PDSP 2025, Lecture 04, 19 August 2025

Checking if a number is prime

- Checking if `n` is a prime: assume it is, and flag that is not if we find a factor between `2` and `sqrt(n)`

In [1]:
```
import math
```

In [2]:
```
n = 25
isprime = True
for i in range(2,int(math.sqrt(n))+1):  # int(...) truncates a float to a
    if n % i == 0:
        isprime = False
```

In [3]:
```
isprime
```

Out[3]:
```
False
```

Computing primes upto `n`

- Instead of checking if `n` is a prime, find all primes upto (and including) `n`
- Generate the sequence `2,3,...,n`
- For each element in this sequence, check if it is a prime
- Accumulate all primes found in a list
  - Recall that `l1 + l2` concatenates two lists into a single list
- Two *nested* loops, use different variables `j` and `i` to iterate

In [4]:
```
n = 100
primelist = []
for j in range(2,n+1):
    isprime = True
    for i in range(2,j):
        if j % i == 0:
            isprime = False
    if isprime:
        primelist = primelist + [j]
```

In [5]:
```
primelist
```

Out[5]:
```
[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]
```

Appending a value to a list

- Can also use `l.append(v)` to add an element `v` to a list
- Note the distinction between `l + [v]` and `l.append(v)`
  - In the first case, we have to make `v` into a singleton list `[v]` to use the operator `+`

In [6]:
```
n = 100
primelist = []
for j in range(2,n+1):
    isprime = True
    for i in range(2,j):
        if j % i == 0:
            isprime = False
    if isprime:
        primelist.append(j)
```

In [7]:
```
primelist
```

Out[7]:
```
[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]
```

Functions

- Modularise code into functional units
- Instead of embedding code to check if `j` is a prime, call a function that returns `True` if `j` is a prime and `False` otherwise
- Function definition starts with `def function_name (argument1, argument2, ...):`
- When the function completes, it should report an answer -- return a value through `return(v)`

In [8]:
```
def isprime(n):
    status = True
    for i in range(2,n):
        if n % i == 0:
            status = False
    return(status)
```

In [9]:
```
isprime(17), isprime(25)
```

Out[9]:
```
(True, False)
```

Exiting a function in between

- If we find a factor, we can declare the number to not be a prime without testing more factors
- In the original implementation, we needed to exit the loop
- `return()` automatically exits, so we can use this optimisation in the function

In [10]:
```
def isprime2(n): # An equivalent defn, terminates with False at first fac
    status = True
    for i in range(2,n):
        if n % i == 0:
            status = False
            return(status)
    return(status)
```

In [11]:
```
isprime2(47), isprime2(44)
```

Out[11]:
```
(True, False)
```

- In fact, we don't even need the variable `status`
- If we find a factor, `return(False)`
- If the search for a factor ends without finding one, `return(True)`

In [12]:
```
def isprime3(n):    # An equivalent defn, without a separate status varia
    for i in range(2,n):
        if n % i == 0:
            return(False)
    return(True)
```

In [13]:
```
isprime3(571), isprime3(573)
```

Out[13]:
```
(True, False)
```

Using functions

- We can rewrite our code to search for primes upto `n` to call the function `isprime` for each candidate
  - Recall that in our earlier, explicit, code, we had to rename the outer loop variable as `j` to avoid a clash with the loop through potential factors
  - If we use a function, the `i` inside the function is different from the `i` outside the function

In [14]:
```
n = 100
primelist = []
for i in range(2,n+1):
    if isprime(i):
        primelist.append(i)
```

In [15]:
```
primelist
```

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

- We can convert this search for primes upto `n` into another function

In [16]:
```
def primesupto(n):
    primelist = []
    for i in range(2,n+1):
        if isprime(i):
            primelist.append(i)
    return(primelist)
```

In [17]:
```
primesupto(30)
```

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

In [18]:
```
primesupto(70)
```

Out[18]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]
```

In [19]:
```
primesupto(1000)
```

Out[19]: [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,
211,
223,
227,
229,
233,
239,
241,
251,
257,
263,
269,
271,
277,
281,

283,
293,
307,
311,
313,
317,
331,
337,
347,
349,
353,
359,
367,
373,
379,
383,
389,
397,
401,
409,
419,
421,
431,
433,
439,
443,
449,
457,
461,
463,
467,
479,
487,
491,
499,
503,
509,
521,
523,
541,
547,
557,
563,
569,
571,
577,
587,
593,
599,
601,
607,
613,
617,
619,
631,
641,
643,
647,
653,
659,

```
661,
673,
677,
683,
691,
701,
709,
719,
727,
733,
739,
743,
751,
757,
761,
769,
773,
787,
797,
809,
811,
821,
823,
827,
829,
839,
853,
857,
859,
863,
877,
881,
883,
887,
907,
911,
919,
929,
937,
941,
947,
953,
967,
971,
977,
983,
991,
997]
```

Functions and modularity

- Functions modularise code
- Each function has an *interface contract* -- if the input x is valid, the output is $f(x)$
- Can change the implementation of the function so long as the interface contract is upheld
    - Any one of our three implmentations of `isprime` can be used

- For instance, can use a naive implementation as a *prototype* and later replace by a more refined, optimised implementation

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 [20]:
```
def nprimes(n):
    primelist = []
    i = 2
    while (len(primelist) < n):
        if (isprime(i)):
            primelist.append(i)
        i = i+1
    return(primelist)
```

In [21]:
```
nprimes(20)
```

Out[21]:
```
[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

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 [22]:
```
l = [1,2,3]
if l:
    x = True
else:
    x = False
```

In [23]:
```
x
```

Out[23]:
```
True
```

In [24]:
```
m = 0
if not(m):
    y = True
else:
    y = False
```

In [25]:
```
y
```

Out[25]:
```
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 [26]:
```
x = True
```

In [27]:
```
type(x)
```

Out[27]:
```
bool
```

In [28]:
```
x = 5
```

In [29]:
```
type(x)
```

Out[29]:
```
int
```

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

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

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

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Cell In[31], line 1
----> 1 type(x)

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