PDSP 2026, Lecture 04, 20 August 2026

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

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

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

In [3]:
```
def primesupto(n):
    primelist = []
    for i in range(2,n+1):
        if isprime(i):
            primelist.append(i)
    return(primelist)
```

In [4]:
```
primesupto(30)
```

Out[4]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

In [5]:
```
primesupto(70)
```

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

In [6]:
```
primesupto(1000)
```

Out[6]: [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]
```

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 the `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 [7]:
```
def nprimes(n):
    primelist = []
    i = 2
    while (len(primelist) < n):
        if (isprime(i)):
            primelist.append(i)
        i = i+1
    return(primelist)
```

In [8]:
```
nprimes(20)
```

Out[8]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
```

In [9]:
```
nprimes(100)
```

Out[9]: [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]
```

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 [10]:
```
l = [1,2,3]
if l: # if l != []
    x = True
else:
    x = False
```

In [11]:
```
x
```

Out[11]:
```
True
```

In [12]:
```
m = 0
if not(m):
    y = True
else:
    y = False
```

In [13]:
```
y
```

Out[13]:
```
True
```

- Be careful. You don't always get what you expect!

In [14]:
```
7 or 5
```

Out[14]:
```
7
```

- 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 [15]:
```
x = True
```

In [16]:
```
type(x)
```

Out[16]:
```
bool
```

In [17]:
```
x = 5
```

In [18]:
```
type(x)
```

Out[18]:
```
int
```

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

In [19]:
```
del(x)
```

In [20]:
```
type(x)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[20], 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 [21]:
```
def myabs(x):  # myabs to avoid any confusion with built-in abs()
    if x < 0:
        return(-x)
    else:
        return(x)
```

In [22]:
```
myabs(-9), myabs(7)
```

Out[22]:
```
(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 [23]:
```
def sign(x):
    if x < 0:
        return(-1)
    elif x == 0:
        return(0)
    else:
        return(1)
```

In [24]:
```
sign(-7)
```

Out[24]:
```
-1
```

In [25]:
```
sign(8)
```

Out[25]:
```
1
```

In [26]:
```
sign(0)
```

Out[26]:
```
0
```

Lists

- Sequences of values, indexed by position
- For a list with `n` values, valid positions are `0` to `n-1`
  - `len(l)` gives the length of a list
- Accessing a position beyond `len(l)-1` results in `IndexError`

In [27]:
```
l = list(range(20,40))
```

In [28]:
```
len(l), l[3], l[19]
```

Out[28]:
```
(20, 23, 39)
```

In [29]:
```
l[20]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[29], line 1
----> 1 l[20]

IndexError: list index out of range
```

- What about indices below `0`?
- Index `-j` is interpreted as `len(l)-j`
  - Useful for accessing values from the end of the list
  - Valid indices in reverse are `-1`, `-2`, ..., `-len(l)`

In [30]:
```
l[-1], l[-20]
```

Out[30]:
```
(39, 20)
```

Slices

- Recall that `nprimes(n)` computed the first `n` primes

In [31]:
```
def isprime(n):
    for j in range(2,n):
        if n % j == 0:
            return(False)
    return(True)

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

In [32]:
```
first20primes = nprimes(20)
```

In [33]:
```
first20primes
```

Out[33]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
```

- What are the primes from 11 to 15?
- Need a sublist of the original list
- `l[i:j]` is the list `[l[i], l[i+1], ..., l[j-1]]`
- Similar to

```
newl = []
for k in range(i,j):
   newl.append(k)
```

In [34]:
```
first20primes[11:16]
```

Out[34]:
```
[37, 41, 43, 47, 53]
```

- like `range()` if the indices don't make sense, you get an empty list

In [35]:
```
first20primes[11:10]
```

Out[35]:
```
[]
```

- Unlike accessing `l[i]`, can give upper bound beyond the list
- `l[i:len(l)+10]` is interpreted as `l[i:len(l)]`

In [36]:
```
first20primes[11:40]
```

Out[36]:
```
[37, 41, 43, 47, 53, 59, 61, 67, 71]
```

- Can omit the upper bound, defaults to `len(l)`

In [37]:
```
first20primes[15:]
```

Out[37]:
```
[53, 59, 61, 67, 71]
```

- Likewise, omit the lower bound, defaults to `0`

In [38]:
```
first20primes[:10]
```

Out[38]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

- Omit both lower and upper bound to get a `full slice`
  - Full slice returns a new list that is a copy of the list
  - Significance will become clearer later

In [39]:
```
first20primes[:]
```

Out[39]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
```

More about `range()`

- `range(i,j)` generates the sequence `i,i+1,...,j-1`
- `range(n)` generates the sequence `0,1,...,n-1` -- implicitly starts with `0`
- What if we want to skip over some numbers
  - All even numbers from `4` to `40`
- Optional third argument is the step size
  - `range(i,j,k)` is `i,i+k,...,i+mk` for the largest `m` such that `i+mk < j` and `i+(m+1)k >= j`

In [40]:
```
list(range(4,41,2)) # Even numbers from 4 to 40
```

Out[40]:
```
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40]
```

In [41]:
```
list(range(4,40,2)) # Even numbers from 4 to 38
```

Out[41]:
```
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
```

- Can also count down -- give a negative step!

In [42]:
```
list(range(10,0,-1))
```

Out[42]:
```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

In [43]:
```
list(range(10,-1,-2))
```

Out[43]:
```
[10, 8, 6, 4, 2, 0]
```

- `range(i,j,k)` generate `i, i+k,...` so that the sequence does not *cross* `j`
- Depending on whether it is increasing or decreasing, the last value will be less than `j` or greater than `j`

Stepped slices

- Can similarly give a third argument in a slice

In [44]:
```
first20primes[2:20:3]
```

Out[44]:
```
[5, 13, 23, 37, 47, 61]
```

In [45]:
```
first20primes[19:0:-1]
```

Out[45]:
```
[71, 67, 61, 59, 53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5, 3]
```

- Explain the following output. (Hint, what is `l[-1]`?)

In [46]:
```
first20primes[19:-1:-1]
```

Out[46]:
```
[]
```

- Can omit upper and lower bounds but give a step
- `l[::-1]` is the entire list in reverse
  - Note that the default lower and upper bound are determined by the step

In [47]:
```
first20primes[::-1]
```

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