PDSP 2024, Lecture 04, 20 August 2024

Primes upto n

Last time we wrote these functions to compute prime numbers upto n.

In [1]:
```
def isprime(n):
    for i in range(2,n):
        if n % i == 0:
            return(False)
    return(True)
```

In [2]:
```
def primesupto(n):
    primelist = []
    for j in range(2,n+1):
        if isprime(j):
            primelist.append(j)
    return(primelist)
```

In [3]:
```
primesupto(30)
```

Out[3]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

First n primes

What if we want a list of the first n primes?

- Generate numbers 2,3,... and check if each one is a prime
- Stop when we have generated n primes

We don't know the upper bound of the list 2,3,...

- Can't use `range()`

Instead, a new kind of loop

- "Manually" generate the sequence
- Stop when we reach the terminating condition

```
while (condition):
  statement 1
  ...
  statement k
```

- If `condition` evaluates to `True` the block of k statements is executed
- After this, the condition is checked again and the same process is repeated
- Compare to `if` where the condition is evaluated once

```
if (condition):
  statement 1
  ...
  statement k
```

In [4]:
```
def nprimes(n):
    primelist = []
    i = 2
    while (len(primelist) < n):
```

```python
        if (isprime(i)):
            primelist.append(i)
        i = i+1
    return(primelist)
```

In [5]:
```python
nprimes(20)
```

Out[5]: `[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]`

Infinite loops

- Need to ensure that the statements make *progress* towards falsifying the condition
- If the condition remains `True` forever, the loop never terminates
- For instance, suppose there were only finitely many primes, say M. For any $n > M$, the length of `primelist` would saturate at M so the condition `len(primelist) < n` would never become `False`

Looping --- `for` and `while`

- `while` is more general than `for`

- Can implement

  ```
  for x in l:
    ...
  ```

  using `while` by explicitly going through `l` from first to last position

  ```
  pos = 0
  while (pos < len(l)):
    ...
    pos = pos + 1
  ```

- Note that we have to move the position "manually" to ensure that we make progress towards termination
- However, using `for` is preferred if it is clearly an iteration over a fixed sequence
  - The intent is capture much more clearly
  - In the `while` form it is slightly obfuscated

Efficiency

- To check if n is a prime, we can stop testing factors at $\sqrt{n}$, rather than checking all numbers in `range(2,n)`
- Another possibility is to only check *prime factors* smaller than n
  - This is the principle behind the Sieve of Eratosthenes
- We modify `isprime` to take a list of primes and only check for factors from this list

In [6]:
```python
def sieve(plist,n):
    for p in plist:
        if n % p == 0:
            return(False)
    return(True)
```

- Note that the function does not check that `plist` contains all primes below `n`

- In fact, it does not even check that elements of `plist` are actually primes
- We have to call it with an appropriate list of primes for it to work correctly

In [7]:
```
def primesuptosieve(n):
    primelist = []
    for j in range(2,n+1):  # INVARIANT: primelist is primes upto j-1
        if sieve(primelist,j):
            primelist.append(j)
    return(primelist)
```

- Each time we update the value of `j` in `for`, we have all primes less than `j` in `primelist`
- This is called an *invariant* of the loop, or *loop invariant*
- This invariant guarantees that the call to `sieve` passes the correct list of primes as the first argument

In [8]:
```
primesuptosieve(70)
```

Out[8]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]
```

In [9]:
```
primesupto(70)
```

Out[9]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]
```

Checking efficiency

- How can we validate whether the new function is more efficient?
- Empirically, one can see how long it takes to run on large inputs
- To avoid generating large lists in the output, we modify the function to report the largest prime below n, rather than the list of all primes

In [10]:
```
def primesupto(n):
    primelist = []
    for j in range(2,n+1):
        if isprime(j):
            primelist.append(j)
            lastprime = j
    return(lastprime)
```

In [11]:
```
def primesuptosieve(n):
    primelist = []
    for j in range(2,n+1):  # INVARIANT: primelist is primes upto j-1
        if sieve(primelist,j):
            primelist.append(j)
            lastprime = j
    return(lastprime)
```

- For $n = 1000$, both functions report the answer "instantaneously"

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

Out[12]:
```
997
```

In [13]:
```
primesuptosieve(1000)
```

Out[13]:
```
997
```

- For $n = 10000$, we see a small delay in the execution of `primesupto`, whereas `primesuptosieve` is still "instantaneous"

```
In [14]: primesupto(10000)
```

```
Out[14]: 9973
```

```
In [15]: primesuptosieve(10000)
```

```
Out[15]: 9973
```

- For $n = 100000$, the difference between the two functions is more clear

```
In [16]: primesupto(100000)
```

```
Out[16]: 99991
```

```
In [17]: primesuptosieve(100000)
```

```
Out[17]: 99991
```

- Later we will see how to measure and report execution time
- We will also understand why the behaviour for $n = 1000$, $n = 10000$ and $n = 100000$ is as observed

Which is better?

- Which do we expect to be faster --- checking all factors upto $\sqrt{n}$ or all primes below n
- If we check factors upto $\sqrt{n}$ we check $\sqrt{n} = \frac{n}{\sqrt{n}}$ numbers
- How many primes are there below n? The Prime Number Theorem tells us this is of the order $\frac{n}{\log n}$, so we check this many numbers
- Since $\log n$ is much smaller than $\sqrt{n}$, the sieve method is actually less efficient than checking all factors upto $\sqrt{n}$

An improved solution

- Combine the two ideas
- Check all prime factors upto $\sqrt{n}$
  - We pass a list of all primes below n, as before
  - When iterating through this list, if we cross $\sqrt{n}$ without finding a factor, we declare the number is a prime
  - Note that the final `return(True)` is probably never executed

```
In [18]: def sievesqrt(plist,n):
             import math
             for p in plist:
                 if p > math.sqrt(n):
                     return(True) # No prime factors below sqrt(n)
                 if n % p == 0:
                     return(False)
             return(True)  # Does this statement ever execute?
```

- Modify our search for primes to call `sievesqrt` rather than `sieve`

In [19]:
```
def primesuptosievesqrt(n):
    primelist = []
    for j in range(2,n+1):  # INVARIANT: primelist is primes upto j-1
        if sievesqrt(primelist,j):
            primelist.append(j)
            lastprime = j
    return(lastprime)
```

- Compare times for large n, say $n = 200000$

In [20]:
```
primesuptosievesqrt(200000)
```

Out[20]: 199999

In [21]:
```
primesuptosieve(200000)
```

Out[21]: 199999

In [22]:
```
primesupto(200000)  # Warning, takes a long time!
```

Out[22]: 199999

Boolean datatypes

- Usually an outcome of comparisons: `==` , `!=` , `<` , `<=` , `>` , `>=`
- Useful shortcut
  - Any "empty" value is interpreted as `False`
  - So `0` , `[]` , `""` (empty string) are all `False`
  - Any other value is interpreted as `True`
- Avoid comparisons such as `if x == 0` or `if l != []`
  - Write `if not(x)` , `if l` instead

In [23]:
```
l = [1,2,3]
if l:
    x = True
else:
    x = False
```

In [24]:
```
x
```

Out[24]: True

In [25]:
```
m = 0
if not(m):
    y = True
else:
    y = False
```

In [26]:
```
y
```

Out[26]: True

- Note that Python does not insist on brackets around the condition in `if` and `while`
  - Can write `if (cond):` or `if cond:` , `while (cond):` or `while cond:`

Variables, values and types

- Variables (names) have no intrinsic types
- Values have types
  - A variable inherits the type of the value it currently holds
- The type of value a variable holds can vary over time
  - But not a good idea to use the same name for different types of values in the same piece of code
  - Reduces readability, maintainability
- The `type()` function returns the type of a variable that is currently assigned a value

In [27]:
```
x = True
```

In [28]:
```
type(x)
```

Out[28]:
```
bool
```

In [29]:
```
x = 5
```

In [30]:
```
type(x)
```

Out[30]:
```
int
```

- The function `del()` unassigns a value from a name

In [31]:
```
del(x)
```

In [32]:
```
type(x)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[32], line 1
----> 1 type(x)

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

Conditional statement

`if` allows conditional execution

```
if condition:
    statement 1
    ...
    statement k
else:
    statement 1'
    ...
    statement k'
```

- If `condition` evaluates to `True`, the first block is executed, otherwise the second block.
- The `else:` block is optional. If there is no `else:` block and the `condition` evaluates to `False`, execution skips over to the next statement after the `if`

- Example: Compute the absolute value of a number

In [33]:
```python
def myabs(x):  # myabs to avoid any confusion with built-in abs()
    if x < 0:
        return(-x)
    else:
        return(x)
```

In [34]:
```python
myabs(-9), myabs(7)
```

Out[34]: (9, 7)

Multiway branching --- `elif`

Suppose we want to compute $sign(x) = \begin{cases} x < 0 & = & -1, \\ x = 0 & = & 0, \\ x > 0 & = & 1 \end{cases}$

In Python, we would have to nest `if` statements like this:

```
if x < 0:
    return(-1)
else:
    if x == 0:
        return(0):
    else:
        return(1)
```

- As we see, the indentation of the nested `if` pushes the code to the right
- With more cases, this would become worse
- Python provides `elif` to avoid this cascaded nesting

```
if x < 0:
    return(-1)
elif x == 0:
    return(0):
else:
    return(1)
```

- Can have as many `elif` blocks as you need
- `else` is still optional

In [35]:
```python
def sign(x):
    if x < 0:
        return(-1)
    elif x == 0:
        return(0)
    else:
        return(1)
```

In [36]:
```python
sign(-7)
```

Out[36]: -1

In [37]:
```python
sign(8)
```

Out[37]: 1

In [38]:
```python
sign(0)
```

Out[38]: 0