function `find_frequent_word` to get words with that frequency

```
word_dict = {'rah':2,          'rom':1,          'ro':1}
```

- The result so far…

```
freq_list = [(['ah','mah'],3)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Remove the entries corresponding to these words from dictionary by mutation

```
word_dict = {                    'rom':1,            'ro':1}
```

- Add them to the result so far

```
freq_list = [(['ah','mah'],3), (['rah'],2)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- The highest frequency is now smaller than x=2, so stop

```
word_dict = {                    'rom':1,          'ro':1}
```

- The final result

```
freq_list = [(['ah','mah'],3), (['rah'],2)]
```

LEVERAGING DICT PROPERTIES

[Python Tutor LINK]

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}

def occurs_often(word_dict, x):
    freq_list = []
    word_freq_tuple = find_frequent_word(word_dict)

    while word_freq_tuple[1] > x:

        word_freq_tuple = find_frequent_word(word_dict)
        freq_list.append(word_freq_tuple)
        for word in word_freq_tuple[0]:
            del(word_dict[word])
    return freq_list
```

Gives us a word tuple
Like ([‘ah’, ‘mah’], 3)

Stay in loop while we still have frequencies higher than x

Add those words to result

Mutate dict to remove ALL those words; on next loop, will find next most common words

SOME OBSERVATIONS

- Conversion of **string into list** of words enables use of list methods
  - Used `words_list = song_words.split()`
- **Iteration over list** naturally follows from structure of lists
  - Used `for w in words_list:`
- Dictionary stored the **same data in a more appropriate way**
- Ability to **access all values and all keys** of dictionary allows natural looping methods
  - Used `for k,v in word_dict.items():`
- **Mutability of dictionary** enables iterative processing
  - Used `del(word_dict[word])`
- **Reused functions** we already wrote!

SUMMARY

- Dictionaries have entries that **map a key to a value**
- **Keys are immutable/hashable and unique** objects
- **Values** can be **any object**
- Dictionaries can make code efficient
  - Implementation-wise
  - Runtime-wise