# Programming and Data Structures with Python

# Lecture 2, 17 December 2020

## Computing gcd: recap

Naive, brute-force algorthm

- Generate lists **fm** and **fn** of factors of $m$ and $n$ by scanning $i$ from $1$ to $m$ and $1$ to $n$, respectively. Compute list of common factors **cf** from **fm** and **fn**. Report largest (right-most) value in **cf**

Refinements

- Sufficient to scan candidate factors from $1$ to $\min(m,n)$
- Overlap the computation of **fm** and **fn** in a single scan
- In a single scan of $1$ to $\min(m,n)$, directly compute **cf**

## Lists, revisited

- Do we need to maintain lists of factors?
- Once we replace $i$ in **cf** by a larger value $j$, we don't need $i$ any more
  - Sufficient to record *most recent common factor*, **mrcf**

In [1]:

```
def gcd4(m,n):
    for i in range(1,min(m,n)+1):
        if (m%i) == 0 and (n%i) == 0:
            mrcf = i
    return(mrcf)
```

In [2]:

```
gcd4(1001,52)
```

Out[2]:

```
13
```

## Reversing the scan

We are interested in largest common factor. Instead of scanning from $1$ to $\min(m, n)$, scan in reverse from $\min(m, n)$ down to $1$. Stop as soon as we find any common factor.

This introduces a new type of loop. Previously **for** ran through a fixed set of values in a list/sequence. Here, instead, we have a **while** loop, governed by a condition. So long as the condition associated with the **while** is true, the loop repeats. Once the condition fails, the loop ends.

In [3]:

```
def gcd5(m,n):
    i = min(m,n)
    while i > 0:
        if (m%i) == 0 and (n%i) == 0:
            return(i)
        else:
            i = i-1
```

## Using basic properties of numbers

- Suppose $d$ is a common factor of $m$ and $n$
- Then we can write $m = ad$ and $n = bd$
- Assuming $m > n$, $m - n = (a - b)d$, so $d$ divides $m - n$

New strategy

- Assume $m > n$. If $n$ divides $m$, then $\gcd(m, n) = n$
- Otherwise, solve a smaller instance of the problem $\gcd(n, m - n)$
- Note that it could be that $m - n > n$

In [4]:

```
def gcd6(m,n):
    # Assume m >= n
    if m < n:
        (m,n) = (n,m)
    if (m%n) == 0:
        return(n)
    else:
        diff = m-n
        return(gcd6(n,diff))
```

## Recursion

This is a *recursive* computation. We compute $\gcd(m, n)$ in terms of smaller arguments $\gcd(x, y)$. When we reach the base case ($n$ divides $m$) we get the answer as $n$.

Here is an example:

$$\gcd(77, 33) \rightsquigarrow \gcd(44, 33) \rightsquigarrow \gcd(33, 11) \rightsquigarrow 11$$

## Converting recursion to iteration

We can also write an *iterative* version of the same algorithm, using a loop to repeatedly replace $(m, n)$ by $(\max(n, \text{diff}), \min(n, \text{diff}))$. Note that we force the pair to be such that the first element is bigger than the second.

In [5]:

```
def gcd7(m,n):
    if m < n: # Assume m >= n
        (m,n) = (n,m)
    while (m%n) != 0:
        diff = m-n
        # diff > n? Possible!
        (m,n) = (max(n,diff),min(n,diff))
    return(n)
```

## Efficiency

How long does this take? Consider $\gcd(x, 2)$ where $x$ is a large odd number. We will compute $\gcd(x, 2) \rightsquigarrow \gcd(x - 2, 2) \rightsquigarrow \cdots \rightsquigarrow \gcd(5, 2) \rightsquigarrow \gcd(3, 2) \rightsquigarrow \gcd(2, 1) \rightsquigarrow 1$.

This takes $x/2$ steps, so the length of the computation is roughly the *magnitude* of the argument.

For arithmetic calculation, we would like operations to grow with the number of digits rather than the magnitude. For instance, a 5-digit number is 100 times as large as a 3-digit number, but adding two 5-digit numbers does not take 100 times more effort than adding two 3-digit numbers. In fact, there are 2 extra columns to add because the number of digits has grown by 2.

The number of digits in $n$ is proportionate to $\log_{10}(n)$. (For any base $b$, the number of digits in a base $b$ representation of $n$ is proportionate $\log_b(n)$.)

This prompts us to our final refinement of the gcd algorithm, that goes back to Euclid.

## Euclid's algorithm

We saw that any common divisor $d$ of $m$ and $m$ must also divide $m-n$. Hence, if $n$ does not divide $m$, we replace $\gcd(m,n)$ by $\gcd(m-n,n)$.

Suppose $n$ does not divide $m$. Then, $m = qn + r$, where $q$ is the *quotient* and $r < n$ is the *remainder.

Now, supposed $d$ divides both $m$ and $n$. As before, we can write $m = ad$ and $n = bd$.

From $m = qn + r$ we get $ad = q(bd) + r$, so $r = (a - qb)d$. In other words, $d$ must divide $r$ as well.

Hence, instead of reducing $\gcd(m,n)$ to $\gcd(m-n,n), we can reduce it to$\gcd(n,r). Notice that $r < n$ because it is the remainder when we divide $m$ by $n$.

In [6]:

```
def gcd8(m,n):
    if m < n: # Assume m >= n
        (m,n) = (n,m)
    if (m%n) == 0:
        return(n)
    else:
        r = m%n
        return(gcd8(n,r))  # m%n < n, always!
```

# Python syntax

## Names and values

In [7]:

```
# This is a comment --- an explanation that is not executed

# Assigning a value to a name
x = 7
```

Can query Python interactively, use as a calculator.

In [8]:

```
x
```

Out[8]:

```
7
```

In [9]:

```
x + 8
```

Out[9]:

```
15
```

In [10]:

```
y = x + 8
```

In [11]:

```
y
```

Out[11]:

```
15
```

Ensure that a name has an associated value before it is used. Using **z** on the right hand side below generates an error.

In [12]:

```
y = z + 9
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most
 recent call last)
<ipython-input-12-77221b398280> in <module>
----> 1 y = z + 9

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

Values have "types" -- numbers, lists, .... : "data type"

- Type defines what operations are allowed on the value
- Numbers allow arithmetic: +, -, *, /
- List: append values, find the value at position i etc

Python types

- Numbers: natural, integer, real, complex, ....
- In Python, essentially two varieties of numbers: integers, reals
  - Historically, these are represented differently inside the computer
- Reals have a decimal point, integers do not
- Usually, arithmetic preserves types

In [13]:

```
x = 7.0
```

In [14]:

```
type(x)
```

Out[14]:

```
float
```

Why float???

- "Floating" decimal point -- $6.02 \times 10^{-23}$
- Integers have a "fixed" decimal point at the right hand end of the number

In [15]:

```
# Operations preserve value
x = 7
a = x + 3
b = x - 4
c = x * 8
d = x / 7
e = x // 6  # Quotient
r = x % 6 # Remainder
```

In [16]:

```
(a,type(a), b, type(b),c, type(c), d,type(d), e, type(e), r, type(r))
```

Out[16]:

```
(10, int, 3, int, 56, int, 1.0, float, 1, int, 1, int)
```

Size limitation of integers -- none!

In [17]:

```
8989878869629444444444444444444 * 7879879796654355431111111111111111
```

Out[17]:

```
7083916487916295334785526558191254721391753533698234781895
0617283950617284
```

In [18]:

```
y = 7.5
z = 8.9
w = y/z
q = z//y
r = z%y
(w, type(w), q, type(q), r, type(r))
```

Out[18]:

```
(0.8426966292134831, float, 1.0, float, 1.400000000000000
4, float)
```

Binary and decimal representations of fractions have different behaviour

- 1/10 is an infinite recurring fraction in binary
- 1/3 and 2/3 are not finite decimal fractions. 0.333333... . 0.66666... = 0.9999999...