CSE 101:
Introduction to Computational Thinking

Unit 3:
Iteration, Lists and Algorithm Design

Kevin McDonnell

The Sieve of Eratosthenes

- As a motivating example for studying (i) **iteration** (code that repeats a list of steps), (ii) **lists**, and (iii) the thought process for designing algorithms, we will look at an ancient algorithm for finding prime numbers called **the Sieve of Eratosthenes**
- An integer is **prime** if it cannot be written as the product of two smaller integers
- In modern times, prime numbers play an important role in encrypting data, including Internet traffic
- Non-prime numbers are called **composite** numbers
  - Example primes: 5, 11, 73, 9967, ...
  - Example composite numbers: 10 (2×5), 99 $(3 \times 3 \times 11)$

The Sieve of Eratosthenes

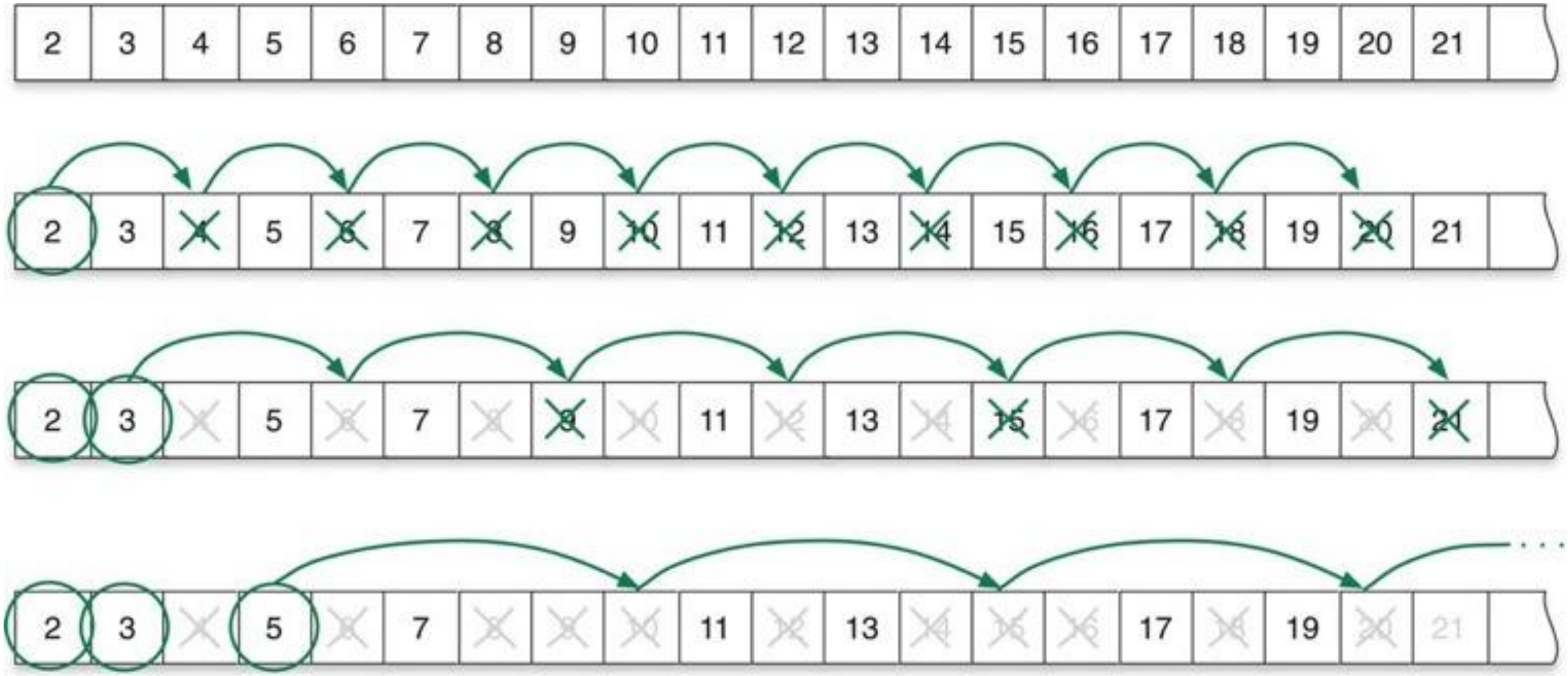
- The basic idea of the algorithm is simple. Below it is briefly described in **pseudocode**:

```
make a list of numbers, starting with 2
repeat the following steps until done:
    the first unmarked number in the list is prime
    cross off multiples of the most recent prime
```

- So, first we cross off multiples of 2.
- Then, we cross off multiples of 3 that were not crossed off in the first round (e.g., 6 is a multiple of 2 and 3, so it was crossed off in the first round).
- Next, we cross of multiples of 5 that were not crossed off in the first two rounds. Note that because 4 is a multiple of 2, all multiples of 4 were crossed off in the first round.

The Sieve of Eratosthenes

- The algorithm continues in this fashion until there are no more numbers to cross off
- We will explore the stopping condition in more detail later



Devising an Algorithm

- The method depicted in the previous slide works well for short lists
- But what if you want to find prime numbers between 2 and 100? 1000?
  - It’s a tedious process to write out a list of 100 numbers
  - It takes a lot of paper to make a list of 1000 numbers
  - Chances are you will make a few arithmetic mistakes (this is a boring job!)
- Can we turn this method into a computation?
  - Yes, but we need to be more precise about the steps

Devising an Algorithm

- A detailed specification of the starting condition is there in the pseudocode (e.g., “make a list”)
- What about the other steps? “Cross off” and “next number” need to be clearly defined if we’re going to use Python
- The stopping condition is not so clear just yet
    - When do we stop the process? Perhaps when all the numbers are crossed off?
- As you’ve probably guessed by now, we will write a program to implement the Sieve of Eratosthenes algorithm
- We will need to explore a few new ideas in Python first, however

Collections

- In everyday life we often encounter collections of things
  - Course catalog: a collection of course descriptions
  - Parking lot: a collection of vehicles
- Mathematicians also work with collections
  - Matrix (a table of numbers)
  - Sequence (e.g., 1, 1, 2, 3, 5, 8, ...)
- In computer science we make a collection by defining a **data structure** that includes references to **objects**
- The term **object** simply means *generic piece of data*
  - Objects include numbers, strings, dates, and others
- Using programming terminology, a **container** is an object that contains other objects

Lists

- The simplest kind of container in Python is called a **list**
- One way to make a list is to enclose a set of objects in square brackets:

```
ages = [61, 32, 19, 37, 42, 39]
```

- The above statement is an assignment statement, actually
- Python **allocates** space in its **object store**, which is a fancy term for the memory of the computer
- Python creates an object to represent the list and associates the name `ages` with the new object
- The `len` function tells us how many elements are in a list:

```
len(ages)  # returns the value 6
```

Lists of Strings

- Any kind of object can be stored in a list
- This statement defines a list with three strings:

```
breakfast = ['green eggs', 'ham', 'toast']
```

- Note what happens when we ask Python how many objects are in this list:

```
len(breakfast)  # returns the value 3
```

- Python did not count the individual letters
- The list contains three string objects, so the return value of the call to **len** is 3

Empty Lists

- We can also make a list with no objects:

  ```
  cars = []
  ```

- The value on the right side of that expression is a valid list
- An empty list is still a list, even though it contains no objects
  - A notebook with no pages is still a notebook, even though it contains no pages
- The length of an empty list is 0

  ```
  len(cars) # returns the value 0
  ```

- It may seem strange to create a list with nothing in it, but usually we do so because we need to wait until later to fill in the contents

Iteration

- After building a container, we often want to do something with each item
- The idea is to “step through” the container to “visit” each object
- This type of operation is called **iteration**
    - From the Latin word *iter*, for “path” or “road”
- For example, to find the largest item in an (unsorted) list, an algorithm would need to visit every item during its search
- We’ll look at this algorithm a little later

for-loops

- The simplest way to "visit" every item in a list is to use a **for-loop**
- This example prints every item in the list **`cars`** :

```
for car in cars:  # "for each car in cars"
    print(car)
```

- Note that the statements inside a for-loop – the **body** of the loop – must be indented
- Python assigns **`car`** to be the first item in the list and then executes the indented statement(s)
- Then it gets the next item, assigns it to **`car`**, and executes the indented statement(s) again
- It repeats until all the items in list have been processed

for-loops

- Suppose we had this code:

```
cars = ['Acura', 'Honda', 'Toyota', 'Ford']
for car in cars:
    print(car + ' ' + str(len(car)))
```

- The for-loop would output this:

```
Acura 5
Honda 5
Toyota 6
Ford 4
```

- Note that **`len(car)`** gives the length of each car string in the list as we “visit” that car

Example: `sum()`

- Consider a function that computes the sum of the numbers in a list
- Such a function exists in Python (it's called **`sum()`**), but let's write our own so we can understand for-loops better
- First, we'll initialize a variable **`total`** to zero
- Then, a for-loop will add each number in the list to **`total`**
  - The statement **`total += num`** means "add **`num`** to the value of **`total`**"
  - An alternative way of writing this would be **`total = total + num`**
- After all items have been added, the loop will terminate, and the function returns the final value of **`total`**

Example: `sum()`

```
def sum(nums):
    total = 0
    for num in nums:
        total += num
    return total
```

- Example:

```
t = sum([3, 5, 1])  # t will equal 9
```

- See unit03/sum_tests.py

Example: `sum()`

```
def sum(nums):
    total = 0
    for num in nums:
        total += num
    return total
```

Initialize a variable to store the running total

- Example:

```
t = sum([3, 5, 1])  # t will equal 9
```

Example: sum()

```
def sum(nums):
    total = 0
    for num in nums:
        total += num
    return total
```

Visit each number in the list of numbers

- Example:

```
t = sum([3, 5, 1])   # t will equal 9
```

Example: `sum()`

```
def sum(nums):
    total = 0
    for num in nums:
        total += num
    return total
```

Add each number to the running total

- Example:

```
t = sum([3, 5, 1])   # t will equal 9
```

Example: sum()

```
def sum(nums):
    total = 0
    for num in nums:
        total += num
    return total
```

Return the final total

- Example:

```
t = sum([3, 5, 1])  # t will equal 9
```

Example: `sum()`

- Let's *trace the execution* of this code to understand it better
- A red arrow will indicate the current line of code we are executing
- A table of values will show how the variables change value over time

Trace Execution: `sum()`

```
def sum(nums):
  → total = 0
    for num in nums:
        total += num
    return total
```

Variable	Value
total	0

- Example:

```
t = sum([3, 5, 1])  # t will equal 9
```

Trace Execution: `sum()`

```
def sum(nums):
    total = 0
    for num in nums:
        total += num
    return total
```

Variable	Value
`total`	`0`
`num`	`3`

- Example:

```
t = sum([3, 5, 1])  # t will equal 9
```

Trace Execution: `sum()`

```
def sum(nums):
    total = 0
    for num in nums:
→       total += num
    return total
```

Variable	Value
`total`	3
`num`	3

- Example:

```
t = sum([3, 5, 1])   # t will equal 9
```

Trace Execution: `sum()`

```
def sum(nums):
    total = 0
    for num in nums:
        total += num
    return total
```

Variable	Value
`total`	`3`
`num`	`5`

- Example:

```
t = sum([3, 5, 1])   # t will equal 9
```

Trace Execution: `sum()`

```
def sum(nums):
    total = 0
    for num in nums:
        total += num
    return total
```

Variable	Value
total	8
num	5

- Example:

```
t = sum([3, 5, 1])   # t will equal 9
```

Trace Execution: `sum()`

```
  def sum(nums):
      total = 0
→     for num in nums:
          total += num
      return total
```

Variable	Value
`total`	8
`num`	1

- Example:

```
t = sum([3, 5, 1])   # t will equal 9
```

Trace Execution: `sum()`

```
def sum(nums):
    total = 0
    for num in nums:
→       total += num
    return total
```

Variable	Value
`total`	`9`
`num`	`1`

- Example:

```
t = sum([3, 5, 1])  # t will equal 9
```

Trace Execution: `sum()`

```
def sum(nums):
    total = 0
    for num in nums:
        total += num
→   return total
```

Variable	Value
total	9
num	1

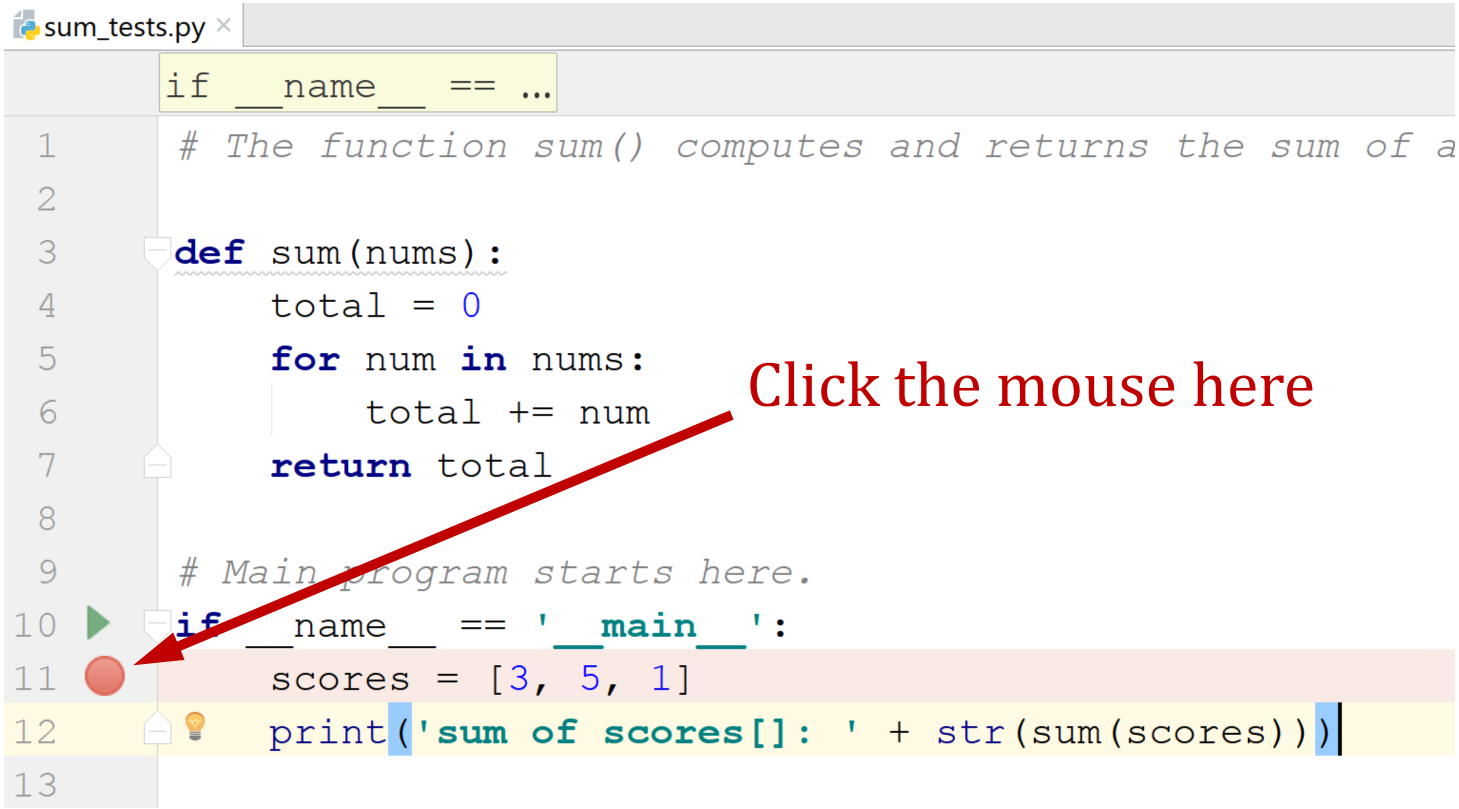
- Example:

```
t = sum([3, 5, 1])  # t will equal 9
```

Trace Execution in PyCharm

- PyCharm features a powerful tool called a **debugger** which can help you trace the execution of your program
  - Usually we use a debugger to help find bugs
- First we will set a *breakpoint* by clicking the mouse to the left of the line where we want the computer to pause execution
- In sum_tests.py let's put a breakpoint on line 11

Trace Execution in PyCharm



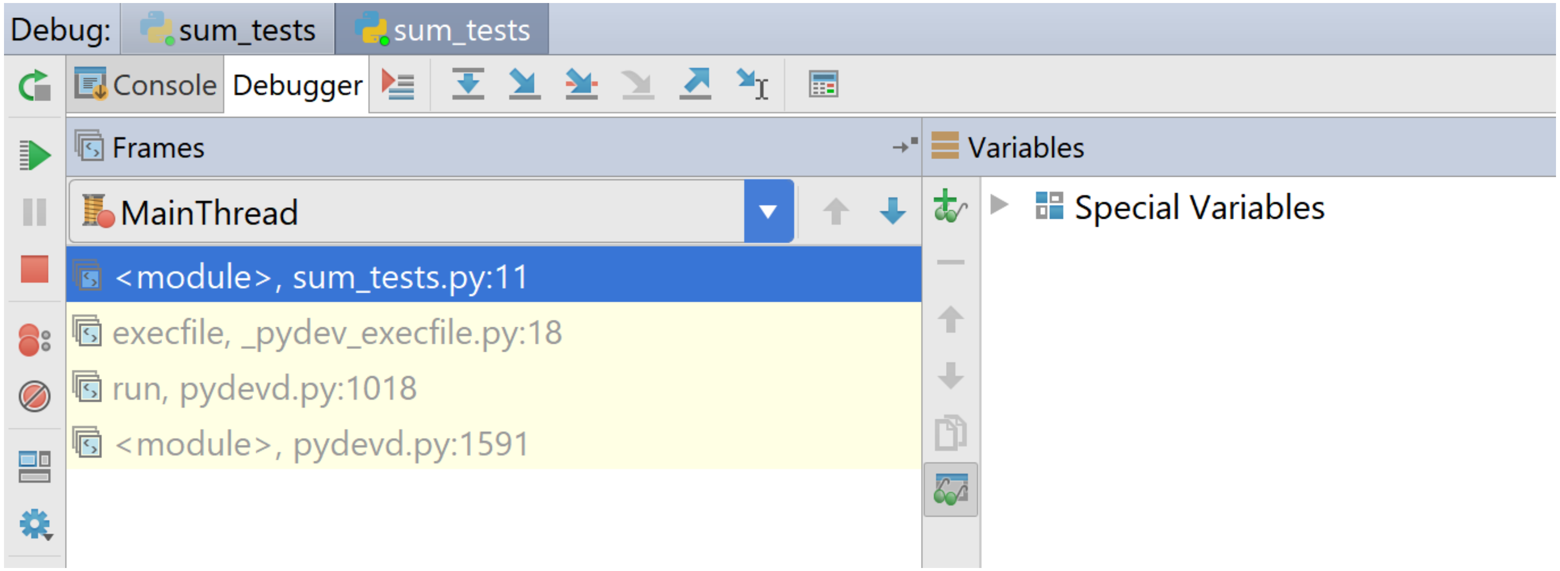
```
# The function sum() computes and returns the sum of a

def sum(nums):
    total = 0
    for num in nums:
        total += num
    return total

# Main program starts here.
if __name__ == '__main__':
    scores = [3, 5, 1]
    print('sum of scores[]: ' + str(sum(scores)))

```

Click the mouse here

- When we tell the computer to run the program, it will stop at that line and not execute it until we tell it to

Trace Execution in PyCharm

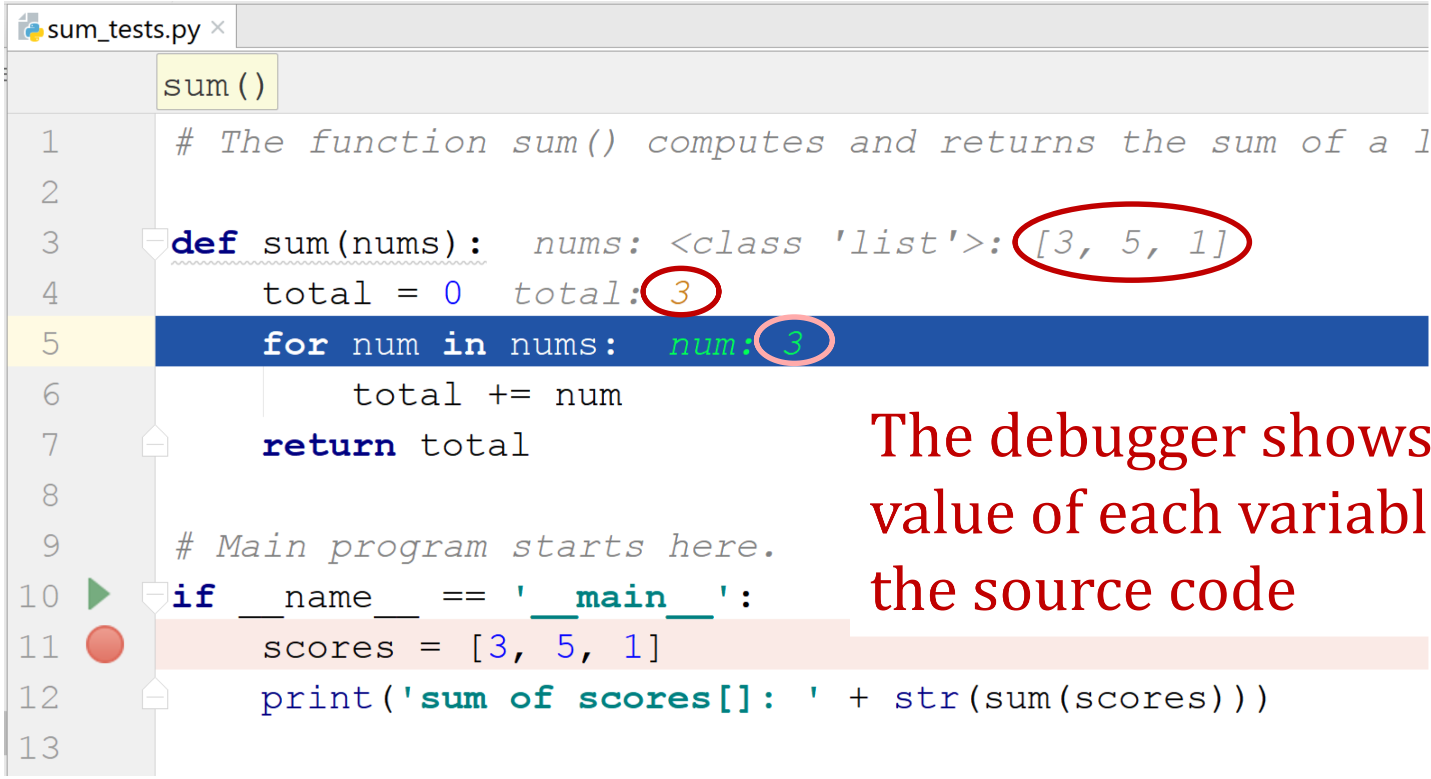
- To begin execution, right-click on sum_tests.py and pick "Debug 'sum_tests'". The computer stops at line 11.
- A "Debugger" panel opens
  - On the right we see a sub-panel named "Variables" that will show the values of variables as our program runs



<module>, sum_tests.py:11
execfile, _pydev_execfile.py:18
run, pydevd.py:1018
<module>, pydevd.py:1591

Trace Execution in PyCharm

- Every time we hit the F7 key on the keyboard the computer will execute another line of code
- PyCharm highlights in blue what line it will execute *next*

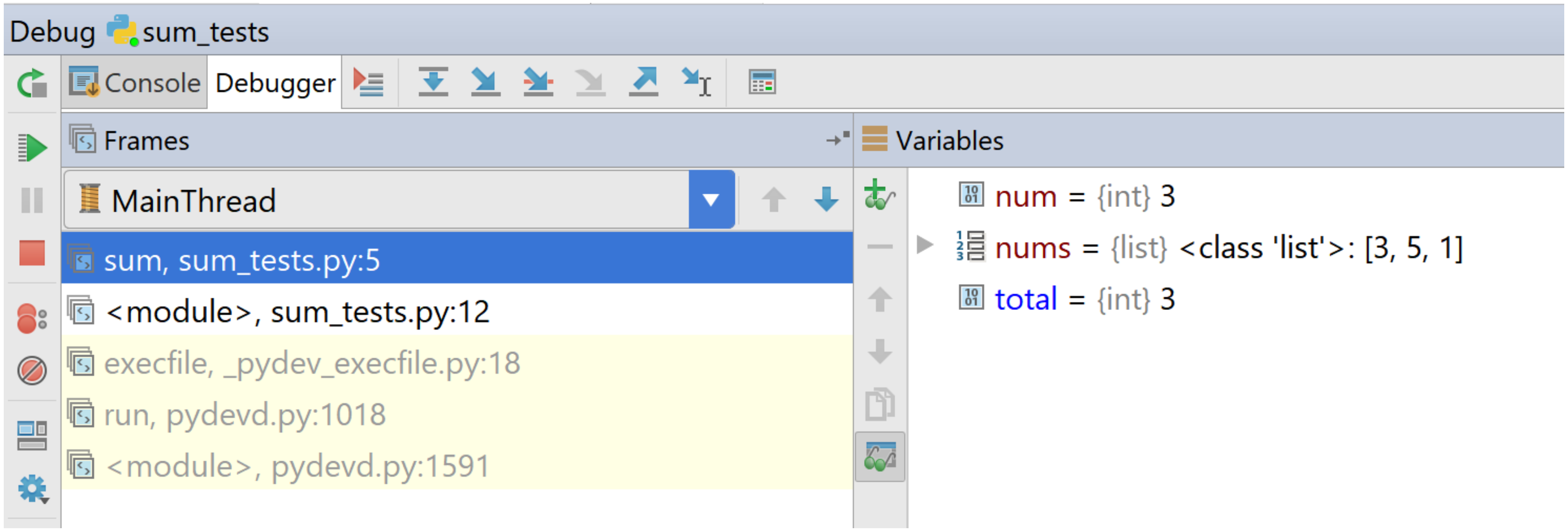


```
# The function sum() computes and returns the sum of a l

def sum(nums):   nums: <class 'list'>: [3, 5, 1]
    total = 0   total: 3
    for num in nums:   num: 3
        total += num
    return total

# Main program starts here.
if __name__ == '__main__':
    scores = [3, 5, 1]
    print('sum of scores[]: ' + str(sum(scores)))

```

The debugger shows the value of each variable in the source code

Trace Execution in PyCharm

- Here’s the state of the program after pressing F7 several times:



sum, sum_tests.py:5
<module>, sum_tests.py:12
execfile, _pydev_execfile.py:18
run, pydevd.py:1018
<module>, pydevd.py:1591

num = {int} 3
nums = {list} <class 'list'>: [3, 5, 1]
total = {int} 3

- In lab you will practice using the debugger – getting familiar with this tool will save you hours of headaches later on

List Indexes

- We often need an item in the middle of a list
- If a list has n items, the locations in the list are numbered from 0 to $n - 1$ (not 1 through n)
- The notation **`a[i]`** stands for "the item at location **`i`** in list **`a`**"
- **`a[i]`** is said aloud as "a sub i"
- In programming, we use the word **index** to refer to the numerical position of an element in a list
- Example: **`scores = [89, 78, 92, 63, 92]`**
- **`scores[0]`** is 89
- **`scores[2]`** is 92
- **`scores[5]`** gives an "index out of range" error (why?)

List Indexes

- The **`index`** method will tell us the position of an element in a list
- If the requested element is not in the list, the Python interpreter will generate an error
- Example: **`scores = [89, 78, 92, 63, 92]`**
- **`scores.index(92)`** is 2, the index of the first occurrence of 92 in the **`scores`** list
- **`scores.index(99)`** generates this error: “ValueError: 99 is not in list”

List Indexes

- If your program needs the index of a value, and you're not sure if the value is in the list, use an if-statement in conjunction with the **`in`** operator to first make sure the item is actually in the list
- Example:

```
vowels = ['a', 'e', 'i', 'o', 'u']
letter = 'e'
if letter in vowels:
    print('That letter is at index ' +
          str(vowels.index(letter)) + '.')
else:
    print('That letter is not in the list.')
```

- Output for this example: **`That letter is at index 1.`**

Iteration Using List Indexes

- A common programming "idiom" uses a for-loop based on a list index:

  ```
  for i in range(n):
     # do something with i
  ```

- `range(n)` means "the sequence of integers starting from zero and ranging up to, but not including, `n`"
- Python executes the body of the loop `n` times
- `i` is set to every value between 0 and `n-1` (i.e., `n` is not included)
- For example, the `partial_total` function on the next slide computes and returns the sum of the first `k` values in a list

Iteration Using List Indexes

- This function computes and returns the sum of the first **`k`** values in a list

```
def partial_total(nums, k):
    total = 0
    for i in range(k):
        total += nums[i]
    return total
```

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)  # returns the value 14`
- `partial_total(a, 1)  # returns the value 4`
- `partial_total(a, 6)  # error`

Iteration Using List Indexes

- This function computes and returns the sum of the first **`k`** values in a list

```
def partial_total(nums, k):
    total = 0
    for i in range(k):
        total += nums[i]
    return total
```

Initialize the variable to store the running total

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)   # returns the value 14`
- `partial_total(a, 1)   # returns the value 4`
- `partial_total(a, 6)   # error`

Iteration Using List Indexes

- This function computes and returns the sum of the first **`k`** values in a list

```
def partial_total(nums, k):
    total = 0
    for i in range(k):
        total += nums[i]
    return total
```

Generate indexes 0 through *k*-1

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)   # returns the value 14`
- `partial_total(a, 1)   # returns the value 4`
- `partial_total(a, 6)   # error`

Iteration Using List Indexes

- This function computes and returns the sum of the first **`k`** values in a list

```
def partial_total(nums, k):
    total = 0
    for i in range(k):
        total += nums[i]
    return total
```

Add each number to the running total

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)  # returns the value 14`
- `partial_total(a, 1)  # returns the value 4`
- `partial_total(a, 6)  # error`

Iteration Using List Indexes

- This function computes and returns the sum of the first **k** values in a list

```
def partial_total(nums, k):
    total = 0
    for i in range(k):
        total += nums[i]
    return total
```

Return the final total

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)   # returns the value 14`
- `partial_total(a, 1)   # returns the value 4`
- `partial_total(a, 6)   # error`

Iteration Using List Indexes

- Let’s trace the execution of this function for one example

```
def partial_total(nums, k):
 ➡  total = 0
    for i in range(k):
        total += nums[i]
    return total
```

Variable	Value
`total`	`0`

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)  # returns the value 14`

Iteration Using List Indexes

- Let’s trace the execution of this function for one example

```
def partial_total(nums, k):
    total = 0
→   for i in range(k):
        total += nums[i]
    return total
```

Variable	Value
total	0
i	0

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)  # returns the value 14`

Iteration Using List Indexes

- Let’s trace the execution of this function for one example

```
def partial_total(nums, k):
    total = 0
    for i in range(k):
        total += nums[i]
    return total
```

Variable	Value
total	4
i	0
nums[i]	4

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)  # returns the value 14`

Iteration Using List Indexes

- Let’s trace the execution of this function for one example

```
def partial_total(nums, k):
    total = 0
    for i in range(k):
        total += nums[i]
    return total
```

Variable	Value
total	4
i	1
nums[i]	4

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)  # returns the value 14`

Iteration Using List Indexes

- Let’s trace the execution of this function for one example

```
def partial_total(nums, k):
    total = 0
    for i in range(k):
→       total += nums[i]
    return total
```

Variable	Value
total	6
i	1
nums[i]	2

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)  # returns the value 14`

Iteration Using List Indexes

- Let’s trace the execution of this function for one example

```
def partial_total(nums, k):
    total = 0
  ➡ for i in range(k):
        total += nums[i]
    return total
```

Variable	Value
`total`	6
`i`	2
`nums[i]`	2

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)  # returns the value 14`

Iteration Using List Indexes

- Let’s trace the execution of this function for one example

```
def partial_total(nums, k):
    total = 0
    for i in range(k):
→       total += nums[i]
    return total
```

Variable	Value
total	14
i	2
nums[i]	8

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)  # returns the value 14`

Iteration Using List Indexes

- Let’s trace the execution of this function for one example

```
def partial_total(nums, k):
    total = 0
    for i in range(k):
        total += nums[i]
➡   return total
```

Variable	Value
`total`	14
`i`	2
`nums[i]`	8

- Example:
- `a = [4, 2, 8, 3, 1]`
- `partial_total(a, 3)  # returns the value 14`

String Indexes

- Strings and lists have much in common, including indexing
- Notation like **`name[i]`** would give us the character at index **`i`** of the string **`name`**, just as **`nums[i]`** gives us the element at index **`i`** of the list **`nums`**
- Examples:

```
title = 'Lord of the Rings'
print(title[0])  # prints L
print(title[6])  # prints f
j = 10
print(title[j])  # prints e
```

Making Lists of Numbers

- **`range`** can be used to make a list of integers
- This example makes a list of the numbers from 0 to 9:

  `nums = list(range(10))`
- Note that **`list`** is the name of a **class** in Python
  - A class describes what kinds of data an object can store
- In general, if we use a class name as a function, Python will create an object of that class
- These functions are called **constructors** because they construct new objects
- More on this topic later in the course

Back to the Sieve Algorithm

- We now have all the pieces we need to make a list of prime numbers
- We will use a Python **`list`** object to represent a “worksheet” of numbers that we will progressively “cross off”
- It will initially have all the integers from 2 to *n* (the upper limit)
- We will use for-loops to iterate over the list to cross off composite numbers
- If we pass *two* values to **`range`**, it uses one as the lower limit and the other as the upper limit (minus 1)
- For example, to make a list of numbers between 2 and 99 we would type **`list(range(2, 100))`**

The Sieve Algorithm

- The steps of the algorithm are easier to explain if we add two “placeholder” values at the front of the list to represent 0 and 1 (neither of which is a prime number)
- Python has a special value called **`None`** that stands for “no object”
- Since the expression **`a + b`** means “concatenate **`a`** and **`b`**” where **`a`** and **`b`** are lists, the statement below creates the initial worksheet:

```
worksheet = [None, None] + list(range(2,100))
```

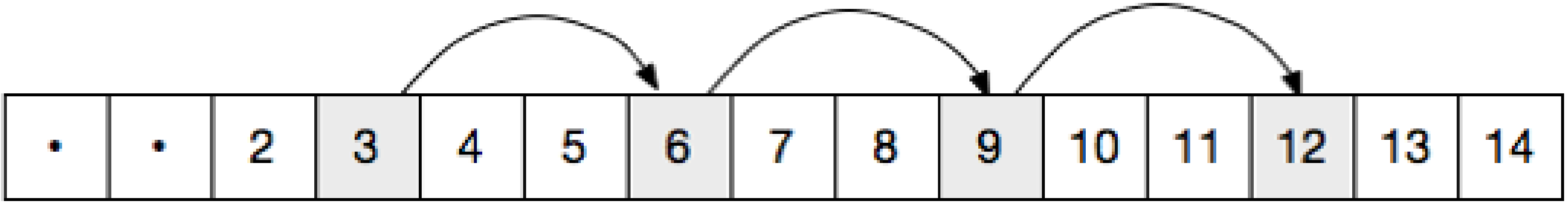
- With the two placeholders at the front, we now know any number **`i`** will be at **`worksheet[i]`**
  - For example, the number 5 will be at **`worksheet[5]`** instead of **`worksheet[3]`**

Sieve Algorithm: A Helper Function

- An important step toward implementing the Sieve algorithm is to write a function that solves a small part of the problem
- The function **`sift`** will make a single pass through the worksheet
- Pass it a number **`k`**, and **`sift`** will find and remove multiples of **`k`**
- For example, to sift out multiples of 5 from the list called **`worksheet`** we would type this: **`sift(5, worksheet)`**
- **`sift`** has a very specific purpose, and it unlikely to be used except as part of an implementation of the Sieve algorithm
- Programmers call special-purpose functions like this **helper functions**

Stepping Through the Worksheet

- On each call to **`sift`** we want to find multiples of k
- The first one is $2 \times k$
- Notice that the remaining multiples $(3 \times k, 4 \times k, \text{etc.})$ are all k steps apart:



- We can use a for-loop with a **`range`** expression to walk through the list:

```
for i in range(2*k, len(a), k):
```

- Note this **`range`** expression has three arguments: the starting point, the ending point, and the **step size** (**`k`**)

Stepping Through the Worksheet

- If we want to remove a number from the worksheet, we could use the Python **`del`** statement, which deletes an item from a list
- But this would shorten the list and make it harder to walk through on future iterations
- Our solution: replace the items with placeholders (**`None`** objects)
- The complete implementation of the **`sift`** function:

```
def sift(k, a):
    for i in range(2*k, len(a), k):
        a[i] = None
```

Stepping Through the Worksheet

```
def sift(k, a):
    for i in range(2*k, len(a), k):
        a[i] = None
```

- An example of `sift` in action:

```
worksheet = [None, None] + list(range(2, 16))
```

- `worksheet` is now:

```
[None, None, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15]
```

- Now call `sift(2, worksheet)`
- `worksheet` becomes this:

```
[None, None, 2, 3, None, 5, None, 7, None,
9, None, 11, None, 13, None, 15]
```

The `sieve()` Function

- Now that we have a helper function to do the hard work, writing the **`sieve`** function is straightforward
- When a program has helpers, a function like **`sieve`** (which is called to solve the complete problem) is known as a **top-level function**
- We have to write a loop that starts by sifting multiples of 2 and keep calling **`sift`** until all composite numbers are removed
- This loop can stop when the next number to send to **`sift`** is greater than the square root of **`n`**
- The for-loop that controls the loop should set **`k`** to every value from 2 up to the square root of **`n`**:

```
for k in range(2, sqrt(n))
```

The `sieve()` Function

```
for k in range(2, sqrt(n))
```

- There is a problem with this code: we cannot pass a floating-point value to **`range`**
- If we "round up" the square root, we'll have what we want: an integer that is greater than the highest possible prime factor of **`n`**
- A function named **`ceil`** in Python's math library does this operation
- **`ceil`** is short for "ceiling"
- A corresponding function named **`floor`** rounds a floating-point value down to the nearest integer

`sieve()`'s Main loop

- One important detail: before sifting out multiples of a number, we make sure we haven't already removed it
- For example, we don't sift multiples of 4 because 4 was already removed when sifting multiples of 2
  - **`sift`** would still work, but our program would be less efficient
- The main loop looks like this:

```
for k in range(2, int(ceil(sqrt(n)))):
   if worksheet[k] is not None:
         sift(k, worksheet)
```

- Note that the expression **`x is not None`** is the preferred way of testing to see if **`x`** is a reference to the **`None`** object

Sieve: Remove the Placeholders

- There is just one last step: to make the final list we have to remove the **`None`** objects from the worksheet
- A new helper function called **`non_nulls`** returns a copy of the worksheet, but without any **`None`** objects
- It makes an initial empty list named **`res`** (for “result”)
- Then it uses a for loop to look at every item in the input list
- If an item is not **`None`**, the item is appended to **`res`** using the **`append`** method for lists
- When the iteration is complete, **`res`** is returned as the result of the function call

Sieve: Remove the Placeholders

```
def non_nulls(a):
    res = []
    for x in a:
        if x is not None:
            res.append(x)
    return res
```

- Example:

```
worksheet = [None, None, 2, 3, None, 5,
             None, 7, None, None, None,
             11, None, 13, None, None]
worksheet = non_nulls(worksheet)
```

- **`worksheet`** is now: **`[2, 3, 5, 7, 11, 13]`**

Sieve: Remove the Placeholders

```
def non_nulls(a):
    res = []
    for x in a:
        if x is not None:
            res.append(x)
    return res
```

Initialize **res[]** to be the empty list

- Example:

```
worksheet = [None, None, 2, 3, None, 5,
             None, 7, None, None, None,
             11, None, 13, None, None]
worksheet = non_nulls(worksheet)
```

- **worksheet** is now: **[2, 3, 5, 7, 11, 13]**

Sieve: Remove the Placeholders

```
def non_nulls(a):
    res = []
    for x in a:
        if x is not None:
            res.append(x)
    return res
```

Visit each element in the list **`a[]`**

- Example:

```
worksheet = [None, None, 2, 3, None, 5,
             None, 7, None, None, None,
             11, None, 13, None, None]
worksheet = non_nulls(worksheet)
```

- **`worksheet`** is now: **`[2, 3, 5, 7, 11, 13]`**

Sieve: Remove the Placeholders

```
def non_nulls(a):
    res = []
    for x in a:
        if x is not None:
            res.append(x)
    return res
```

← See if **`x`** is actually a number

- Example:

```
worksheet = [None, None, 2, 3, None, 5,
             None, 7, None, None, None,
             11, None, 13, None, None]
worksheet = non_nulls(worksheet)
```

- **`worksheet`** is now: **`[2, 3, 5, 7, 11, 13]`**

Sieve: Remove the Placeholders

```
def non_nulls(a):
    res = []
    for x in a:
        if x is not None:
            res.append(x)
    return res
```

If **`x`** is a number, append it to **`res[]`**

- Example:

```
worksheet = [None, None, 2, 3, None, 5,
             None, 7, None, None, None,
             11, None, 13, None, None]
worksheet = non_nulls(worksheet)
```

- **`worksheet`** is now: **`[2, 3, 5, 7, 11, 13]`**

Aside: Appending to a List

- **+=** can be used to concatenate one string to the end of another
- This syntax can also be used to append one list to another
- Example:

```
fruits = ['apple', 'orange’]
fruits += ['banana', 'mango', 'pear’]
```

  - **fruits** is now: **['apple', 'orange’, 'banana', 'mango', 'pear']**

```
fruits += ['pineapple']
```

  - **fruits** is now: **['apple', 'orange’, 'banana', 'mango', 'pear’, 'pineapple']**

The Sieve Algorithm: Completed

- We can now put all the pieces together
- Import the **`math`** library to get access to **`sqrt`** and **`ceil`**
- In the body of the **`sieve`** function we need to:
    - Create the **`worksheet`** with two initial **`None`** objects and all integers from 2 to **`n`**
    - Add the for-loop that calls **`sift`**
    - Call **`non_nulls`** to remove the **`None`** objects from the **`worksheet`**
- See unit03/sieve.py and the next slide for the code

Completed `sieve()` Function

See unit03/sieve.py

```
from math import *

def sift(k, a):
    ...  # see earlier slides

def non_nulls(a):
    ...  # see earlier slides

def sieve(n):
    worksheet = [None, None] + list(range(2, n))
    for k in range(2, int(ceil(sqrt(n)))):
        if worksheet[k] is not None:
            sift(k, worksheet)
    return non_nulls(worksheet)

primes = sieve(100)
print(primes)
```

Abstraction

- Now that we have a function for making lists of prime numbers we can save it for later use
- We can use it to answer questions about primes
  - How many primes are less than *n*? What is the largest gap between successive primes? What are some twin primes (two prime numbers that differ only by 2, like 17 and 19)? Many other questions are possible.
- This is a good example of abstraction: we have a nice, neat package that we can save and reuse
- In the future, we don’t have to worry about the implementation details of **`sieve`**: we can just use it!
- We (and people who use it) just need to know that **`sieve(n)`** makes a list of prime numbers from 2 to **`n`**

Additional Examples

- We’ll now take a look at some additional examples of how to use for-loops and lists to solve problems in Python

Example: Find the Maximum

- Even though there already exists a function in Python that finds the maximum value in a list (it's called **`max`**), we will write our own algorithm for performing this task
- The basic idea is to *iterate* over the list and keep track of the largest value we have seen up to that point
- We begin by taking the value at index 0 as the maximum
- We continue with the remainder of the list, comparing the next value with the current maximum and updating the maximum if and when we find a value larger than the current maximum

Example: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum

ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum

ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Variable	Value
maximum	20

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	20
i	1

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	20
i	1
nums[i]	16

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum: False
            maximum = nums[i]
    return maximum


ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Variable	Value
maximum	20
i	1
nums[i]	16

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	20
i	2
nums[i]	22

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
→       if nums[i] > maximum:
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	20
i	2
nums[i]	22

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum: True
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	20
i	2
nums[i]	22

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
  def find_max(nums):
      maximum = nums[0]
      for i in range(1, len(nums)):
          if nums[i] > maximum:
→             maximum = nums[i]
      return maximum
```

Variable	Value
maximum	22
i	2
nums[i]	22

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	22
i	3
nums[i]	30

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	22
i	3
nums[i]	30

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
    def find_max(nums):
        maximum = nums[0]
        for i in range(1, len(nums)):
→           if nums[i] > maximum: True
                maximum = nums[i]
        return maximum
```

Variable	Value
maximum	22
i	3
nums[i]	30

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
→           maximum = nums[i]
    return maximum
```

Variable	Value
`maximum`	30
`i`	3
`nums[i]`	30

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	30
i	4
nums[i]	17

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	30
i	4
nums[i]	17

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum: False
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	30
i	4
nums[i]	17

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```python
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	30
i	5
nums[i]	24

```python
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	30
i	5
nums[i]	24

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum: False
            maximum = nums[i]
    return maximum
```

Variable	Value
maximum	30
i	5
nums[i]	24

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Trace Execution: find_max.py

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
→   return maximum
```

Variable	Value
maximum	30
i	5
nums[i]	24

```
ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages)  # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Example: Count the Vowels

- A for-loop can be used to iterate over the characters of a string
- To see how this works, let's look a function called **`count_vowels`** that counts the number of vowels (lowercase or uppercase) in a word
- To make this problem a little easier to solve, we will call the **`lower()`** method for strings, which makes a copy of a given string and changes all the uppercase letters to lowercase (**`upper()`** makes all letters uppercase)
- Strings are **immutable** (unchangeable) quantities
- If we want to convert a string into lowercase, all we can really do is make a lowercase copy of it and then replace the original string with the new one

Example: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower(): # search through a
        if letter in vowels:    # lowercase copy of
            num_vowels += 1     # the original word
    return num_vowels

word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
        if letter in vowels:
            num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))   # will print 2
```

Variable	Value

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
➡   num_vowels = 0
    for letter in word.lower():
        if letter in vowels:
            num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	0

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
➡   for letter in word.lower():
        if letter in vowels:
            num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	0
letter	c

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
        if letter in vowels:
            num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	0
letter	c

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
→       if letter in vowels: False
            num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	0
letter	c

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
        if letter in vowels:
            num_vowels += 1
    return num_vowels

word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))   # will print 2
```

Variable	Value
num_vowels	0
letter	i

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
→       if letter in vowels:
            num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))   # will print 2
```

Variable	Value
num_vowels	0
letter	i

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
→       if letter in vowels: True
            num_vowels += 1
    return num_vowels

word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	0
letter	i

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
        if letter in vowels:
            num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	1
letter	i

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
➡   for letter in word.lower():
        if letter in vowels:
            num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	1
letter	d

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
        if letter in vowels:
            num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	1
letter	d

Trace Execution: vowels.py

Variable	Value
num_vowels	1
letter	d

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
        if letter in vowels: False
            num_vowels += 1
    return num_vowels

word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word))  # will print 2
```

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
        if letter in vowels:
            num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	1
letter	e

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
→       if letter in vowels:
            num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	1
letter	e

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
        if letter in vowels: True
            num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	1
letter	e

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
        if letter in vowels:
→           num_vowels += 1
    return num_vowels


word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))   # will print 2
```

Variable	Value
num_vowels	2
letter	e

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
        if letter in vowels:
            num_vowels += 1
    return num_vowels

word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	2
letter	r

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
→       if letter in vowels:
            num_vowels += 1
    return num_vowels
```

Variable	Value
`num_vowels`	`2`
`letter`	`r`

```
word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
        if letter in vowels: False
            num_vowels += 1
    return num_vowels

word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))   # will print 2
```

Variable	Value
num_vowels	2
letter	r

Trace Execution: vowels.py

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower():
        if letter in vowels:
            num_vowels += 1
    return num_vowels

word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
      str(count_vowels(word)))  # will print 2
```

Variable	Value
num_vowels	2
letter	r

A List of Lists

- In Python, a list can contain objects of any type
- A list is an object. Therefore, a list can contain other lists!
- Imagine that we have a group of 4 students, and for each student we have 3 exam scores:

-
  ```
  scores = [[89, 85, 90], [78, 85, 72],
            [99, 86, 92], [82, 84, 79]]
  ```
- To access a particular score, we need to give two indexes: first, which student we are interested in (0 through 3)
- Second, which score of that student we are interested in (0 through 2)
- Example: **`scores[3][1]`** is student 3’s score on exam 1 (which is 84)

Example: Compute Averages (v1)

- Let’s write some code that will compute the average score the students earned on each exam
- We will write more than one version of the program, but let’s start things off simply
- In the first version we will “hard-code” several values (the number of students and the number of scores) in the program
- The we will generalize things a bit and use variables for these values

Example: averages_v1.py

```
scores = [[89, 85, 90], [78, 85, 72],
          [99, 86, 92], [82, 84, 79]]

averages = [0, 0, 0]

for student in scores:
    averages[0] += student[0]
    averages[1] += student[1]
    averages[2] += student[2]

for i in range(3):
    averages[i] /= 4

print(averages)
```

Example: averages_v1.py

```
scores = [[89, 85, 90], [78, 85, 72],
          [99, 86, 92], [82, 84, 79]]

averages = [0, 0, 0]

for student in scores:
    averages[0] += student[0]
    averages[1] += student[1]
    averages[2] += student[2]

for i in range(3):
    averages[i] /= 4

print(averages)
```

When this loop ends, **`averages[0]`** will contain the sum 89+78+99+82

Example: averages_v1.py

```
scores = [[89, 85, 90], [78, 85, 72],
          [99, 86, 92], [82, 84, 79]]

averages = [0, 0, 0]

for student in scores:
    averages[0] += student[0]
    averages[1] += student[1]
    averages[2] += student[2]

for i in range(3):
    averages[i] /= 4

print(averages)
```

When this loop ends, `averages[1]` will contain the sum 85+85+86+84

Example: averages_v1.py

```
scores = [[89, 85, 90], [78, 85, 72],
          [99, 86, 92], [82, 84, 79]]

averages = [0, 0, 0]

for student in scores:
    averages[0] += student[0]
    averages[1] += student[1]
    averages[2] += student[2]
```

When this loop ends, **averages[2]** will contain the sum 90+72+92+79

```
for i in range(3):
    averages[i] /= 4

print(averages)
```

Example: Compute Averages (v2)

- The first version of our code has a major negative: the algorithm will work only for a class of four students who took three exams
- Suppose we had a larger or smaller class? Or suppose the students took more or fewer exams?
- We'll develop a better (but more complicated) version of the algorithm that can adapt to larger/smaller class sizes and more/fewer exams
- Our approach will rely on **nested loops**, which means we will have one a loop inside of another
- Nested loops will become increasingly important as we progress through the course

Example: averages_v2.py

- One other thing before we look at the program
- Recall that syntax like **`'Hi'*3`** will create a new string by repeating a given string a desired number of times
  - For instance, **`'Hi'*3`** equals **`'HiHiHi'`**
- In a similar manner, **`[0]*3`** would create a list containing 3 zeroes, namely, **`[0, 0, 0]`**
- As we can see, the **`*`** notation with strings and lists is essentially a form of concatenation

Example: averages_v2.py

```
scores = [[89, 85, 90], [78, 85, 72], [99, 86, 92],
          [82, 84, 79]]

num_students = len(scores)
num_exams = len(scores[0])  # each student took the
averages = [0] * num_exams  # same number of exams

for student in scores:
    for i in range(0, num_exams):  # nested loops
        averages[i] += student[i]

for i in range(0, num_exams):
    averages[i] /= num_students

print(averages)
```

Example: Compute Averages (v3)

- In a third and final version of our exam average calculator, we will *encapsulate* (enclose or wrap) the computations inside of a function **`compute_averages(students)`**
- The function will take the list of scores as its argument
- After computing the exam averages, the function will return a list of the exam scores
- So we'll now see that Python functions can return many values at once (via a list), not just a single number or string

Example: averages_v3.py

```
scores = [[89, 85, 90], [78, 85, 72], [99, 86, 92],
          [82, 84, 79]]

def compute_averages(students):
    num_students = len(students)
    num_exams = len(students[0])
    avgs = [0] * num_exams

    for student in students:
        for i in range(0, num_exams):
            avgs[i] += student[i]

    for i in range(0, num_exams):
        avgs[i] /= num_students

    return avgs

averages = compute_averages(scores)
print(averages)
```

Example: Bottles of Beer/Milk

- We'll conclude the examples on a lighter note by looking at a program that prints the lyrics of the song "99 Bottles of Beer on the Wall"
- In this song, the singer needs to count from 99 down to 0
- The **`range`** command lets us count up, but it also can count down if we give a negative number for the step size
- For example, **`range(10,-1,-1)`** will count down from 10 to 0 by 1s
- So **`list(range(10,-1,-1))`** would generate the list **`[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]`**
- The code on the next slide asks the user for the starting number so that we can start from a value other than 99

Example: bottles.py

```
age = int(input('How old are you? '))

if age < 21:
    drink_type = 'milk'
else:
    drink_type = 'beer';

num_bottles = int(input('How many bottles of ' +
                        drink_type + ' do you have? '))

for bottle in range(num_bottles, -1, -1):
   if bottle > 1:
      print(str(bottle) + ' bottles of ' + drink_type +
              ' on the wall!')
   elif bottle == 1:
      print('1 bottle of ' + drink_type + ' on the wall!')
   else:
      print('No bottles of ' + drink_type + ' on the wall!')
```