ITERATION

(download slides and .py files to follow along)

6.100L Lecture 3

Ana Bell

LAST LECTURE RECAP

- Strings provide a new data type
  - They are **sequences of characters**, the **first one at index 0**
  - They can be indexed and sliced
- Input
  - Done with the **`input`** command
  - Anything the user inputs is **read as a string object**!
- Output
  - Is done with the **`print`** command
  - Only objects that are printed in a .py code file will be **visible in the shell**
- Branching
  - Programs execute **code blocks** when conditions are true
  - In an `if-elif-elif`… structure, the **first condition that is True** will be executed
  - **Indentation matters** in Python!

BRANCHING RECAP

```
   if <condition>:
         < code >
         < code >
         ...
```

```
   if <condition>:
         < code >
         < code >
         ...
   else:
         < code >
         < code >
         ...
```

```
   if <condition>:
         < code >
         < code >
         ...
     elif <condition>:
         < code >
         < code >
         ...
     elif <condition>:
         < code >
         < code >
         ...
```

```
   if <condition>:
         < code >
         < code >
         ...
     elif <condition>:
         < code >
         < code >
         ...
     else:
         < code >
         < code >
         ...
```

- `<condition>` has a value `True` or `False`
- Evaluate the **first block** whose corresponding `<condition>` is `True`
  - A block is started by an `if` statement
- **Indentation matters** in Python!

- If you keep going right, you are stuck in the same spot forever
- To exit, take a chance and go the opposite way

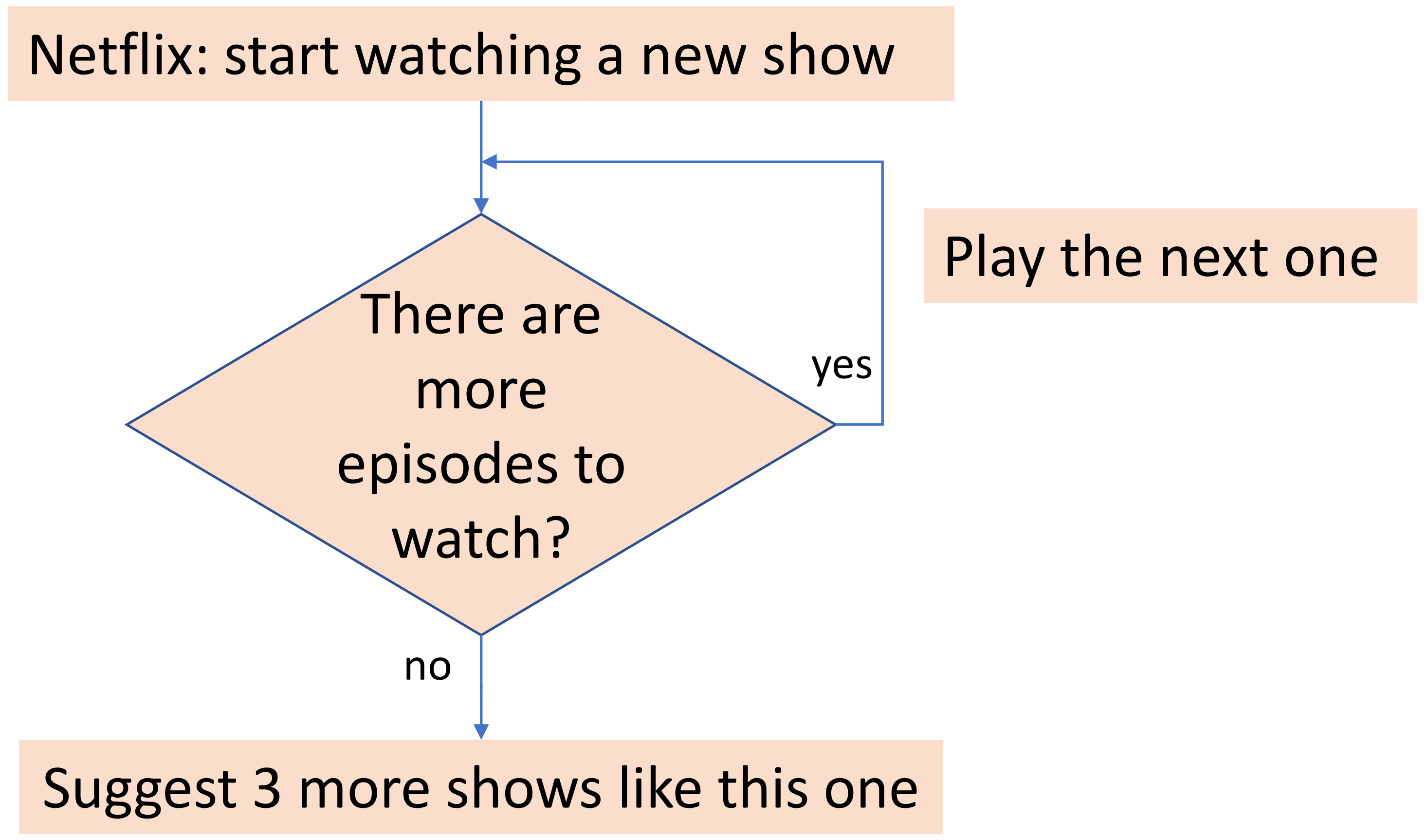
```
if <exit right>:
    <set background to woods_background>
    if <exit right>:
        <set background to woods_background>
        if <exit right>:
            <set background to woods_background>
            and so on and on and on...
        else:
            <set background to exit_background>
    else:
        <set background to exit_background>
else:
    <set background to exit_background>
```

- If you keep going right, you are stuck in the same spot forever
- To exit, take a chance and go the opposite way

```
while <exit_right>:
    <set background to woods_background>
    <ask user which way to go>
<set background to exit_background>
```

`while` LOOPS

BINGE ALL EPISODES OF ONE SHOW


Netflix: start watching a new show
There are more episodes to watch?
yes
Play the next one
no
Suggest 3 more shows like this one

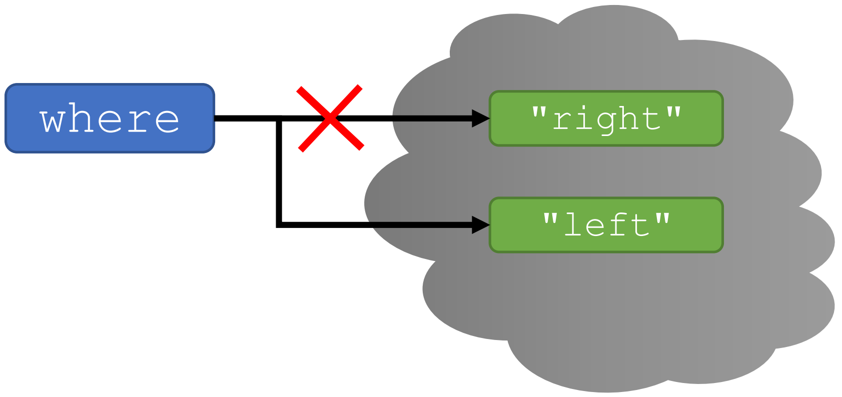
CONTROL FLOW: while LOOPS

```
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:

```python
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!")
```

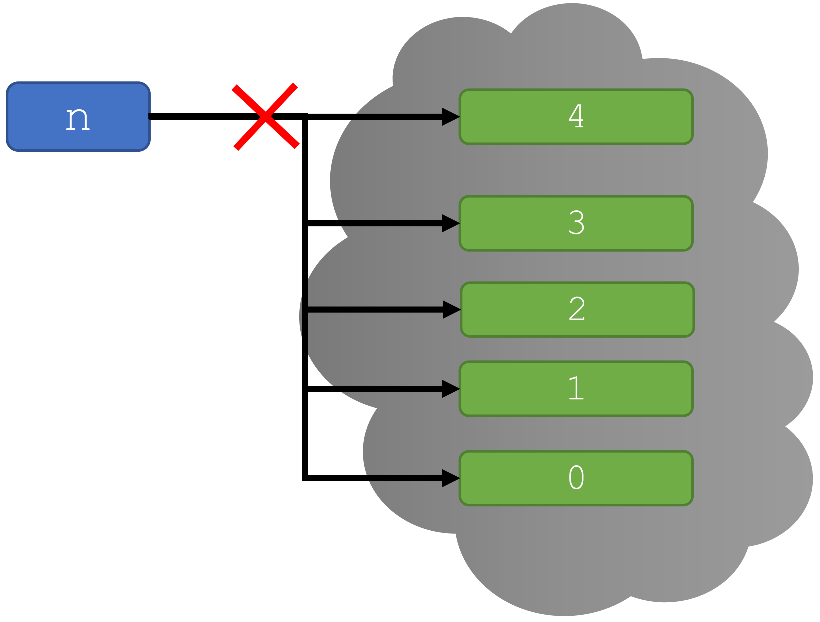
YOU TRY IT!

- 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

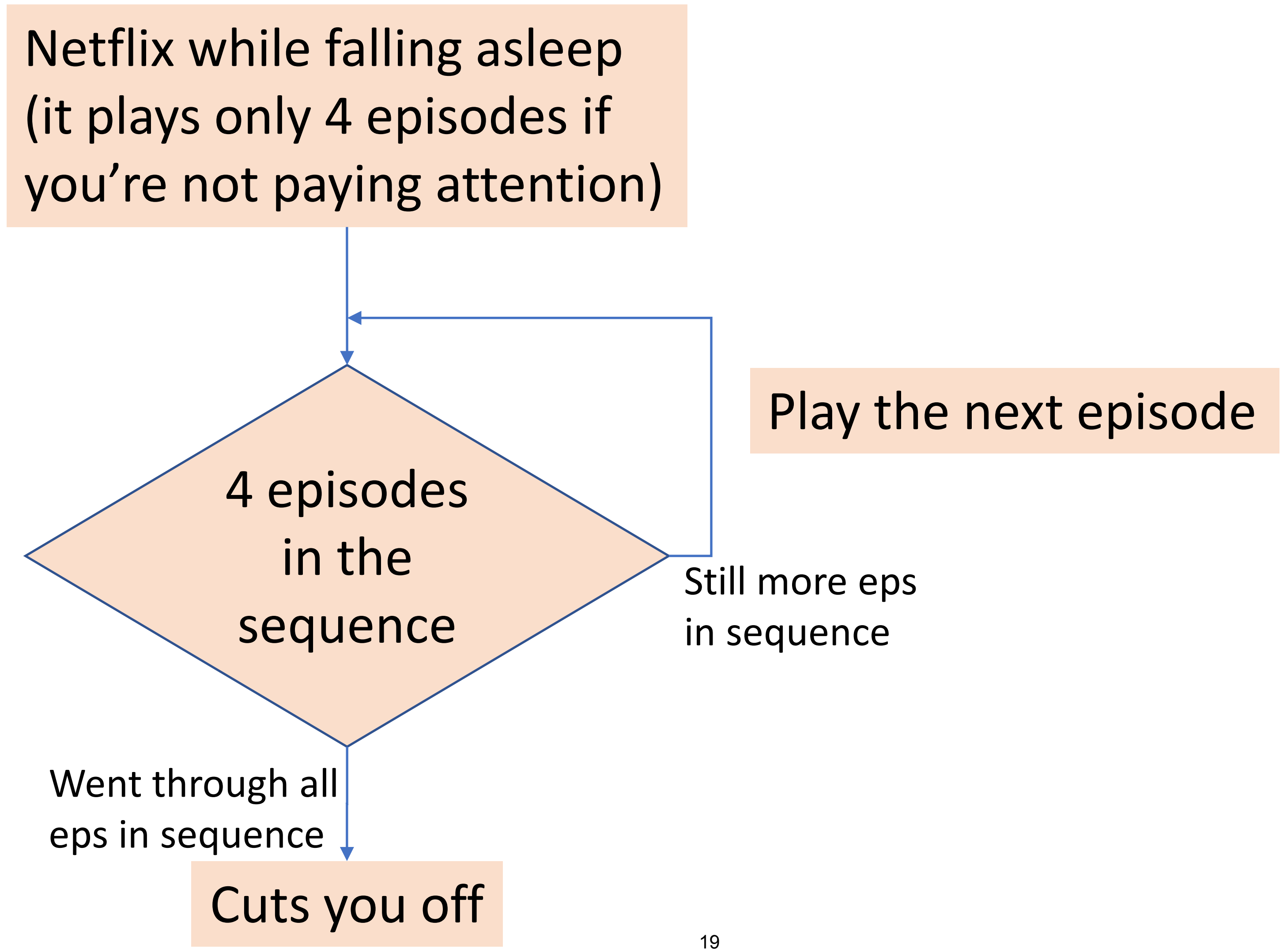
```
x = 4
i = 1
factorial = 1
while i <= x:
    factorial *= i
    i += 1
print(f'{x} factorial is {factorial}')
```

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?


Netflix while falling asleep (it plays only 4 episodes if you’re not paying attention)
4 episodes in the sequence
Play the next episode
Still more eps in sequence
Went through all eps in sequence
Cuts you off

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

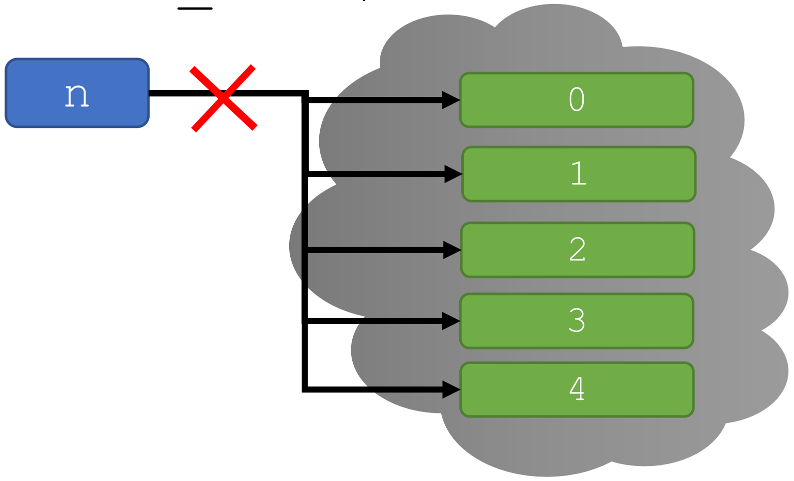
```
for n in range(5):
    print(n)
```

- **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)
  ```

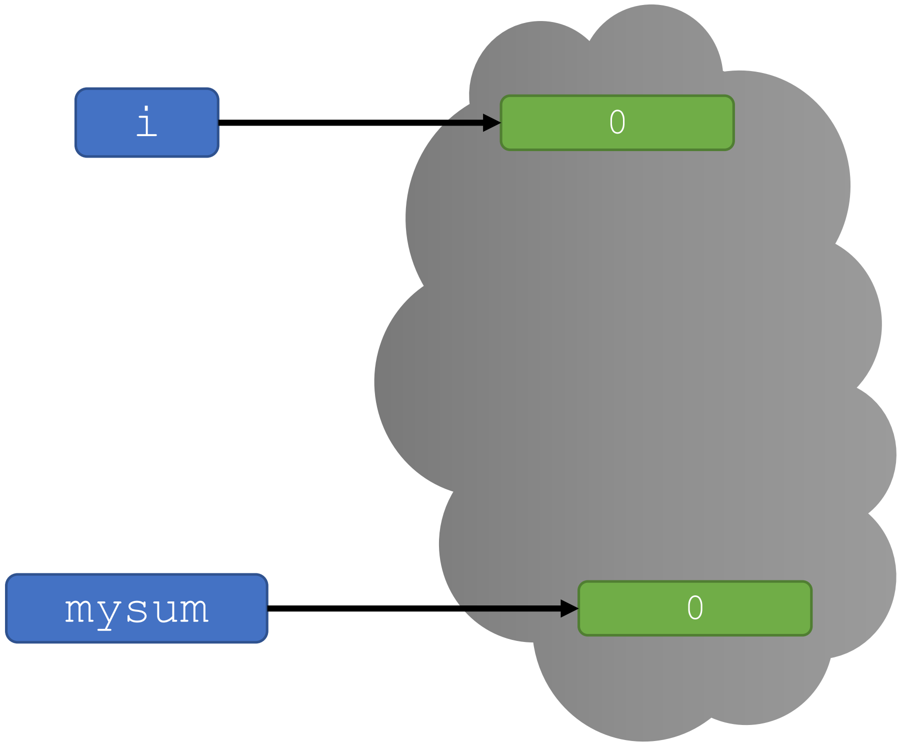
-
  ```
  for j in range(1,4,2):
      print(j*2)
  ```

-
  ```
  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

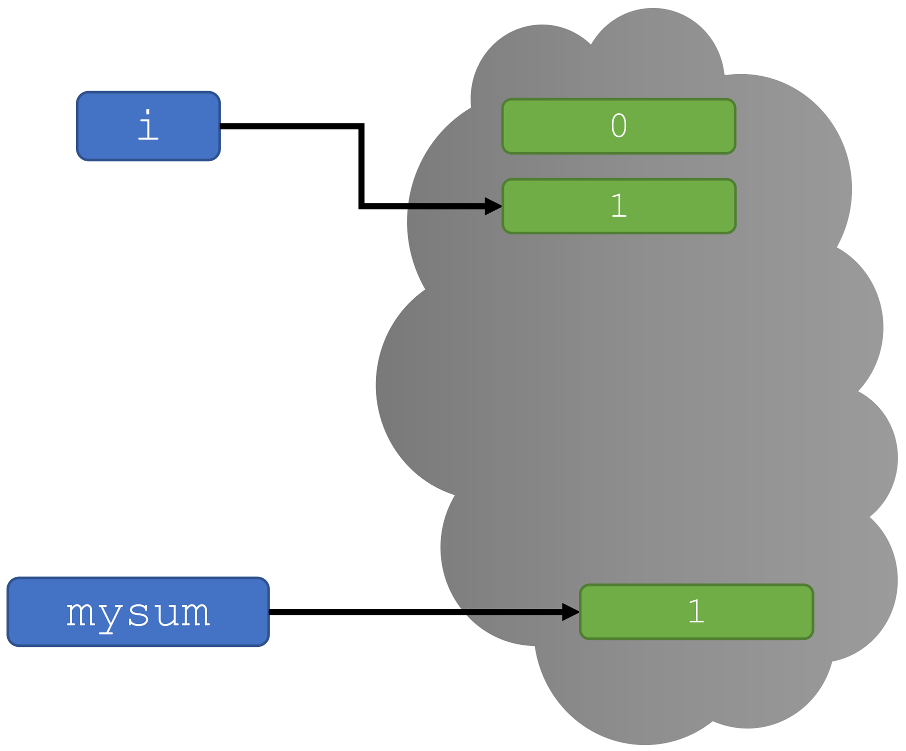
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



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)
```

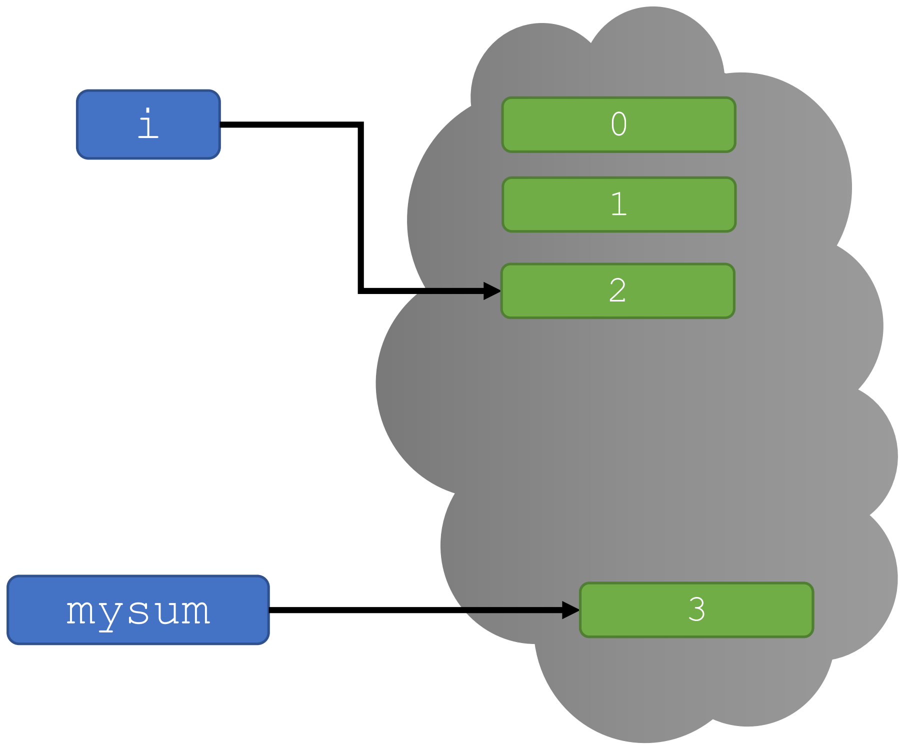


27

RUNNING SUM

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

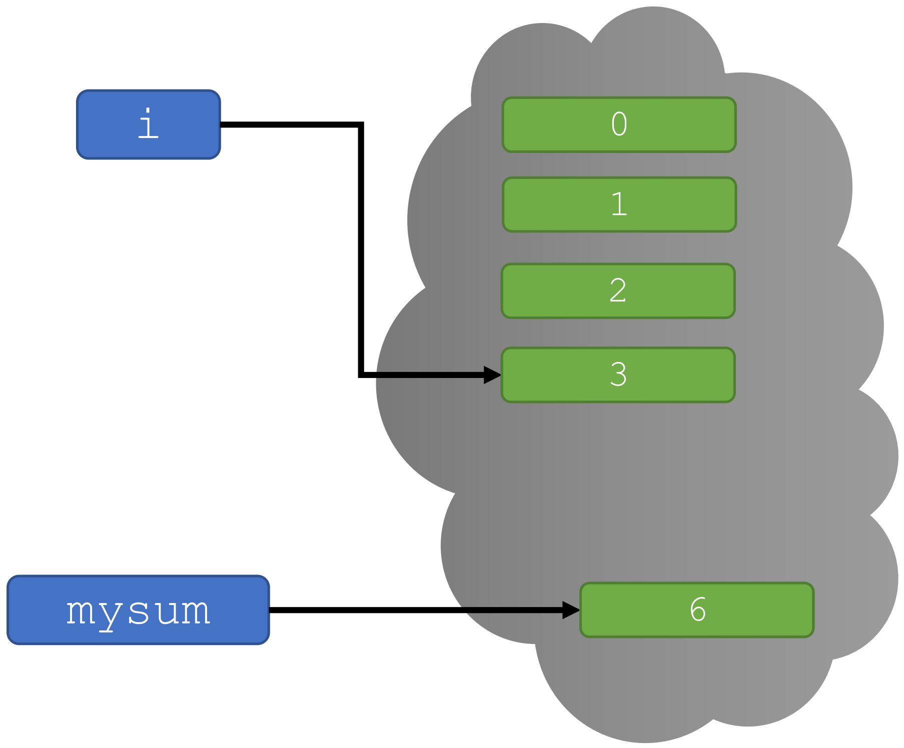
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

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

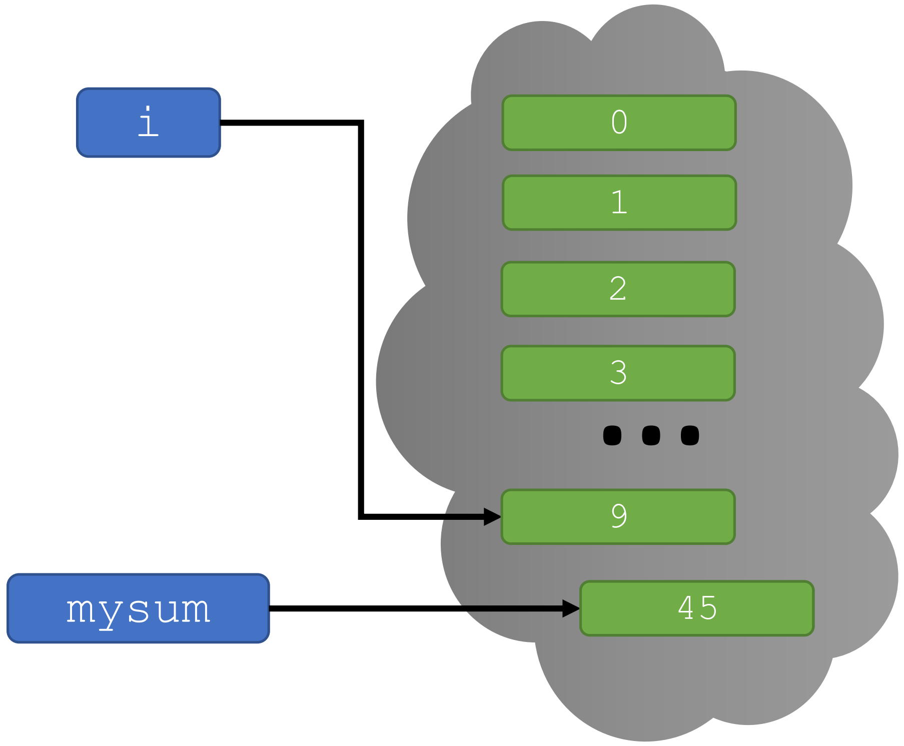
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



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.

SUMMARY

- Looping mechanisms
  - `while` and `for` loops
  - Lots of **syntax** today, be sure to get lots of **practice**!
- While loops
  - Loop as long as a **condition is true**
  - Need to make sure you don’t enter an **infinite loop**
- For loops
  - Can loop over **ranges** of numbers
  - Can loop over **elements** of a string
  - Will soon see many other things are easy to loop over