Part 3: Lists

Python provides a powerful set of tools to create and manipulate lists of data. In this part, we take a deep dive into the Python list type. This includes mutability, list methods, and slicing.

We will use Python lists to understand permutations, understanding the sign of a permutation in terms of transpositions, cycle-type, and inversions.

Then we use Python lists to implement and optimize the Sieve of Eratosthenes, which will produce a list of all prime numbers up to a big number (like 10 million) in a snap. Along the way, we introduce some Python techniques for data analysis and visualization.

Table of Contents

Primality testing

Before diving into lists, we recall the **brute force** primality test that we created in the last lesson. To test whether a number `n` is prime, we can simply check for factors. This yields the following primality test.

In [7]:

```
def is_prime(n):
    '''
    Checks whether the argument n is a prime number.
    Uses a brute force search for factors between 1 and n.
    '''
    for j in range(2,n):  # the range of numbers 2,3,...,n-1.
        if n%j == 0:  # is n divisible by j?
            print("{} is a factor of {}.".format(j,n))
            return False
    return True
```

We can also implement this test with a **while loop** instead of a for loop. This doesn't make much of a difference, in Python 3.x. (In Python 2.x, this would save memory).

In [8]:
```
def is_prime(n):
    '''
    Checks whether the argument n is a prime number.
    Uses a brute force search for factors between 1 and n.
    '''
    j = 2
    while j < n:  # j will proceed through the list of numbers 2,3,...,n-1.
        if n%j == 0:  # is n divisible by j?
            print("{} is a factor of {}.".format(j,n))
            return False
        j = j + 1  # There's a Python abbreviation for this:  j += 1.
    return True
```

In [9]:
```
is_prime(10001)
```

```
73 is a factor of 10001.
```

Out[9]:
```
False
```

In [10]:
```
is_prime(101)
```

Out[10]:
```
True
```

If n is a prime number, then the `is_prime(n)` function will iterate through all the numbers between 2 and $n-1$. But this is overkill! Indeed, if n is not prime, it will have a factor between 2 and the square root of n. This is because factors come in pairs: if $ab = n$, then one of the factors, a or b, must be less than or equal to the square root of n. So it suffices to search for factors up to (and including) the square root of n.

Even though we've made our own sqrt function, we load a fast one from the standard math package (https://docs.python.org/3/library/math.html). You can use this for square roots, trig functions, logs, and more. Click the previous link for documentation. This package doesn't load automatically when you start Python, so you have to load it with a little Python code.

In [11]:
```
from math import sqrt
```

This command **imports** the square root function (`sqrt`) from the **package** called `math` . Now you can find square roots.

In [12]:
```
sqrt(1000)
```

Out[12]:
```
31.622776601683793
```

There are a few different ways to import functions from packages. The above syntax is a good starting point, but sometimes problems can arise if different packages have functions with the same name. Here are a few methods of importing the `sqrt` function and how they differ.

`from math import sqrt` : After this command, `sqrt` will refer to the function from the `math` package (overriding any previous definition).

`import math` : After this command, all the functions from the `math` package will be imported. But to call `sqrt` , you would type a command like `math.sqrt(1000)` . This is convenient if there are potential conflicts with other packages.

`from math import *` : After this command, all the functions from the `math` package will be imported. To call them, you can access them directly with a command like `sqrt(1000)` . This can easily cause conflicts with other packages, since packages can have hundreds of functions in them!

`import math as mth` : Some people like abbreviations. This imports all the functions from the `math` package. To call one, you type a command like `mth.sqrt(1000)` .

In [13]:

```
import math
```

In [14]:

```
math.sqrt(1000)
```

Out[14]:

```
31.622776601683793
```

In [15]:

```
factorial(10)   # This will cause an error!
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-15-e379444d4994> in <module>()
----> 1 factorial(10)   # This will cause an error!

NameError: name 'factorial' is not defined
```

In [16]:

```
math.factorial(10)   # This is ok, since the math package comes with a function called factorial.
```

Out[16]:

```
3628800
```

Now let's improve our `is_prime(n)` function by searching for factors only up to the square root of the number `n` . We consider two options.

In [17]:

```
def is_prime_slow(n):
    '''
    Checks whether the argument n is a prime number.
    Uses a brute force search for factors between 1 and n.
    '''
    j = 2
    while j <= sqrt(n):   # j will proceed through the list of numbers 2,3,... up to sqrt(n).
        if n%j == 0:   # is n divisible by j?
            print("{} is a factor of {}.".format(j,n))
            return False
        j = j + 1   # There's a Python abbreviation for this:  j += 1.
    return True
```

```
In [18]: def is_prime_fast(n):
             '''
             Checks whether the argument n is a prime number.
             Uses a brute force search for factors between 1 and n.
             '''
             j = 2
             root_n = sqrt(n)
             while j <= root_n:  # j will proceed through the list of numbers 2,3,... up to
         sqrt(n).
                 if n%j == 0:  # is n divisible by j?
                     print("{} is a factor of {}.".format(j,n))
                     return False
                 j = j + 1  # There's a Python abbreviation for this:  j += 1.
             return True
```

```
In [19]: is_prime_fast(1000003)

Out[19]: True
```

```
In [20]: is_prime_slow(1000003)

Out[20]: True
```

I've chosen function names with "fast" and "slow" in them. But what makes them faster or slower? Are they faster than the original? And how can we tell?

Python comes with a great set of tools for these questions. The simplest (for the user) are the time utilities. By placing the **magic** `%timeit` before a command, Python does something like the following:

1. Python makes a little container in your computer devoted to the computations, to avoid interference from other running programs if possible.
2. Python executes the command lots and lots of times.
3. Python averages the amount of time taken for each execution.

Give it a try below, to compare the speed of the functions `is_prime` (the original) with the new `is_prime_fast` and `is_prime_slow`. Note that the `%timeit` commands might take a little while.

```
In [21]: %timeit is_prime_fast(1000003)

         262 µs ± 54.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
In [22]: %timeit is_prime_slow(1000003)

         526 µs ± 57.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
In [23]: %timeit is_prime(1000003)

         211 ms ± 34.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Time is measured in seconds, milliseconds (1 ms = 1/1000 second), microseconds (1 µs = 1/1,000,000 second), and nanoseconds (1 ns = 1/1,000,000,000 second). So it might appear at first that `is_prime` is the fastest, or about the same speed. But check the units! The other two approaches are about a thousand times faster! How much faster were they on your computer?

In [24]:

```
is_prime_fast(10000000000037)  # Don't try this with `is_prime` unless you want to
wait for a long time!
```

Out[24]:

```
True
```

Indeed, the `is_prime_fast(n)` function will go through a loop of length about `sqrt(n)` when `n` is prime. But `is_prime(n)` will go through a loop of length about `n` . Since `sqrt(n)` is much less than `n` , especially when `n` is large, the `is_prime_fast(n)` function is much faster.

Between `is_prime_fast` and `is_prime_slow` , the difference is that the `fast` version **precomputes** the square root `sqrt(n)` before going through the loop, where the `slow` version repeats the `sqrt(n)` every time the loop is repeated. Indeed, writing `while j <= sqrt(n):` suggests that Python might execute `sqrt(n)` every time to check. This *might* lead to Python computing the same square root a million times... unnecessarily!

A basic principle of programming is to **avoid repetition**. If you have the memory space, just compute once and store the result. It will probably be faster to pull the result out of memory than to compute it again.

Python does tend to be pretty smart, however. It's possible that Python **is precomputing** `sqrt(n)` even in the slow loop, just because it's clever enough to tell in advance that the same thing is being computed over and over again. This depends on your Python version and takes place behind the scenes. If you want to figure it out, there's a whole set of tools (for advanced programmers) like the [disassembler (https://docs.python.org/3/library/dis.html)](https://docs.python.org/3/library/dis.html) to figure out what Python is doing.

If you feel like looking under the hood, the next few lines will display the `is_prime_fast` and `is_prime_slow` functions to bytecode. Can you see how the `sqrt(n)` computation is carried out differently?

In [25]:

```
from dis import dis
```

```
In [27]: dis(is_prime_fast)
```

```
  6           0 LOAD_CONST               1 (2)
              2 STORE_FAST               1 (j)

  7           4 LOAD_GLOBAL              0 (sqrt)
              6 LOAD_FAST                0 (n)
              8 CALL_FUNCTION            1
             10 STORE_FAST               2 (root_n)

  8          12 SETUP_LOOP              52 (to 66)
        >>   14 LOAD_FAST                1 (j)
             16 LOAD_FAST                2 (root_n)
             18 COMPARE_OP               1 (<=)
             20 POP_JUMP_IF_FALSE       64

  9          22 LOAD_FAST                0 (n)
             24 LOAD_FAST                1 (j)
             26 BINARY_MODULO
             28 LOAD_CONST               2 (0)
             30 COMPARE_OP               2 (==)
             32 POP_JUMP_IF_FALSE       54

 10          34 LOAD_GLOBAL              1 (print)
             36 LOAD_CONST               3 ('{} is a factor of {}.')
             38 LOAD_METHOD              2 (format)
             40 LOAD_FAST                1 (j)
             42 LOAD_FAST                0 (n)
             44 CALL_METHOD              2
             46 CALL_FUNCTION            1
             48 POP_TOP

 11          50 LOAD_CONST               4 (False)
             52 RETURN_VALUE

 12     >>   54 LOAD_FAST                1 (j)
             56 LOAD_CONST               5 (1)
             58 BINARY_ADD
             60 STORE_FAST               1 (j)
             62 JUMP_ABSOLUTE           14
        >>   64 POP_BLOCK

 13     >>   66 LOAD_CONST               6 (True)
             68 RETURN_VALUE
```

```
In [28]: dis(is_prime_slow)
```

```
  6           0 LOAD_CONST               1 (2)
              2 STORE_FAST               1 (j)

  7           4 SETUP_LOOP              56 (to 62)
        >>    6 LOAD_FAST                1 (j)
              8 LOAD_GLOBAL              0 (sqrt)
             10 LOAD_FAST                0 (n)
             12 CALL_FUNCTION            1
             14 COMPARE_OP               1 (<=)
             16 POP_JUMP_IF_FALSE       60

  8          18 LOAD_FAST                0 (n)
             20 LOAD_FAST                1 (j)
             22 BINARY_MODULO
             24 LOAD_CONST               2 (0)
             26 COMPARE_OP               2 (==)
             28 POP_JUMP_IF_FALSE       50

  9          30 LOAD_GLOBAL              1 (print)
             32 LOAD_CONST               3 ('{} is a factor of {}.')
             34 LOAD_METHOD              2 (format)
             36 LOAD_FAST                1 (j)
             38 LOAD_FAST                0 (n)
             40 CALL_METHOD              2
             42 CALL_FUNCTION            1
             44 POP_TOP

 10          46 LOAD_CONST               4 (False)
             48 RETURN_VALUE

 11     >>   50 LOAD_FAST                1 (j)
             52 LOAD_CONST               5 (1)
             54 BINARY_ADD
             56 STORE_FAST               1 (j)
             58 JUMP_ABSOLUTE            6
        >>   60 POP_BLOCK

 12     >>   62 LOAD_CONST               6 (True)
             64 RETURN_VALUE
```

```
In [29]: %timeit is_prime_fast(10**14 + 37) # This might get a bit of delay.
```

```
1858741 is a factor of 100000000000037.
1858741 is a factor of 100000000000037.
1858741 is a factor of 100000000000037.
1858741 is a factor of 100000000000037.
1858741 is a factor of 100000000000037.
1858741 is a factor of 100000000000037.
1858741 is a factor of 100000000000037.
1858741 is a factor of 100000000000037.
382 ms ± 15 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Now we have a function `is_prime_fast(n)` that is speedy for numbers `n` in the trillions! You'll probably start to hit a delay around 10^{15} or so, and the delays will become intolerable if you add too many more digits. In a future lesson, we will see a different primality test that will be essentially instant even for numbers around 10^{1000}!

Exercises

1. To check whether a number `n` is prime, you can first check whether `n` is even, and then check whether `n` has any odd factors. Change the `is_prime_fast` function by implementing this improvement. How much of a speedup did you get?
2. Use the `%timeit` tool to study the speed of `is_prime_fast` for various sizes of `n` . Using about 10 data points, relate the size of `n` to the time taken by the `is_prime_fast` function.
3. Write a function `is_square(n)` to test whether a given integer `n` is a perfect square (like 0, 1, 4, 9, 16, etc.). How fast can you make it run? Describe the different approaches you try and which are fastest.

In [30]:

```
#1
def is_prime_fast(n):
    if n%2==0:
        #print("{} is even.".format(n))
        return False
    for i in range(1,n,2):
        if n%i==0:
            #print("{} has an odd factor.".format(n))
            return False
    j = 2
    root_n = sqrt(n)
    while j <= root_n:
        if n%j == 0:
            #print("{} is a factor of {}.".format(j,n))
            return False
        j = j + 1
    return True
#%timeit is_prime_fast(10**14 + 37)
#The function speed up from 438ms to about 900ns.
#2
#%timeit is_prime_fast(929)
#%timeit is_prime_fast(2153)
#%timeit is_prime_fast(3469)
#%timeit is_prime_fast(4073)
#%timeit is_prime_fast(5003)
#%timeit is_prime_fast(6101)
#%timeit is_prime_fast(7517)
#%timeit is_prime_fast(8999)
#%timeit is_prime_fast(9787)
#With "n" on the x-axis and ns on the y-axis, the linear regression with 9 points i
s "y=.008083x+873".

#3
def is_square(n):
    for i in range((n//2)+1):
        if i**2==n:
            return True
    return False



def is_square2(n):
    if sqrt(n)==(sqrt(n)//1):
        return True
    else:
        return False

def is_square3(n):
    if sqrt(n)==int(sqrt(n)):
        return True
    else:
        return False

%timeit is_square(64)
%timeit is_square2(64)
%timeit is_square3(64)
#The is_square2 fucntion is faster.
```

```
3.64 µs ± 47.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
540 ns ± 47.3 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
541 ns ± 10.9 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

List manipulation

We have already (briefly) encountered the `list` type in Python. Recall that the `range` command produces a range, which can be used to produce a list. For example, `list(range(10))` produces the list `[0,1,2,3,4,5,6,7,8,9]` . You can also create your own list by a writing out its terms, e.g. `L = [4,7,10]` .

Here we work with lists, and a very Pythonic approach to list manipulation. With practice, this can be a powerful tool to write fast algorithms, exploiting the hard-wired capability of your computer to shift and slice large chunks of data. Our eventual application will be to implement the Sieve of Eratosthenes, producing a long list of prime numbers (without using any `is_prime` test along the way).

We begin by creating a list to play with. We mix numbers and strings... just for fun.

```
In [31]: L = [0,'one',2,'three',4,'five',6,'seven',8,'nine',10]
```

List terms and indices

Notice that the entries in a list can be of any type. The above list `L` has some integer entries and some string entries. Lists are **ordered** in Python, **starting at zero**. One can access the n^{th} entry in a list with a command like `L[n]` .

```
In [32]: L[3]

Out[32]: 'three'
```

```
In [33]: print(L[3])  # Note that Python has slightly different approaches to the print-func
         tion, and the output above.

         three
```

```
In [34]: print(L[4])  # We will use the print function, because it makes our printing intent
         ions clear.

         4
```

```
In [35]: print(L[0])

         0
```

The location of an entry is called its **index**. So *at* the index 3, the list `L` stores the entry `three` . Note that the same entry can occur in many places in a list. E.g. `[7,7,7]` is a list with 7 at the zeroth, first, and second index.

```
In [36]: print(L[-1])
         print(L[-2])

         10
         nine
```

The last bit of code demonstrates a cool Python trick. The "-1st" entry in a list refers to the last entry. The "-2nd entry" refers to the second-to-last entry, and so on. It gives a convenient way to access both sides of the list, even if you don't know how long it is.

Of course, you can use Python to find out how long a list is.

```
In [37]: len(L)

Out[37]: 11
```

You can also use Python to find the sum of a list of numbers.

```
In [38]: sum([1,2,3,4,5])

Out[38]: 15
```

```
In [39]: sum(range(100))  # Be careful.  This is the sum of which numbers?  # The sum functi
         on can take lists or ranges.

Out[39]: 4950
```

List slicing

Slicing lists allows us to create new lists (or ranges) from old lists (or ranges), by chopping off one end or the other, or even slicing out entries at a fixed interval. The simplest syntax has the form `L[a:b]` where `a` denotes the index of the starting entry and index of the final entry is one less than `b` . It is best to try a few examples to get a feel for it.

Slicing a list with a command like `L[a:b]` doesn't actually *change* the original list `L` . It just extracts some terms from the list and outputs those terms. Soon enough, we will change the list `L` using a list assignment.

```
In [40]: L[0:5]

Out[40]: [0, 'one', 2, 'three', 4]
```

```
In [41]: L[5:11]  # Notice that L[0:5] and L[5:11] together recover the whole list.

Out[41]: ['five', 6, 'seven', 8, 'nine', 10]
```

```
In [42]: L[3:7]

Out[42]: ['three', 4, 'five', 6]
```

This continues the strange (for beginners) Python convention of starting at the first number and ending just before the last number. Compare to `range(3,7)` , for example.

The command `L[0:5]` can be replaced by `L[:5]` to abbreviate. The empty opening index tells Python to start at the beginning. Similarly, the command `L[5:11]` can be replaced by `L[5:]` . The empty closing index tells Python to end the slice and the end. This is helpful if one doesn't know where the list ends.

```
In [43]: L[:5]

Out[43]: [0, 'one', 2, 'three', 4]
```

```
In [44]: L[3:]

Out[44]: ['three', 4, 'five', 6, 'seven', 8, 'nine', 10]
```

Just like the `range` command, list slicing can take an optional third argument to give a step size. To understand this, try the command below.

```
In [45]: L[2:10]

Out[45]: [2, 'three', 4, 'five', 6, 'seven', 8, 'nine']
```

```
In [46]: L[2:10:3]

Out[46]: [2, 'five', 8]
```

If, in this three-argument syntax, the first or second argument is absent, then the slice starts at the beginning of the list or ends at the end of the list accordingly.

```
In [47]: L  # Just a reminder.  We haven't modified the original list!

Out[47]: [0, 'one', 2, 'three', 4, 'five', 6, 'seven', 8, 'nine', 10]
```

```
In [48]: L[:9:3]  # Start at zero, go up to (but not including) 9, by steps of 3.

Out[48]: [0, 'three', 6]
```

```
In [49]: L[2: :3] # Start at two, go up through the end of the list, by steps of 3.

Out[49]: [2, 'five', 8]
```

```
In [50]: L[::3]  # Start at zero, go up through the end of the list, by steps of 3.

Out[50]: [0, 'three', 6, 'nine']
```

Changing list slices

Not only can we extract and study terms or slices of a list, we can change them by assignment. The simplest case would be changing a single term of a list.

```
In [51]: print(L) # Start with the list L.

         [0, 'one', 2, 'three', 4, 'five', 6, 'seven', 8, 'nine', 10]
```

```
In [52]: L[5] = 'Bacon!'
```

```
In [53]: print(L)   # What do you think L is now?

         [0, 'one', 2, 'three', 4, 'Bacon!', 6, 'seven', 8, 'nine', 10]
```

```
In [54]: print(L[2::3]) # What do you think this will do?

         [2, 'Bacon!', 8]
```

We can change an entire slice of a list with a single assignment. Let's change the first two terms of `L` in one line.

```
In [55]: L[:2] = ['Pancakes', 'Ham']   # What was L[:2] before?
```

```
In [56]: print(L) # Oh... what have we done!

         ['Pancakes', 'Ham', 2, 'three', 4, 'Bacon!', 6, 'seven', 8, 'nine', 10]
```

```
In [57]: L[0]

Out[57]: 'Pancakes'
```

```
In [58]: L[1]

Out[58]: 'Ham'
```

```
In [59]: L[2]

Out[59]: 2
```

We can change a slice of a list with a single assignment, even when that slice does not consist of consecutive terms. Try to predict what the following commands will do.

```
In [60]: print(L)  # Let's see what the list looks like before.

         ['Pancakes', 'Ham', 2, 'three', 4, 'Bacon!', 6, 'seven', 8, 'nine', 10]
```

```
In [61]: L[::2] = ['A','B','C','D','E','F']  # What was L[::2] before this assignment?
```

```
In [62]: print(L)  # What do you predict?

         ['A', 'Ham', 'B', 'three', 'C', 'Bacon!', 'D', 'seven', 'E', 'nine', 'F']
```

List methods

A method is a function that is attached to an object. We have already used one method: the `format` method that is attached to all strings. You might have seen the `replace` method for strings too. Note that single-quotes `'Hello'` or double-quotes `"Hello"` can be used for strings.

```
In [63]: "Hello {}!".format('programming student')

Out[63]: 'Hello programming student!'
```

```
In [64]: "Programming is fun!".replace('fun','lit')

Out[64]: 'Programming is lit!'
```

List methods are functions attached to lists. Some useful methods include `append` and `sort` . A fuller listing can be found at the official documentation (https://docs.python.org/3/tutorial/datastructures.html).

In [65]:

```python
L = [1,2,3]
L.append(4)
print(L)
```

```
[1, 2, 3, 4]
```

The `append` method can be used to add new items to the end of a list. But be careful if you want to add multiple items!

In [66]:

```python
L.append([5,6])
print(L)
```

```
[1, 2, 3, 4, [5, 6]]
```

Behind the scenes, methods are functions which have a special input parameter called `self` . So when you use a command like `L.append(4)` , you are effectively running `append(L, 4)` . The `self` parameter is the object the method is attached to.

Like all functions, methods have outputs too. But what can be confusing is that methods can *modify* `self` and can sometimes *return* `None` .

In [67]:

```python
#print([1,2,3].append(4))
print("123".replace("3","4"))
```

```
124
```

This is very confusing at first! The list `append` method *does* change `self` by appending something to `self` . But as a function, it returns `None` .

On the other hand, the string `replace` method *does not* change `self` and instead *returns* the modified string.

This will make more sense after we study *mutable* and *immutable* types. Lists are mutable (and thus are often changed by their methods). Strings are immutable, and so changes are effected by producing new strings. Another example of a string method is `sort()` . The only parameter of `sort` is `self` , and so nothing needs to go between the paraentheses.

In [68]:

```python
L = [4,2,1]   # Make a list.
L.sort()  # Sort the list.  This *changes* L and returns None.
print(L)  # Let's see what L is now.
```

```
[1, 2, 4]
```

In [69]:

```python
L = ['Ukelele', 'Apple', 'Dog', 'Cat' ]
L.sort()
print(L)
```

```
['Apple', 'Cat', 'Dog', 'Ukelele']
```

Sorting numbers is possible, because the Python operator `<` is defined for numbers. Sorting strings is possible, because the Python operator `<` is interpreted alphabetically among strings. If you mix types, Python might not know how to behave... you'll get a TypeError.

In [70]:

```
L = [1,'Apple', 3.14]
L.sort()
print(L)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-70-e3d2589f72dc> in <module>()
      1 L = [1,'Apple', 3.14]
----> 2 L.sort()
      3 print(L)

TypeError: '<' not supported between instances of 'str' and 'int'
```

Exercises

1. Create a list L with L = [1,2,3,...,100] (all the numbers from 1 to 100). What is L[50]?
2. Take the same list L, and extract a slice of the form [5,10,15,...,95] with a command of the form L[a:b:c].
3. Take the same list L, and change all the even numbers to zeros, so that L looks like [1,0,3,0,5,0,...,99,0]. Hint: You might wish to use the list [0]*50.
4. Try the command `L[-1::-1]` on a list. What does it do? Can you guess before executing it? Can you understand why? In fact, strings are indexed like lists. Try setting `L = 'Hello'` and the previous command.
5. Create the list [1,100,3,98,5,96,...,99,0], where the odd terms are in order and the even terms are in reverse order. There are multiple methods!
6. Use the append method with a loop to create a list of perfect squares, [0,1,4,9,16,25,...,10000].

In [ ]:

```
```

In [ ]:

```
#1
L=list(range(1,100))
L[50] #L[50]=51
#2
L=L[4::5]
print(L)
#3
L[1::2]=49*[0]
print(L)
#4
L[-1::-1]
print(L[-1::-1])#The following is the reverse of the original list because the func
tion starts at the end and works towards the front by subtracting one until it reac
hes L[0].
L="Hello"
print(L[-1::-1])#prints "olleH"
#5
L=list(range(1,101))
L[1:100:2]=L[-1:0:-2]
print(L)
#6
list_of_squares=[]
for i in range(0,101):
        list_of_squares.append(i**2)
print(list_of_squares)
```

Sieve of Eratosthenes

The **Sieve of Eratosthenes** (hereafter called "the sieve") is a very fast way of producing long lists of primes, without doing repeated primality checking. The basic idea is to start with all of the natural numbers, and successively filter out, or **sieve** (https://en.wikipedia.org/wiki/Sieve), the multiples of 2, then the multiples of 3, then the multiples of 5, etc., until only primes are left. You can read more about the sieve, and experimental number theory, at The Conversation (https://theconversation.com/why-prime-numbers-still-fascinate-mathematicians-2-300-years-later-92484)

Using list slicing, we can carry out this sieving process efficiently. And with a few more tricks we encounter here, we can carry out the Sieve **very** efficiently.

The basic sieve

The first approach we introduce is a bit naive, but is a good starting place. We will begin with a list of numbers up to 100, and sieve out the appropriate multiples of 2,3,5,7.

In [ ]:
```
primes = list(range(100)) # Let's start with the numbers 0...99.
```

Now, to "filter", i.e., to say that a number is *not* prime, let's just change the number to the value `None` .

In [ ]:
```
primes[0] = None # Zero is not prime.
primes[1] = None # One is not prime.
print(primes) # What have we done?
```

Now let's filter out the multiples of 2, starting at 4. This is the slice `primes[4::2]`

In [ ]:
```
primes[4::2] = [None] * len(primes[4::2])  # The right side is a list of Nones, of
the necessary length.
print(primes) # What have we done?
```

Now we filter out the multiples of 3, starting at 9.

In [ ]:
```
primes[9::3] = [None] * len(primes[9::3])  # The right side is a list of Nones, of
the necessary length.
print(primes) # What have we done?
```

Next the multiples of 5, starting at 25 (the first multiple of 5 greater than 5 that's left!)

In [ ]:
```
primes[25::5] = [None] * len(primes[25::5])  # The right side is a list of Nones, o
f the necessary length.
print(primes) # What have we done?
```

Finally, the multiples of 7, starting at 49 (the first multiple of 7 greater than 7 that's left!)

In [ ]:
```
primes[49::7] = [None] * len(primes[49::7])  # The right side is a list of Nones, o
f the necessary length.
print(primes) # What have we done?
```

What's left? A lot of `None` s and the prime numbers up to 100. We have successfully sieved out all the nonprime numbers in the list, using just four sieving steps (and setting 0 and 1 to `None` manually).

But there's a lot of room for improvement, from beginning to end!

1. The format of the end result is not so nice.
2. We had to sieve each step manually. It would be much better to have a function `prime_list(n)` which would output a list of primes up to `n` without so much supervision.
3. The memory usage will be large, if we need to store all the numbers up to a large `n` at the beginning.

We solve these problems in the following way.

1. We will use a list of **booleans** rather than a list of numbers. The ending list will have a `True` value at prime indices and a `False` value at composite indices. This reduces the memory usage and increases the speed.
2. A `which` function (explained soon) will make the desired list of primes after everything else is done.
3. We will proceed through the sieving steps algorithmically rather than entering each step manually.

Here is a somewhat efficient implementation of the Sieve in Python.

In [ ]:

```
def isprime_list(n):
    flags = [True] * (n+1)  # A list [True, True, True,...] to start.
    flags[0] = False  # Zero is not prime.  So its flag is set to False.
    flags[1] = False  # One is not prime.  So its flag is set to False.
    p = 2  # The first prime is 2.  And we start sieving by multiples of 2.

    while p <= sqrt(n):  # We only need to sieve by p is p <= sqrt(n).
        if flags[p]:  # We sieve the multiples of p if flags[p]=True.
            flags[p*p::p] = [False] * len(flags[p*p::p]) # Sieves out multiples of p, starting at p*p.
        p = p + 1 # Try the next value of p.

    return flags
```

In [ ]:

```
print(isprime_list(100))
```

If you look carefully at the list of booleans, you will notice a `True` value at the 2nd index, the 3rd index, the 5th index, the 7th index, etc.. The indices where the values are `True` are precisely the **prime** indices. Since booleans take the smallest amount of memory of any data type (one **bit** of memory per boolean), your computer can carry out the `isprime_list(n)` function even when `n` is very large.

To be more precise, there are 8 bits in a **byte**. There are 1024 bytes (about 1000) in a kilobyte. There are 1024 kilobytes in a megabyte. There are 1024 megabytes in a gigabyte. Therefore, a gigabyte of memory is enough to store about 8 billion bits. That's enough to store the result of `isprime_list(n)` when `n` is about 8 billion. Not bad! And your computer probably has 4 or 8 or 12 or 16 gigabytes of memory to use.

To transform the list of booleans into a list of prime numbers, we create a function called `where` . This function uses another Python technique called **list comprehension**. We discuss this technique later in this lesson, so just use the `where` function as a tool for now, or [read about list comprehension (https://docs.python.org/2/tutorial/datastructures.html#list-comprehensions)](https://docs.python.org/2/tutorial/datastructures.html#list-comprehensions) if you're curious.

```
In [ ]: def where(L):
            '''
            Take a list of booleans as input and
            outputs the list of indices where True occurs.
            '''
            return [n for n in range(len(L)) if L[n]]
```

Combined with the `isprime_list` function, we can produce long lists of primes.

```
In [ ]: print(where(isprime_list(100)))
```

Let's push it a bit further. How many primes are there between 1 and 1 million? We can figure this out in three steps:

1. Create the isprime_list.
2. Use where to get the list of primes.
3. Find the length of the list of primes.

But it's better to do it in two steps.

1. Create the isprime_list.
2. Sum the list! (Note that `True` is 1, for the purpose of summation!)

```
In [111]: sum(isprime_list(1000000))   # The number of primes up to a million!

Out[111]: 78498
```

```
In [ ]: %timeit isprime_list(10**6)   # 1000 ms = 1 second.
```

```
In [ ]: %timeit sum(isprime_list(10**6))
```

This isn't too bad! It takes a fraction of a second to identify the primes up to a million, and a smaller fraction of a second to count them! But we can do a little better.

The first improvement is to take care of the even numbers first. If we count carefully, then the sequence 4,6,8,...,n (ending at n-1 if n is odd) has the floor of (n-2)/2 terms. Thus the line `flags[4::2] = [False] * ((n-2)//2)` will set all the flags to False in the sequence 4,6,8,10,... From there, we can begin sieving by *odd* primes starting with 3.

The next improvement is that, since we've already sieved out all the even numbers (except 2), we don't have to sieve out again by *even multiples*. So when sieving by multiples of 3, we don't have to sieve out 9,12,15,18,21,etc.. We can just sieve out 9,15,21,etc.. When `p` is an odd prime, this can be taken care of with the code `flags[p*p::2*p] = [False] * len(flags[p*p::2*p])`.

```
In [ ]: def isprime_list(n):
            '''
            Return a list of length n+1
            with Trues at prime indices and Falses at composite indices.
            '''
            flags = [True] * (n+1)  # A list [True, True, True,...] to start.
            flags[0] = False  # Zero is not prime.  So its flag is set to False.
            flags[1] = False  # One is not prime.  So its flag is set to False.
            flags[4::2] = [False] * ((n-2)//2)
            p = 3
            while p <= sqrt(n):  # We only need to sieve by p is p <= sqrt(n).
                if flags[p]:  # We sieve the multiples of p if flags[p]=True.
                    flags[p*p::2*p] = [False] * len(flags[p*p::2*p]) # Sieves out multiples
        of p, starting at p*p.
                p = p + 2 # Try the next value of p.  Note that we can proceed only through
        odd p!

            return flags
```

```
In [ ]: %timeit sum(isprime_list(10**6))  # How much did this speed it up? The new change s
        peeded it up approximately 15ms.
```

Another modest improvement is the following. In the code above, the program *counts* the terms in sequences like 9,15,21,27,..., in order to set them to `False` . This is accomplished with the length command `len(flags[p*p::2*p])` . But that length computation is a bit too intensive. A bit of algebraic work shows that the length is given formulaically in terms of `p` and `n` by the formula:

$$len = \lfloor \frac{n - p^2 - 1}{2p} \rfloor + 1$$

(Here $\lfloor x \rfloor$ denotes the floor function, i.e., the result of rounding down.) Putting this into the code yields the following.

```
In [ ]: def isprime_list(n):
            '''
            Return a list of length n+1
            with Trues at prime indices and Falses at composite indices.
            '''
            flags = [True] * (n+1)  # A list [True, True, True,...] to start.
            flags[0] = False  # Zero is not prime.  So its flag is set to False.
            flags[1] = False  # One is not prime.  So its flag is set to False.
            flags[4::2] = [False] * ((n-2)//2)
            p = 3
            while p <= sqrt(n):  # We only need to sieve by p is p <= sqrt(n).
                if flags[p]:  # We sieve the multiples of p if flags[p]=True.
                    flags[p*p::2*p] = [False] * ((n-p*p-1)//(2*p)+1) # Sieves out multiples
        of p, starting at p*p.
                p = p + 2 # Try the next value of p.

            return flags
```

```
In [ ]: %timeit sum(isprime_list(10**6))  # How much did this speed it up? The new function
        is 15.6ms faster than the last function.
```

That should be pretty fast! It should be under 100 ms (one tenth of one second!) to determine the primes up to a million, and on a newer computer it should be under 50ms. We have gotten pretty close to the fastest algorithms that you can find in Python, without using external packages (like SAGE or sympy). See the related discussion on StackOverflow (https://stackoverflow.com/questions/2068372/fastest-way-to-list-all-primes-below-n)... the code in this lesson was influenced by the code presented there.

Exercises

1. Prove that the length of `range(p*p, n, 2*p)` equals $\lfloor \frac{n-p^2-1}{2p} \rfloor + 1$.
2. A natural number n is called squarefree if it has no perfect square divides n except for 1. Write a function `squarefree_list(n)` which outputs a list of booleans: `True` if the index is squarefree and `False` if the index is not squarefree. For example, if you execute `squarefree_list(12)` , the output should be `[False, True, True, True, False, True, True, True, False, False, True, True, False]` . Note that the `False` entries are located the indices 0, 4, 8, 9, 12. These natural numbers have perfect square divisors besides 1.
3. Your DNA contains about 3 billion base pairs. Each "base pair" can be thought of as a letter, A, T, G, or C. How many bits would be required to store a single base pair? In other words, how might you convert a sequence of booleans into a letter A,T,G, or C? Given this, how many megabytes or gigabytes are required to store your DNA? How many people's DNA would fit on a thumb-drive?

In [92]:

```
#1
#len(range(a:b:c))
#a,a+c,a+2c,...a+kc<b-1 or eqaul to b-1
#0,c,2c...kc<b-a-1 or equal to b-a-1
#0,1,2...k<(b-a-1)/c or equal to (b-a-1)/c
#Take the lower bound of ((b-a-1)/c)+1 to get the range.
#Plug in the values and you get the lower bound of (((b-a-1)/c)+1) to get ((n-p**2-1)/2p)+1

#2
def squarefree_list(n):
    squares=[]
    square_free=[True]*(n+1)
    for i in range(2,int(sqrt(n))+1):
        squares.append(i**2)
    for x in squares:
        square_free[x::x]= [False]*len(square_free[x::x])
    square_free[0]=False
    return square_free

squarefree_list(12)

#3
#A single base pair can be stored in two bits. You could assign a sequence of booleans into A,T,G, and C by assigning 00 to A,
#01 to T, 10 to G, and 11 to C.
#Three billion base pairs would require 6 billion bits. 8 billion bits are in a gigabyte. 6/8 or .75 of a gigabyte are required ot store DNA.
#Assuming a flashdrive holds 64 gigabytes, 64/.75 or 85.33 people's DNA can fit on a thumbdrive.
```

Out[92]:

```
[False,
 True,
 True,
 True,
 False,
 True,
 True,
 True,
 False,
 False,
 True,
 True,
 False]
```

In [ ]:

```
#3
```

Data analysis

Now that we can produce a list of prime numbers quickly, we can do some data analysis: some experimental number theory to look for trends or patterns in the sequence of prime numbers. Since Euclid (about 300 BCE), we have known that there are infinitely many prime numbers. But how are they distributed? What proportion of numbers are prime, and how does this proportion change over different ranges? As theoretical questions, these belong the the field of analytic number theory. But it is hard to know what to prove without doing a bit of experimentation. And so, at least since Gauss (read Tschinkel's article about Gauss's tables) (http://www.ams.org/journals/bull/2006-43-01/S0273-0979-05-01096-7/S0273-0979-05-01096-7.pdf) started examining his extensive tables of prime numbers, mathematicians have been carrying out experimental number theory.

Analyzing the list of primes

Let's begin by creating our data set: the prime numbers up to 1 million.

```
In [ ]: primes = where(isprime_list(1000000))
```

```
In [ ]: len(primes) # Our population size.  A statistician might call it N.
```

```
In [ ]: primes[-1]  # The last prime in our list, just before one million.
```

```
In [ ]: type(primes) # What type is this data?
```

```
In [ ]: print(primes[:100]) # The first hundred prime numbers.
```

To carry out serious analysis, we will use the method of **list comprehension** to place our population into "bins" for statistical analysis. Our first type of list comprehension has the form `[x for x in LIST if CONDITION]` . This produces the list of all elements of LIST satisfying CONDITION. It is similar to list slicing, except we pull out terms from the list according to whether a condition is true or false.

For example, let's divide the (odd) primes into two classes. Red primes will be those of the form 4n+1. Blue primes will be those of the form 4n+3. In other words, a prime `p` is red if `p%4 == 1` and blue if `p%4 == 3` . And the prime 2 is neither red nor blue.

```
In [ ]: redprimes = [p for p in primes if p%4 == 1] # Note the [x for x in LIST if CONDITIO
        N] syntax.
        blueprimes = [p for p in primes if p%4 == 3]

        print('Red primes:',redprimes[:20]) # The first 20 red primes.
        print('Blue primes:',blueprimes[:20]) # The first 20 blue primes.
```

```
In [ ]: print("There are {} red primes and {} blue primes, up to 1 million.".format(len(red
        primes), len(blueprimes)))
```

This is pretty close! It seems like prime numbers are about evenly distributed between red and blue. Their remainder after division by 4 is about as likely to be 1 as it is to be 3. In fact, it is proven that *asymptotically* the ratio between the number of red primes and the number of blue primes approaches 1. However, Chebyshev noticed a persistent slight bias towards blue primes along the way.

Some of the deepest conjectures in mathematics relate to the [prime counting function (https://en.wikipedia.org/wiki/Prime-counting_function)](https://en.wikipedia.org/wiki/Prime-counting_function) $\pi(x)$. Here $\pi(x)$ is the **number of primes** between 1 and x (inclusive). So $\pi(2) = 1$ and $\pi(3) = 2$ and $\pi(4) = 2$ and $\pi(5) = 3$. One can compute a value of $\pi(x)$ pretty easily using a list comprehension.

In [ ]:
```python
def primes_upto(x):
    return len([p for p in primes if p <= x]) # List comprehension recovers the primes up to x.
```

In [ ]:
```python
primes_upto(1000)  # There are 168 primes between 1 and 1000.
```

Now we graph the prime counting function. To do this, we use a list comprehension, and the visualization library called matplotlib. For graphing a function, the basic idea is to create a list of x-values, a list of corresponding y-values (so the lists have to be the same length!), and then we feed the two lists into matplotlib to make the graph.

We begin by loading the necessary packages.

In [ ]:
```python
import matplotlib  #  A powerful graphics package.
import numpy  #  A math package
import matplotlib.pyplot as plt  # A plotting subpackage in matplotlib.
```

Now let's graph the function $y = x^2$ over the domain $-2 \leq x \leq 2$ for practice. As a first step, we use numpy's `linspace` function to create an evenly spaced set of 11 x-values between -2 and 2.

In [ ]:
```python
x_values = numpy.linspace(-2,2,11)  # The argument 11 is the *number* of terms, not the step size!
print(x_values)
type(x_values)
```

You might notice that the format looks a bit different from a list. Indeed, if you check `type(x_values)` , it's not a list but something else called a numpy array. Numpy is a package that excels with computations on large arrays of data. On the surface, it's not so different from a list. The `numpy.linspace` command is a convenient way of producing an evenly spaced list of inputs.

The big difference is that operations on numpy arrays are interpreted differently than operations on ordinary Python lists. Try the two commands for comparison.

In [ ]:
```python
[1,2,3] + [1,2,3]
```

In [ ]:
```python
x_values + x_values
```

In [ ]:
```python
y_values = x_values * x_values  # How is multiplication interpreted on numpy arrays?
print(y_values)
```

Now we use matplotlib to create a simple line graph.

In [ ]:

```
%matplotlib inline
plt.plot(x_values, y_values)
plt.title('The graph of $y = x^2$')  # The dollar signs surround the formula, in LaTeX format.
plt.ylabel('y')
plt.xlabel('x')
plt.grid(True)
plt.show()
```

Let's analyze the graphing code a bit more. See the official pyplot tutorial (https://matplotlib.org/users/pyplot_tutorial.html) for more details.

```
%matplotlib inline
plt.plot(x_values, y_values)
plt.title('The graph of $y = x^2$')  # The dollar signs surround the formula, in LaTeX format.
plt.ylabel('y')
plt.xlabel('x')
plt.grid(True)
plt.show()
```

The first line contains the **magic** `%matplotlib inline`. We have seen a magic word before, in `%timeit`. Magic words (http://ipython.readthedocs.io/en/stable/interactive/magics.html) can call another program to assist. So here, the magic `%matplotlib inline` calls matplotlib for help, and places the resulting figure within the notebook.

The next line `plt.plot(x_values, y_values)` creates a `plot object` based on the data of the x-values and y-values. It is an abstract sort of object, behind the scenes, in a format that matplotlib understands. The following lines set the title of the plot, the axis labels, and turns a grid on. The last line `plt.show` renders the plot as an image in your notebook. There's an infinite variety of graphs that matplotlib can produce -- see the gallery (https://matplotlib.org/gallery.html) for more! Other graphics packages include bokeh (http://bokeh.pydata.org/en/latest/) and seaborn (http://seaborn.pydata.org/), which extends matplotlib.

Analysis of the prime counting function

Now, to analyze the prime counting function, let's graph it. To make a graph, we will first need a list of many values of x and many corresponding values of $\pi(x)$. We do this with two commands. The first might take a minute to compute.

In [ ]:

```
x_values = numpy.linspace(0,1000000,1001) # The numpy array [0,1000,2000,3000,...,1000000]
pix_values = numpy.array([primes_upto(x) for x in x_values])  # [FUNCTION(x) for x in LIST] syntax
```

We created an array of x-values as before. But the creation of an array of y-values (here, called `pix_values` to stand for $\pi(x)$) probably looks strange. We have done two new things!

1. We have used a list comprehension `[primes_upto(x) for x in x_values]` to create a **list** of y-values.
2. We have used numpy.array(LIST) syntax to convert a Python list into a numpy array.

First, we explain the list comprehension. Instead of pulling out values of a list according to a condition, with `[x for x in LIST if CONDITION]` , we have created a new list based on performing a function each element of a list. The syntax, used above, is `[FUNCTION(x) for x in LIST]` . These two methods of list comprehension can be combined, in fact. The most general syntax for list comprehension is `[FUNCTION(x) for x in LIST if CONDITION]` .

Second, a list comprehension can be carried out on a numpy array, but the result is a plain Python list. It will be better to have a numpy array instead for what follows, so we use the `numpy.array()` function to convert the list into a numpy array.

```
In [ ]: type(numpy.array([1,2,3]))  # For example.
```

Now we have two numpy arrays: the array of x-values and the array of y-values. We can make a plot with matplotlib.

```
In [ ]: len(x_values) == len(pix_values)  # These better be the same, or else matplotlib will be unhappy.
```

```
In [ ]: %matplotlib inline
        plt.plot(x_values, pix_values)
        plt.title('The prime counting function')
        plt.ylabel('$\pi(x)$')
        plt.xlabel('x')
        plt.grid(True)
        plt.show()
```

In this range, the prime counting function might look nearly linear. But if you look closely, there's a subtle downward bend. This is more pronounced in smaller ranges. For example, let's look at the first 10 x-values and y-values only.

```
In [ ]: %matplotlib inline
        plt.plot(x_values[:10], pix_values[:10])  # Look closer to 0.
        plt.title('The prime counting function')
        plt.ylabel('$\pi(x)$')
        plt.xlabel('x')
        plt.grid(True)
        plt.show()
```

It still looks almost linear, but there's a visible downward bend here. How can we see this bend more clearly? If the graph were linear, its equation would have the form $\pi(x) = mx$ for some fixed slope m (since the graph *does* pass through the origin). Therefore, the quantity $\pi(x)/x$ would be *constant* if the graph were linear.

Hence, if we graph $\pi(x)/x$ on the y-axis and x on the x-axis, and the result is nonconstant, then the function $\pi(x)$ is nonlinear.

```
In [ ]: m_values = pix_values[1:] / x_values[1:]  # We start at 1, to avoid a division by zero error.
```

In [ ]:

```
%matplotlib inline
plt.plot(x_values[1:], m_values)
plt.title('The ratio $\pi(x) / x$ as $x$ varies.')
plt.xlabel('x')
plt.ylabel('$\pi(x) / x$')
plt.grid(True)
plt.show()
```

That is certainly not constant! The decay of $\pi(x)/x$ is not so different from $1/\log(x)$, in fact. To see this, let's overlay the graphs. We use the `numpy.log` function, which computes the natural logarithm of its input (and allows an entire array as input).

In [ ]:

```
%matplotlib inline
plt.plot(x_values[1:], m_values, label='$\pi(x)/x$')  # The same as the plot above.
plt.plot(x_values[1:], 1 / numpy.log(x_values[1:]), label='$1 / \log(x)$')  # Overlay the graph of 1 / log(x)
plt.title('The ratio of $\pi(x) / x$ as $x$ varies.')
plt.xlabel('x')
plt.ylabel('$\pi(x) / x$')
plt.grid(True)
plt.legend()  # Turn on the legend.
plt.show()
```

The shape of the decay of $\pi(x)/x$ is very close to $1/\log(x)$, but it looks like there is an offset. In fact, there is, and it is pretty close to $1/\log(x)^2$. And that is close, but again there's another little offset, this time proportional to $2/\log(x)^3$. This goes on forever, if one wishes to approximate $\pi(x)/x$ by an "asymptotic expansion" (not a good idea, it turns out).

The closeness of $\pi(x)/x$ to $1/\log(x)$ is expressed in the **prime number theorem**:

$$\lim_{x \rightarrow \infty} \frac{\pi(x)}{x/\log(x)} = 1.$$

In [ ]:

```
%matplotlib inline
plt.plot(x_values[1:], m_values * numpy.log(x_values[1:])  )  # Should get closer to 1.
plt.title('The ratio $\pi(x) / (x / \log(x))$ approaches 1... slowly')
plt.xlabel('x')
plt.ylabel('$\pi(x) / (x / \log(x)) $')
plt.ylim(0.8,1.2)
plt.grid(True)
plt.show()
```

Comparing the graph to the theoretical result, we find that the ratio $\pi(x)/(x/\log(x))$ approaches 1 (the theoretical result) but very slowly (see the graph above!).

A much stronger result relates $\pi(x)$ to the "logarithmic integral" $li(x)$. The Riemann hypothesis (http://www.claymath.org/millennium-problems/riemann-hypothesis) is equivalent to the statement

$$|\pi(x) - li(x)| = O(\sqrt{x}\log(x)).$$

In other words, the error if one approximates $\pi(x)$ by $li(x)$ is bounded by a constant times $\sqrt{x}\log(x)$. The logarithmic integral function isn't part of Python or numpy, but it is in the mpmath package. If you have this package installed, then you can try the following.

In [ ]:

```
from mpmath import li
```

```
In [ ]: print(primes_upto(1000000))  # The number of primes up to 1 million.
        print(li(1000000))  # The logarithmic integral of 1 million.
```

Not too shabby!

Prime gaps

As a last bit of data analysis, we consider the **prime gaps**. These are the numbers that occur as differences between consecutive primes. Since all primes except 2 are odd, all prime gaps are even except for the 1-unit gap between 2 and 3. There are many unsolved problems about prime gaps; the most famous might be that a gap of 2 occurs infinitely often (as in the gaps between 3,5 and between 11,13 and between 41,43, etc.).

Once we have our data set of prime numbers, it is not hard to create a data set of prime gaps. Recall that `primes` is our list of prime numbers up to 1 million.

```
In [ ]: len(primes) # The number of primes up to 1 million.
```

```
In [ ]: primes_allbutlast = primes[:-1]  # This excludes the last prime in the list.
        primes_allbutfirst = primes[1:]  # This excludes the first (i.e., with index 0) prime in the list.
```

```
In [ ]: primegaps = numpy.array(primes_allbutfirst) - numpy.array(primes_allbutlast) # Numpy is fast!
```

```
In [ ]: print(primegaps[:100])  # The first hundred prime gaps!
```

What have we done? It is useful to try out this method on a short list.

```
In [ ]: L = [1,3,7,20]  # A nice short list.
```

```
In [ ]: print(L[:-1])
        print(L[1:])
```

Now we have two lists of the same length. The gaps in the original list `L` are the differences between terms of the *same* index in the two new lists. One might be tempted to just subtract, e.g., with the command `L[1:] - L[:-1]` , but subtraction is not defined for lists.

Fortunately, by converting the lists to numpy arrays, we can use numpy's term-by-term subtraction operation.

```
In [ ]: L[1:] - L[:-1]  # This will give a TypeError.  You can't subtract lists!
```

```
In [ ]: numpy.array(L[1:]) - numpy.array(L[:-1])  # That's better.  See the gaps in the list [1,3,7,20] in the output.
```

Now let's return to our primegaps data set. It contains all the gap-sizes for primes up to 1 million.

```
In [ ]: print(len(primes))
        print(len(primegaps))  # This should be one less than the number of primes.
```

As a last example of data visualization, we use matplotlib to produce a histogram of the prime gaps.

```
In [ ]: max(primegaps)   # The largest prime gap that appears!
```

```
In [ ]: %matplotlib inline
        plt.figure(figsize=(12, 5))   #  Makes the resulting figure 12in by 5in.
        plt.hist(primegaps, bins=range(1,115)) #  Makes a histogram with one bin for each possible gap from 1 to 114.
        plt.ylabel('Frequency')
        plt.xlabel('Gap size')
        plt.grid(True)
        plt.title('The frequency of prime gaps, for primes up to 1 million')
        plt.show()
```

Observe that gaps of 2 (twin primes) are pretty frequent. There are over 8000 of them, and about the same number of 4-unit gaps! But gaps of 6 are most frequent in the population, and there are some interesting peaks at 6, 12, 18, 24, 30. What else do you observe?

Exercises

1. Create functions `redprimes_upto(x)` and `blueprimes_upto(x)` which count the number of red/blue primes up to a given number `x`. Recall that we defined red/blue primes to be those of the form 4n+1 or 4n+3, respectively. Graph the relative proportion of red/blue primes as `x` varies from 1 to 1 million. E.g., are the proportions 50%/50% or 70%/30%, and how do these proportions change? Note: this is also visualized in An Illustrated Theory of Numbers (http://bookstore.ams.org/mbk-105) and you can read an article by Rubinstein and Sarnak (https://projecteuclid.org/euclid.em/1048515870) for more.
2. Does there seem to be a bias in the last digits of primes? Note that, except for 2 and 5, every prime ends in 1,3,7, or 9. Note: the last digit of a number `n` is obtained from `n % 10`.
3. Read about the "Prime Conspiracy" (https://www.quantamagazine.org/mathematicians-discover-prime-conspiracy-20160313), recently discovered by Lemke Oliver and Soundararajan. Can you detect their conspiracy in our data set of primes?

In [6]:

```
#1
def redprimes_upto(n):
    redprimes = [p for p in primes if p%4 == 1]
    list=[]
    for i in redprimes:
        list.append(i)
        if i>n:
            return (len(list)-1)

def blueprimes_upto(n):
    blueprimes = [p for p in primes if p%4 == 3]
    list=[]
    for i in blueprimes:
        list.append(i)
        if i>n:
            return (len(list)-1)
#2
mod_primes=[]
list_of_primes=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151,
157, 163, 167, 173, 179, 181, 191, 193, 197, 199]
print(len(list_of_primes))
for i in list_of_primes:
    mod_primes.append(i%10)
print(mod_primes)
#There is clearly a bias for certain last digits in primes with only 1,3,7, and 9.
#3
#A prime ending in "1" is 40% likely to be followed by a 3, 50% to be followed by a
7, and 10% to be followed by a 1 in the first 46 primes.
```

```
46
[2, 3, 5, 7, 1, 3, 7, 9, 3, 9, 1, 7, 1, 3, 7, 3, 9, 1, 7, 1, 3, 9, 3, 9, 7, 1, 3
, 7, 9, 3, 7, 1, 7, 9, 9, 1, 7, 3, 7, 3, 9, 1, 1, 3, 7, 9]
```

In [ ]:

In [1]:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-f7a63d5ada3e> in <module>()
----> 1 km

NameError: name 'km' is not defined
```

In [ ]:

In [ ]:

In [ ]: