Finding Primes

Because primes are the numbers that fall through the gaps of the multiplication table, they are notoriously hard to find, and it's difficult even for computers to quickly test if a very large number is prime or not. Modern encryption methods such as RSA depend on the fact that large numbers are hard to factorise.

A prime number is an integer which:

- ***Has <u>exactly</u> two factors (whole numbers that divide perfectly into it).***
- ***Cannot be broken down into smaller integer factors (except 1 and itself).***

Jot down all the prime numbers below 20:

How do you know these numbers are prime? How do you know you've found them all?

True or False: see how good your instincts are for these prime number 'facts':

1. There is a formula for the n^{th} prime.
2. There is a formula for the number of primes below n.
3. There are infinitely many primes.
4. There are infinitely many twin primes (eg 17 & 19; primes that differ by 2).
5. All primes are one more or less than a multiple of 6.
6. Any even integer greater than 2 can be written as the sum of two primes.
7. Every integer greater than 1 can be written uniquely as the product of primes.
8. $100,000 was offered to anyone who could find the factors of a 309 digit number.

If you were tasked with finding all the prime numbers below 1000, how would you do it? Don't worry about the efficiency of your method for now: just try to describe a process that, however tedious, would reliably identify all prime numbers.
You don't need to get too specific (eg "Check if it divides by 17" is sufficient).

As you write your instructions, try to keep in mind how your instructions could be interpreted by the computer. A good rule of thumb when writing code is to imagine the computer as a genius with no common sense. They will follow your instructions exactly, which is not always what you want. The classic 'coder goes shopping' joke sums it up:

"Buy me 2 pints of milk. If they have eggs, get 6." *<returns with 6 pints of milk>*

"Why did you bring me 6 pints of milk?!?" *"Because they had eggs!"*

Turn over to see how to code it up …

Finding Primes *with Python*

Finding primes is an ideal task for a computer: it is tedious and computation-heavy. The largest known prime is $2^{82589933} - 1$, which has over 24 million digits, and we have to use all sorts of sophisticated methods (plus crowd-sourcing the computer power) to deal with numbers that massive. For the first few thousand primes, though, we can use brute force.

Pseudo-English

This is a brief description of what I want the program to do, written in no specific programming language.

1. Make an empty list called **primes**, which I will use to store the primes I find as I go along.

2. Loop through all values **i** from 2 to 1000, testing each one as follows:
   i. Make an accumulator called **num_factors** to keep track of how many factors **i** has.
   ii. Loop through each value **j** from 2 up to (but not including) **i**, testing to see if **j** is a factor of **i**.
   iii. If I find a factor, increment **num_factors** by 1.
   iv. At the end of this inner loop, if **num_factors** is equal to 0, append **i** to **primes**.

3. After the outer loop has finished, print out **primes** (or maybe just the *length* of **primes** if it's a big list!)

4. Output the list to a text file called **primes_to_1000.txt**.

Optimisation: You might want to consider if there are any obvious optimisations you could implement at this point. Are there any unnecessary tests carried out by the code, or any numbers it can safely skip?

Now try to turn each of these steps into Python code, then check back to see mine…

Prime Finder (version 1)

In general, it is hard even for the author of a program to easily make sense of an entire batch of code. First, read through the program from top to bottom, not worrying about the details, but concentrating instead on the general 'flow' of the program:
"Which bits happen when?" and "What loops are involved?" and "What gets printed and when?"

After that, you can read through again, focusing in detail on each individual block to work out exactly what it does and how: "What happens in this For loop?" or "What is meant by that If statement?"

The outer loop

```
1   primes = []
2   for i in range(2, 1000):
3     factors = 0
4     for j in range(2, i):
5       if i % j == 0:
6         factors += 1
7     if factors == 0:
8       primes.append(i)
9
10  print(f"Found {len(primes)} primes")
11
12  with open("primes_to_1000.txt", "w") as output:
13    for p in primes:
14      output.write(f"{p}\n")
```

An empty list is made at the beginning, then a loop, starting at 2 and continuing up to (but not including) 1000.

The body of the loop uses an accumulator (the variable **factors**) to keep track of how many values **i** divides by. If it's 0, it means we've found a prime.

The inner loop

```
1   primes = []
2   for i in range(2, 1000):
3     factors = 0
4     for j in range(2, i):
5       if i % j == 0:
6         factors += 1
7     if factors == 0:
8       primes.append(i)
9
10  print(f"Found {len(primes)} primes")
11
12  with open("primes_to_1000.txt", "w") as output:
13    for p in primes:
14      output.write(f"{p}\n")
```

The inner **j** loop cycles through all values from 2 up to (but not including) **i**, which means that when the outer loop sets **i** to 7, the inner loop will test to see if 7 divides by 2, 3, 4, 5 and finally 6.

It tests using the **%** operation: if the result is 0 it means **i** divides by **j**.

If no factors are found (that is, **factors** is still equal to 0 when the inner loop is finished), then **i** is appended to the **primes** list.

Displaying results

```
1   primes = []
2   for i in range(2, 1000):
3     factors = 0
4     for j in range(2, i):
5       if i % j == 0:
6         factors += 1
7     if factors == 0:
8       primes.append(i)
9
10  print(f"Found {len(primes)} primes")
11
12  with open("primes_to_1000.txt", "w") as output:
13    for p in primes:
14      output.write(f"{p}\n")
```

We could choose to include a print statement at the same time as appending **i** to **primes**, but apart from slowing down the program, it is of limited use to have all the primes appear on the screen.

Instead, the *length* of the final list is displayed, using the built-in **len** function.

Next, a new text document is created (in *write* mode: **"w"**), using the **with** block. The **write** statement adds the text enclosed to the file (and **\n** is the 'new line' character).

Optimising the code

The algorithm (that is, the method) used above is pretty inefficient. It will find all primes below 10,000 in about 15 seconds, and over a minute to get up to 20,000. With a few tweaks, we can do a lot better…

A few ideas for optimisation:

- Even if the number being tested is even, our code continues to blindly test every single number, despite the fact that, after testing division by 2, we already know it's not a prime. It would be a big time saver if we could jump straight to the next candidate. See **Using break** on the next page.
- 52 can't be a factor of 100, because it's more than half of 100. Our code spends half its time testing numbers that we know are too big. See **Clever limits** on the next page to improve this.
- Our code tests divisibility against all possible numbers, but if a number doesn't divide by 2 or 3, we don't need to test division by 4, 6, 8, 9, etc. In reality, we only need to test against prime numbers. Even though the program's job is to find them, we still have enough in our list to test against the next potential candidate. See **Using primes to find primes** on the next page.

Extending the code

- The *Goldbach Conjecture* states that every even number greater than 2 can be expressed as the sum of two primes (eg $18 = 5 + 13$). Can you verify this for the first 1000 even numbers?

- The *Twin Primes* conjecture states that there are an infinite number of primes with a difference of two (eg 11 and 13). Can you find all the twin primes below 1,000,000?

- Can you write a program that takes an input value and produces a list of all *factor pairs*?

Using break

Often it is necessary to 'break out of' a loop of code before it has fully completed. In this case, it would be helpful if we could stop testing for divisibility as soon as we hit a number which is a factor.

```
1  primes = []
2  for i in range(2, 1000):
3      for j in range(2, i):
4          if i % j == 0:
5              break
6      else:
7          primes.append(i)
```

The **break** statement informs the code to exit the loop prematurely. This means that, when the outer loop provides a value of, say, 333, for **i**, the **j** loop starts by testing divisibility by 2, then changes **j** to 3, but then discovers a factor and breaks out before the **j** loop is done.

The **else** statement is in line with the inner **for** loop. This encloses a block of code that *only* executes if the loop it's attached to has terminated normally (ie, the loop completed and *didn't* find a factor).

Clever limits

```
1  primes = []
2
3  for i in range(2, 1000):
4      for j in range(2, i // 2 + 1):
5          if i % j == 0:
6              break
7      else:
8          primes.append(i)
```

```
1  primes = []
2
3  for i in range(2, 1000):
4      for j in range(2, int(i ** 0.5)):
5          if i % j == 0:
6              break
7      else:
8          primes.append(i)
```

The inner loop need not test values that are larger than half of **i**. We can use integer division (the **//** operation) to ensure we get a whole number answer here. Note that adding 1 is necessary because when **i** is 4, **i // 2** gives 2, and since the object **range(2, 2)** does not include the final value 2, the loop would execute 0 times, complete without a **break**, and append the value 4 to our list of primes.

With a bit more thought, we can improve our limits still further. Imagine listing factors of 100 *in pairs*. Once you reach 10×10 you are done because all factors greater than 10 already appear in your list. In other words, if a number has a factor greater than its square-root, it must pair with a factor less than its square-root. We only need check values up to the largest whole number no greater than **i ** 0.5**.

Using primes to find primes

```
1  primes = [2, 3]
2
3  for i in range(5, 1000):
4      maxp = int(i ** 0.5)
5      for p in primes:
6          if p <= maxp and i % p == 0:
7              break
8      else:
9          primes.append(i)
```

```
1  primes = [2, 3]
2
3  for i in range(5, 1000, 2):
4      maxp = int(i ** 0.5)
5      p = 0
6      while primes[p] <= maxp:
7          if i % primes[p] == 0:
8              break
9          p += 1
10     else:
11         primes.append(i)
```

By the time our code tests to see if 349 divides by 35, it has already tested to see if 5 is a factor. We only need check divisibility by primes, not every number.

By 'priming' our list of primes with a couple of starting values, we can check against values in that list only.

maxp allows us to skip over any primes which are too big.

But if we can define our loop so that it only tests primes which are small enough in the first place, we can save even more time (the version above still runs the loop to the end, testing each time to see if **p** is less than **maxp**).

For this, we need a **while** loop, which will continue to run as long as the prime in position **p** isn't too big. (Note: in the code above, **p** was an element in **primes**, so it was a prime number. In this code, it's being used as an **index**, so that the prime number in position **p** is **primes[p]**).