PDSP 2024, Lecture 03, 13 August 2024

Generating sequences of numbers

- Generate numbers 2 to n-1
- `range(n)` generates the sequence `0, 1, 2, ..., n-1`
- Use `list(range(n))` to display as a list

In [1]:
```
n = 17
```

In [2]:
```
range(n)  # Like a list, but not quite
```

Out[2]:
```
range(0, 17)
```

In [3]:
```
list(range(n))  # Make it into a list
```

Out[3]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

- `range(n)` translates to `range(0,n)`, implicitly starting with `0`
- Can add an explicit starting point: `range(i,n)` generates `i,i+1,...,n-1`

In [4]:
```
list(range(2,n))
```

Out[4]:
```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

- If the starting point is $\geq$ the target, `range` generates an empty sequence

In [5]:
```
list(range(3,3))
```

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

In [6]:
```
list(range(7,4))
```

Out[6]:
```
[]
```

Numbers in Python

- Numbers in Python can be integers (`int`) or reals -- actually rationals -- (`float`)
- Internal representation is different, but arithmetic operation symbols are *overloaded* to apply to both types of numbers
- `+`, `-`, `*` stand for addition, subtraction, multiplication, as usual
- `/` is division, and always produces a `float`

In [7]:
```
8/4
```

Out[7]:
```
2.0
```

- There are separate operators for *quotient* (`//`) and *remainder* (`%`)
    - These can also be applied to `float` arguments, but the answer is also `float`

In [8]:
```
8//4
```

Out[8]: 2

In [9]:
```
7 % 3
```

Out[9]: 1

In [10]:
```
8.0//3.0, 8.0 % 5.0
```

Out[10]: (2.0, 3.0)

Data types

- A data type is a set of values with associated operations
- Python has two numeric data types, `int` and `float`
- In the IPL example, we saw text data, which is of type `String` -- we shall examine this later
- The boolean data type has two values `True` and `False`

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 `n-1`

In [11]:
```
n = 17
isprime = True
for i in range(2,n):
    if n % i == 0:
        isprime = False
```

In [12]:
```
n, isprime
```

Out[12]: (17, True)

In [13]:
```
n = 18
isprime = True
for i in range(2,n):
    if n % i == 0:
        isprime = False
```

In [14]:
```
n, isprime
```

Out[14]: (18, False)

Optimising the search for factors

- Factors occur in pairs, sufficient to check from 2 to $\sqrt{n}$
- Python has a function `sqrt` to compute square roots
- However it is not automatically available

In [15]:
```
sqrt(n)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[15], line 1
----> 1 sqrt(n)

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

Libraries

- Libraries are collections of code implementing different groups of functions relevant to a given theme
- We will later see libraries specific to data science, machine learning
- The `math` library has mathematical functions like `sqrt`, `log`, `sin`, `cos` etc
- We `import` the `math` library to use it
  - Note that we use `math.sqrt` to tell Python the full context of the function `sqrt`
  - This is useful in case two different libraries have different functions with the same name

In [16]:
```
import math
```

In [17]:
```
n = 17
math.sqrt(n)
```

Out[17]: 4.123105625617661

Optimised primality checking

- We can optimize our search for factors by restricting the range to `(2,math.sqrt(n))`
- `range` expects only `int` arguments, so use `int()` to convert `math.sqrt(n)` to an `int` -- truncates the fractional part

In [18]:
```
n = 17
isprime = True
for i in range(2,int(math.sqrt(n))):  # int(...) truncates a float to an int
    if n % i == 0:
        isprime = False
```

In [19]:
```
isprime
```

Out[19]: True

- We have to be careful, because `range(j,m)` stops at `m-1`
- The code above wrongly claims `25` is a prime -- the search for factors runs from `2` to `4` rather than `2` to `5`

In [20]:
```
n = 25
isprime = True
for i in range(2,int(math.sqrt(n))):  # int(...) truncates a float to an int
    if n % i == 0:
        isprime = False
```

In [21]:
```
isprime
```

Out[21]: True

- To fix this, modify the upper bound of `range` to `sqrt(n)+1`

In [22]:
```
n = 25
isprime = True
for i in range(2,int(math.sqrt(n))+1):  # int(...) truncates a float to an int
    if n % i == 0:
        isprime = False
```

In [23]:
```
isprime
```

Out[23]: False

Large and small numbers

- Python allows us to work with very large (and very small numbers)
- The operatoer `**` is exponentiation

In [24]:
```
7**2, 2**10
```

Out[24]:
```
(49, 1024)
```

- What is $2^{2^{10}}$, in other words, 2^{1024}?

In [25]:
```
2**(2**10)
```

Out[25]:
```
1797693134862315907729305190789024733617976978942306572734300811577326758055009631
3270847732240753602112011387987139335765878976881441662249284743063947412437776789
3424865485276302219601246094119453082952085005768838150682342462881473913110540827
237163350510684586298239947245938479716304835356329624224137216
```

- How about 2^{-1024}?

In [26]:
```
2**(-(2**10))
```

Out[26]:
```
5.562684646268003e-309
```

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 [27]:
```
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 + [i]
```

In [28]:
```
primelist
```

Out[28]:
```
[5,
 2,
 4,
 6,
 10,
 12,
 16,
 18,
 22,
 28,
 30,
 36,
 40,
 42,
 46,
 52,
 58,
 60,
 66,
 70,
 72,
 78,
 82,
 88,
 96]
```

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 [29]:
```
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 [30]:
```
primelist
```

Out[30]:
```
[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 [31]:
```
def isprime(n):
    status = True
    for i in range(2,n):
        if n % i == 0:
            status = False
    return(status)
```

In [32]:
```
isprime(17), isprime(25)
```

Out[32]:
```
(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 [33]:
```
def isprime2(n): # An equivalent defn, terminates with False at first factor
    status = True
    for i in range(2,n):
        if n % i == 0:
            status = False
            return(status)
    return(status)
```

In [34]:
```
isprime2(47), isprime2(44)
```

Out[34]: (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 [35]:
```
def isprime3(n):    # An equivalent defn, without a separate status variable
    for i in range(2,n):
        if n % i == 0:
            return(False)
    return(True)
```

In [36]:
```
isprime3(571), isprime3(573)
```

Out[36]: (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 [37]:
```
n = 100
primelist = []
for i in range(2,n+1):
    if isprime(i):
        primelist.append(i)
```

In [38]:
```
primelist
```

Out[38]:
```
[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 [39]:
```
def primesupto(n):
    primelist = []
    for i in range(2,n+1):
        if isprime(i):
            primelist.append(i)
    return(primelist)
```

In [40]:
```
primesupto(30)
```

Out[40]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

In [41]:
```
primesupto(70)
```

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

In [42]:
```
primesupto(1000)
```

```
Out[42]: [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