Programming and Data Structures with Python

Lecture 1, 14 December 2020

What is programming?

Writing systematic procedures in precise notation

- Systematic procedure: *algorithm*
- Precise notation: *programming language*

Example: Prepare a classroom for a seminar by a guest speaker

- Various things to be done: arrange chairs, check projector/screen, check audio system, turn on a/c early, ...
- Need to instruct support staff to do this task. Nature of instructions varies according to who is doing the job.
  - Outsouce to professionals: Know the process, just provide the time of the talk and the expected audience size.
  - Somewhat experienced staff: Provide a high-level checklist, but each step need not be described explicitly
  - Inexperienced staff: Each high-level step needs detailed instructions
    - Arranging chairs: arrange m rows of chairs, k chairs per row, leave an aisle in between to walk to the back, ...)
- In all cases, instructions are in terms of *basic steps*
  - Basic steps can executed without further clarification
  - Granularity of the explanation varies according to the nature of basic steps

Greatest Common Divisor

$\gcd(m, n)$: largest d that divides both m and n

- Also called hcf, *highest common factor*
- 1 divides any number, so there is at least one such d, always

How can we systematically compute $\gcd(m, n)$?

First look for a systematic procedure

- "*Brute force*" - need not be clever or efficient, but must always be correct

From the definition fo $\gcd(m, n)$:

- Generate the set of factors/divisors of m
- Generate the set of factors/divisors of n
- Compare these sets and find the largest element common to both sets

Computing the factors of m

- Smallest factor of m is 1, largest factor is m
- Search for factors in the range 1,2,...,m
  - For each number is in this range, check that i divides m
- Factors of 14
  - 1, 2, ~~3~~, ~~4~~, ~~5~~, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, ~~11~~, ~~12~~, ~~13~~, 14
  - $\{1, 2, 7, 14\}$, set of factors of 14
  - $[1, 2, 7, 14]$, list of factors of 14
- Sets vs lists
  - In a list, order matters
    - $[1, 2, 7, 14] \neq [14, 7, 2, 1]$, whereas $\{1, 2, 7, 14\} = \{14, 7, 2, 1\}$
  - In a list, duplicates matter
    - $[1, 2, 7, 14, 14] \neq [1, 2, 7, 14]$, whereas $\{1, 2, 7, 14\} = \{1, 2, 7, 14, 14\}$
- Compute $\gcd(14, 63)$
  - Factors of 14 are $[1, 2, 7, 14]$
  - Factors of 63 are $[1, 3, 7, 9, 21, 63]$
  - Find largest element common to both lists
    - For each factor of 14, check if it is also in the list of factors 63
    - Check 1 (yes), 2 (no), 7 (yes), 14 (no)
  - Common factors are $[1, 7]$
  - Largest of these is 7, so $\gcd(14, 63) = 7$

Using names to store intermediate values

In a maths book you might see an equation labelled (*) or (3.7) so that later in the text the author can refer to equation (*) or equation (3.7).

Likewise, we need to refer to intermediate values in our procedure

- Generate the list of factors of m
- Generate the list of factors of n
- Compare the **first** list with the **second** list

- Generate a **third** list of common factors of m and n

As the number of items to keep track of increases, we need to name them systematically

- **fm**, the list of factors of m
- **fn**, the list of factors of n
- **cf**, the list of common factors of m and n

Often, names are also referred to as *variables*

- Different from variables in mathematics
- In mathematics, a variable is typically an unknown, but fixed quantity
  - I bought some apples and gave 3 to my sister. I now have 2 apples left. How many did I buy initially?
  - Let x be the number of apples I bought. I know that $x - 5 = 3$. So $x = 2$.
  - The value of x is not known till you resolve the constraints, but the value of x does not change
- In programming, intermediate values are periodically updated
  - Initially, **fm** is the empty list []
  - Each factor i we discover is appended to **fm**
  - Even i is a name, for the potential factor we are checking!

Our first look at Python

Here is Python code for the procedure we have described

In [1]:

```
def gcd1(m,n):
    fm = []
    for i in range(1,m+1):
        if (m%i) == 0:
            fm.append(i)
    fn = []
    for j in range(1,n+1):
        if (n%j) == 0:
            fn.append(j)
    cf = []
    for f in fm:
        if f in fn:
            cf.append(f)
    return(cf[-1])
```

Some explanations

Define a function to compute $\gcd(m, n)$.

```
def gcd1(m,n):
```

- gcd1 because this is the first of multiple versions that we will see

Go through each i from from 1 to m

```
for i in range(1,m+1):
```

- range(1,m+1) generates the list [1,2,...,m]

and update *fm* if i divides m

```
if (m%i) == 0:
    fm.append(i)
```

- m%i is the remainder of m divided by i, ($m \bmod i$, in mathematical notation)
- We use = for assigning a value, == to check equality

Checking for common factors

```
for f in fm:
    if f in fn:
        cf.append(f)
```

Factors are discovered in ascending order, so the largest common factor is the rightmost value in the list **cf**. Lists are indexed forwards from 0 and backwards from -1. The function returns the rightmost value in **cf**.

```
return(cf[-1])
```

Run the code and check that it works

In [2]:

```
gcd1(14,63)
```

Out[2]:

7

In [3]:

```
gcd1(837,775)
```

Out[3]:

```
31
```

Improving the brute force algorithm

Restricting the range to search for factors

- Factors of m are between 1 and m
- 21 and 63 are factors of 63 that cannot be factors of 14
- Enough to check for factors from 1 to $\min(m, n)$

Overlapping the search

- Two scans from 1 to $\min(m, n)$ to generate **fm** and **fn**
- Instead, check and update both **fm** and **fn** in a single scan

In [4]:

```
def gcd2(m,n):
    fm = []
    fn = []
    for i in range(1,min(m,n)+1):
        if (m%i) == 0:
            fm.append(i)
        if (n%i) == 0:
            fn.append(i)
    cf = []
    for f in fm:
        if f in fn:
            cf.append(f)
    return(cf[-1])
```

Still better

While scanning values from 1 to $\min(m, n)$ can directly identify common factors

- No need to separately generate **fm** and **fn** and then scan for common elements

In [5]:

```
def gcd3(m,n):
    cf = []
    for i in range(1,min(m,n)+1):
        if (m%i) == 0 and (n%i) == 0:
            cf.append(i)
    return(cf[-1])
```

Downloading and installing Python

- We will use Python 3
  - Latest version is 3.9.x, but any recent version will do
- Download from the official Python site
  - Python Software Foundation [https://www.python.org/ (https://www.python.org/)](https://www.python.org/)
- Anaconda distribution has a complete set of libraries to use Python for data science
  - Anaconda Individual Edition [https://www.anaconda.com/products/individual (https://www.anaconda.com/products/individual)](https://www.anaconda.com/products/individual)
  - Will automatically install Jupyter notebook, which is being used to generate these notes
    - Very useful to write/update code and also include documentation